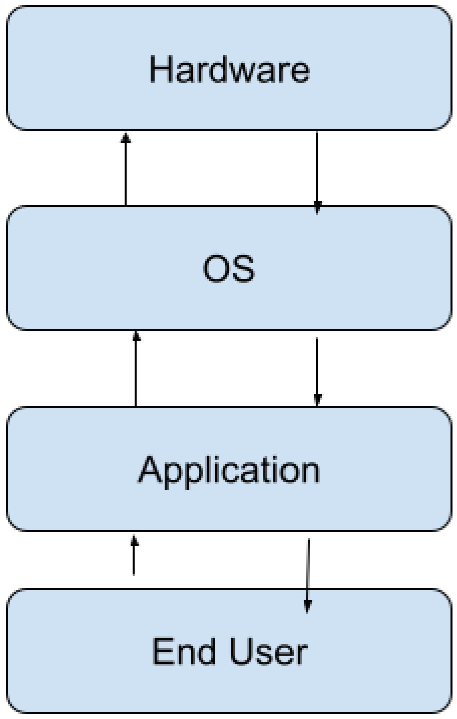
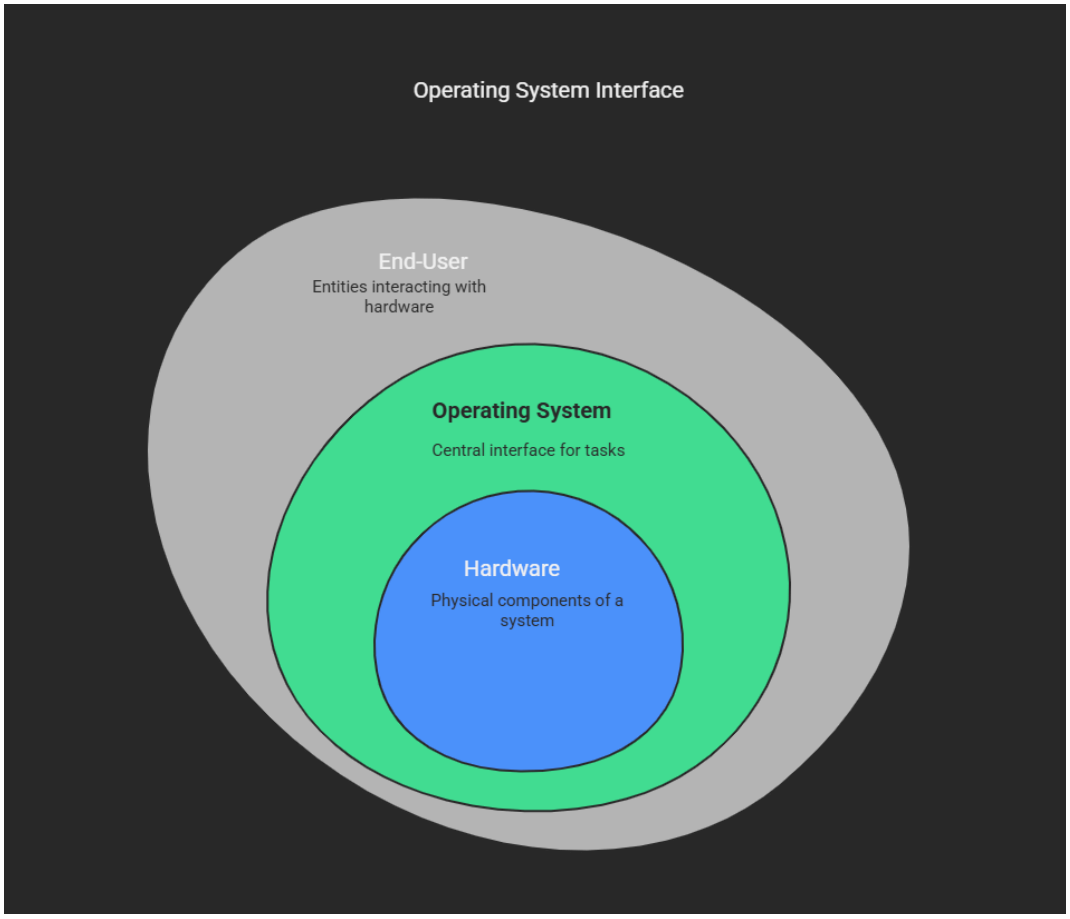
What is an operating System ?

This is an interface between hardware and end-user, where the end-user can be human, robot or any other software.
It provides communication between end-user and hardware to accomplish any task.

Ex: Linux, Windows, Mac OS, Ubuntu, Cent OS, Fedora.




Operating System Interface
End-User
Entities interacting with hardware
Operating System
Central interface for tasks
Hardware
Physical components of a system

Linux was developed by Linus Torvalds in 1991, he took this as his hobby project to work on the limitations of UNIX. He used the source code of MiNUX and released the first development under GNU-GPL.

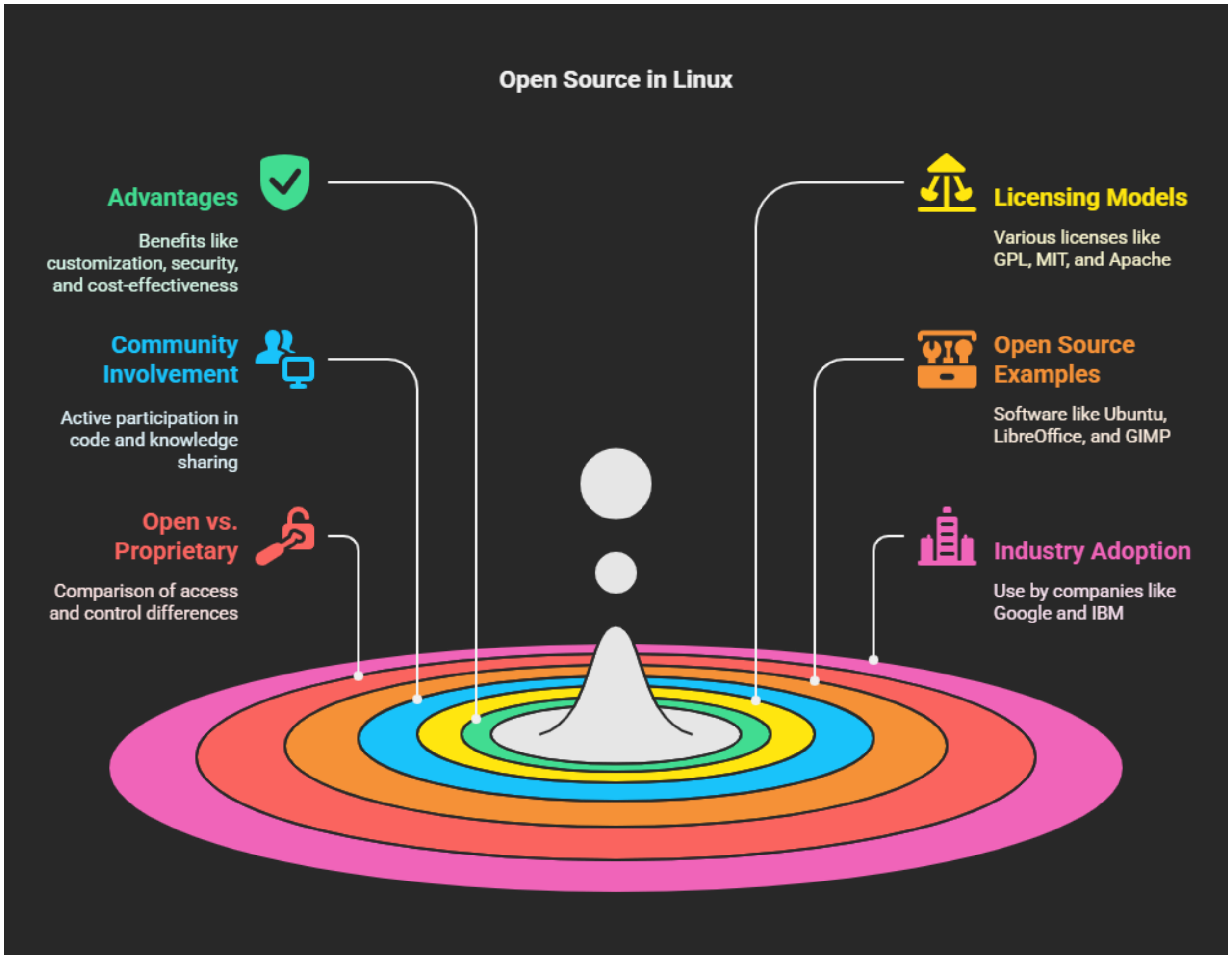
What do we understand by open source in Linux ?

In linux, we can have access to any source code and we have freedom to modify those codes. The codes are openly available to anyone who wants to examine it. This gives you freedom to understand how the software works, identify the bugs, suggest improvements or to contribute in any enhancement to the software.

As it is open source, it is typically free of cost and allows you to download, install and use without any upfront costs.

Open Source in the context of Linux refers to, licensing and distribution model of operating systems
Where the licensing under Linux runs under open source licenses such as GNU-General Public License(GPL) or License Approved by Open Source Initiative (OSI).

Open source in the context of Linux indeed involves specific licensing and distribution models that ensure the software's source code is accessible to everyone.


Open Source in Linux
Advantages
Benefits like customization, security, and cost-effectiveness
Community Involvement
Active participation in code and knowledge sharing
Open vs. Proprietary
Comparison of access and control differences
Licensing Models
Various licenses like GPL, MIT, and Apache
Open Source Examples
Software like Ubuntu, LibreOffice, and GIMP
Industry Adoption
Use by companies like Google and IBM

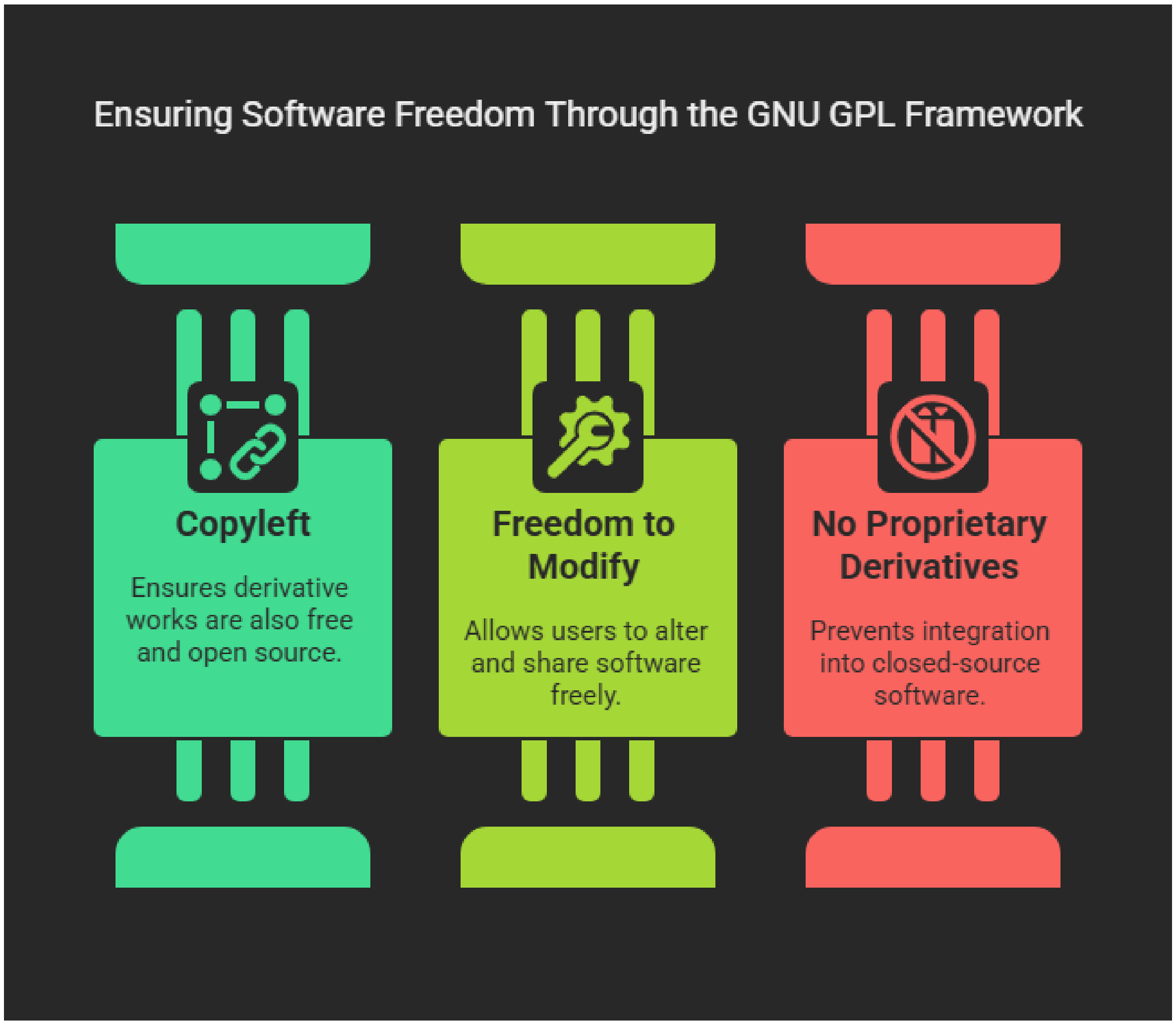
Licensing and Distribution Model in Linux:

- Licensing: Most Linux distributions are released under the GPL or other OSI-approved licenses, ensuring that users have access to the source code and the ability to modify and redistribute it.
- Distribution: Linux distributions (distros) can be freely distributed, and users can create and share their own versions of Linux, contributing to the rich diversity of the Linux ecosystem.

Here's a breakdown of the key components you mentioned:

GNU General Public License (GPL):

- Definition: The GNU General Public License (GPL) is a widely used free software license that ensures end users have the freedom to run, study, share, and modify the software.
- Key Features:
    - Copyleft: The GPL license is a "copyleft" license, meaning any derivative work based on GPL-licensed software must also be distributed under the same license. This ensures that the software remains free and open for all future users.
    - Freedom to Modify: Users can modify the source code and distribute their modifications, but they must also release the source code of their modified version.
    - No Proprietary Derivatives: GPL-licensed software cannot be integrated into proprietary (closed-source) software without violating the license.


Ensuring Software Freedom Through the GNU GPL Framework
Copyleft
Ensures derivative works are also free and open source.
Freedom to Modify
Allows users to alter and share software freely.
No Proprietary Derivatives
Prevents integration into closed-source software.

Open Source Initiative (OSI) Approved Licenses:

- Definition: The Open Source Initiative (OSI) is a non-profit organization that promotes and protects open source software by reviewing and approving licenses that meet their definition of "open source."
- Key Features:
  - Freedom to Use: OSI-approved licenses allow users to freely use the software for any purpose.
  - Freedom to Modify: Users can access the source code, make changes, and distribute those changes.
  - Freedom to Distribute: Users can distribute the software, whether it’s in its original form or with modifications, to anyone and for any purpose.
  - Variety of Licenses: OSI approves a variety of licenses, ranging from permissive ones like the MIT License, which allows proprietary use, to copyleft licenses like the GPL, which require derivative works to remain open source.

Open-source licenses come in different types, depending on how much freedom they give users and developers to modify and distribute the software. The Open Source Initiative (OSI) approves a variety of these licenses, which generally fall into two main categories:

1. Permissive Licenses (More Flexibility)

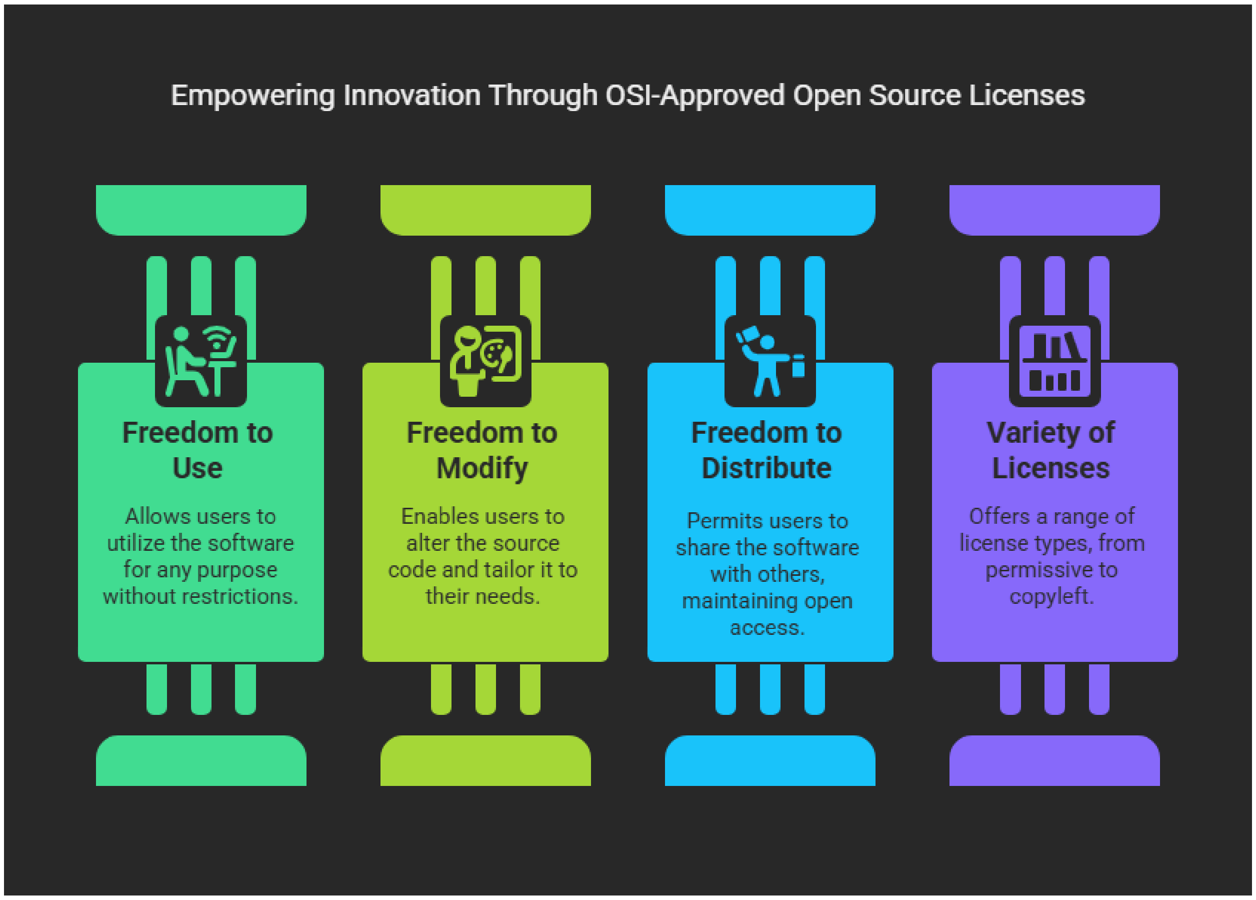
These licenses give developers the freedom to use, modify, and distribute the software without many restrictions. Even if someone includes the code in a closed-source (proprietary) project, they don’t have to share their changes.
Examples: MIT License, Apache License, BSD License.

2. Copyleft Licenses (More Restrictions)

These licenses ensure that any modifications or derived versions of the software must remain open-source. If someone modifies and redistributes software under a GPL license, they are required to share their modified source code under the same open-source license.
Examples: GNU General Public License (GPL), Lesser General Public License (LGPL).

Key Difference:

- MIT License (Permissive) → Developers can freely use the code in both open-source and commercial projects without any obligation to share their changes.
- GPL License (Copyleft) → Developers can modify and distribute the code, but any derivative work must also be open-source, ensuring that improvements remain accessible to everyone.

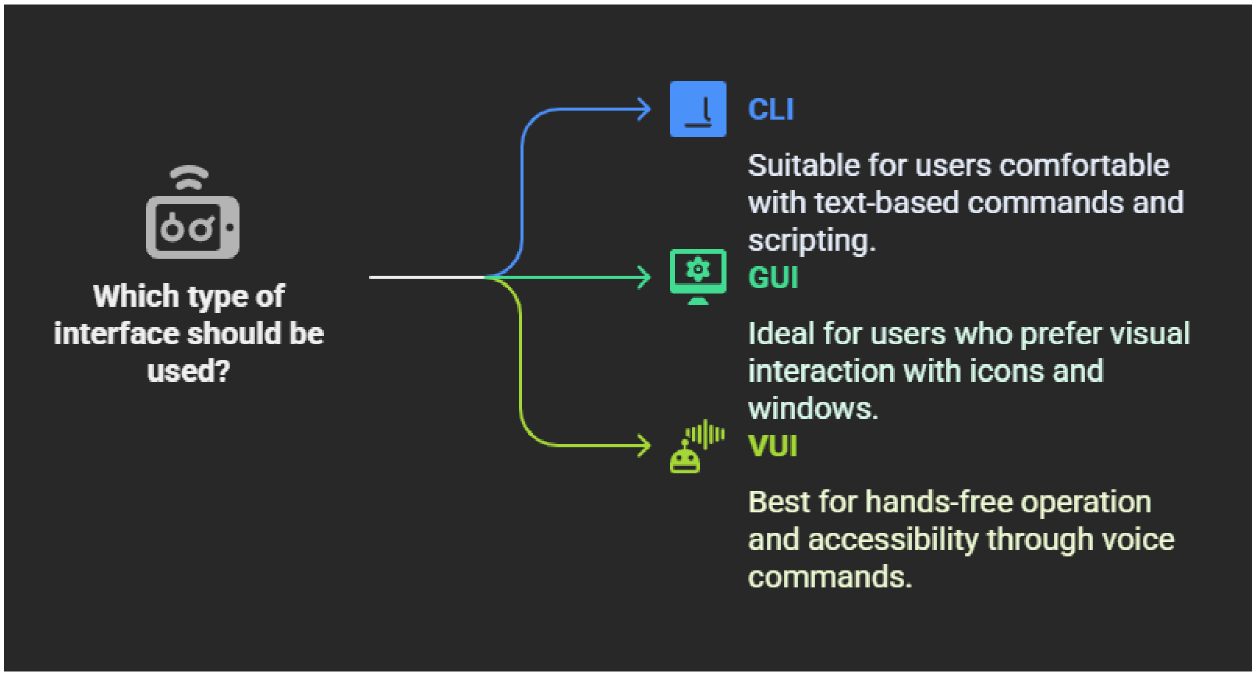

Empowering Innovation Through OSI-Approved Open Source Licenses
Freedom to Use
Allows users to utilize the software for any purpose without restrictions.
Freedom to Modify
Enables users to alter the source code and tailor it to their needs.
Freedom to Distribute
Permits users to share the software with others, maintaining open access.
Variety of Licenses
Offers a range of license types, from permissive to copyleft.

Although Linux is open source it more secure operating system

- There are n number of developers worldwide who are scrutinizing the code, checking the bugs and more often identifying and fixing it rapidly. The collaborative efforts ensure that the issues with the code are addressed promptly.
- As Linux is based on a permission model, that dictates who can access the file and who can execute the file. This kind of control allows administrator to restrict access to sensitive resources.
- Linux distribution provides you security updates that strengthen your system defenses and this protects against any emerging threats.

- Community Audits: Open source code allows global experts to continuously review and patch vulnerabilities.
- Permissions System: Linux has a strong user and permissions system, limiting unauthorized access.
- Modularity: The modular design of Linux allows for greater control over installed components, reducing security risks.
- Lower Target: Linux's smaller market share among casual users makes it a less frequent target for malware and attacks.

In OS we get two types of interfaces


Which type of interface should be used?
CLI
Suitable for users comfortable with text-based commands and scripting.
GUI
Ideal for users who prefer visual interaction with icons and windows.
VUI
Best for hands-free operation and accessibility through voice commands.

- CLI - Command Line Interface, this is a command based interface where a user types some command to execute the program.

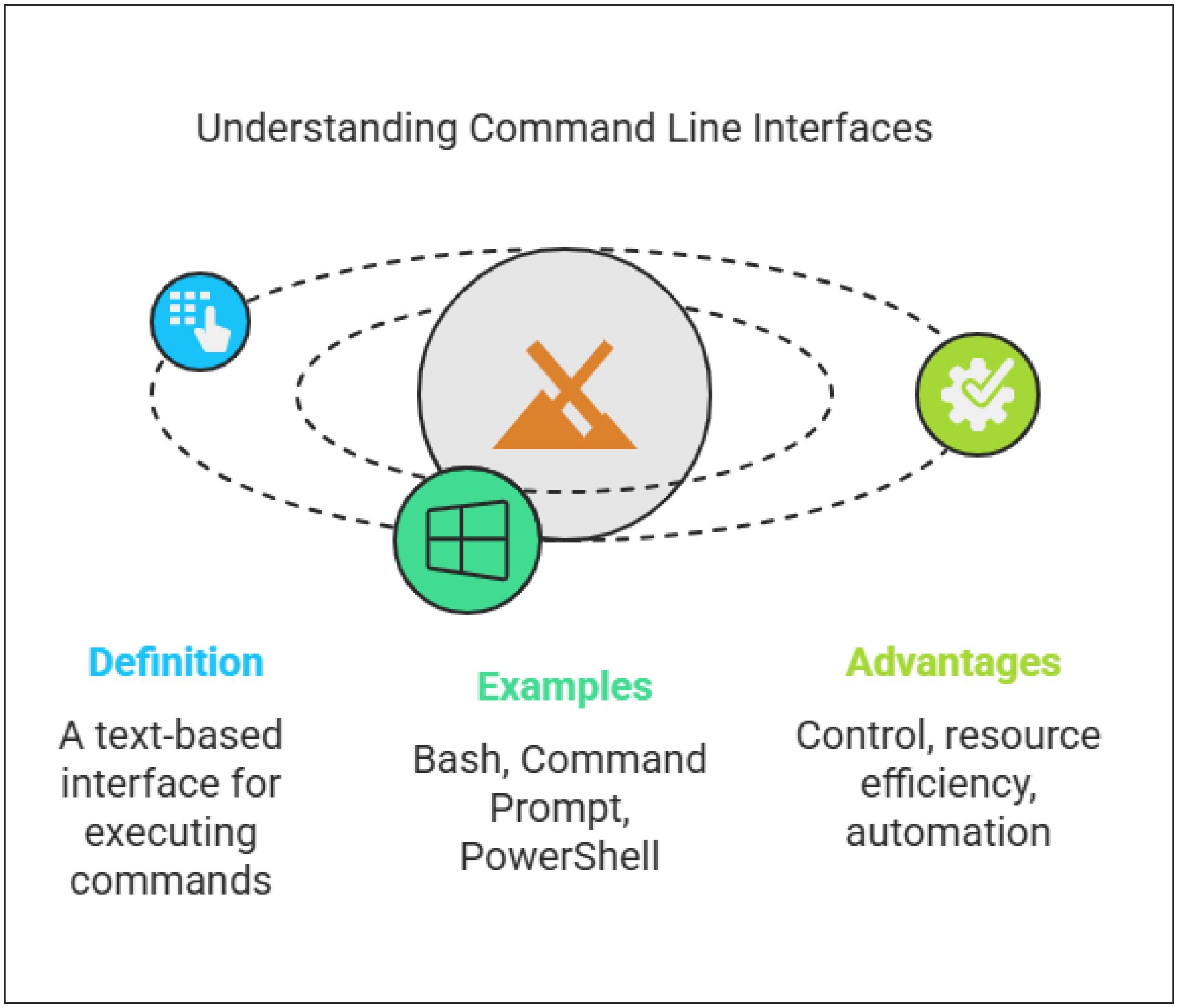
Definition: A text-based interface where users type commands into a terminal or console window to perform specific tasks.

Examples:

- Bash in Linux and UNIX systems.
- Command Prompt and PowerShell in Windows.

Advantages:

  - Offers more control and flexibility for system management tasks.
  - Consumes fewer system resources than GUIs.
  - Powerful for automating tasks through scripting.


Understanding Command Line Interfaces
Definition
A text-based interface for executing commands
Examples
Bash, Command Prompt, PowerShell
Advantages
Control, resource efficiency, automation

- GUI - Graphical User Interface, this is an icon based OS where a user can drag or click on the icons.

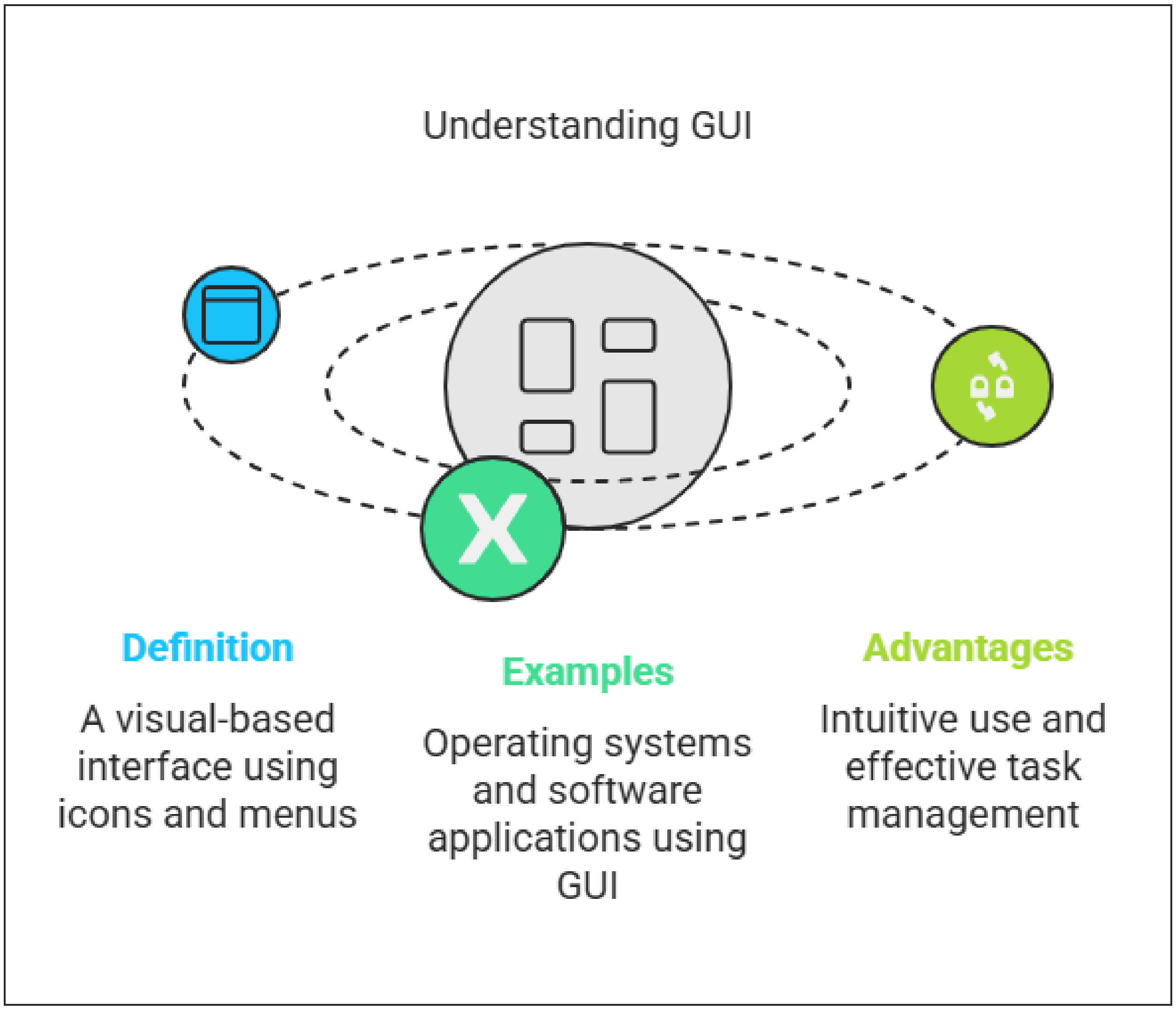
Definition: A visual-based interface that allows users to interact with electronic devices using graphical icons, visual indicators, and menus instead of text-based command labels or text navigation.

Examples:

- Operating systems like Windows and macOS use GUIs extensively.
- Software applications like Microsoft Word or Adobe Photoshop.

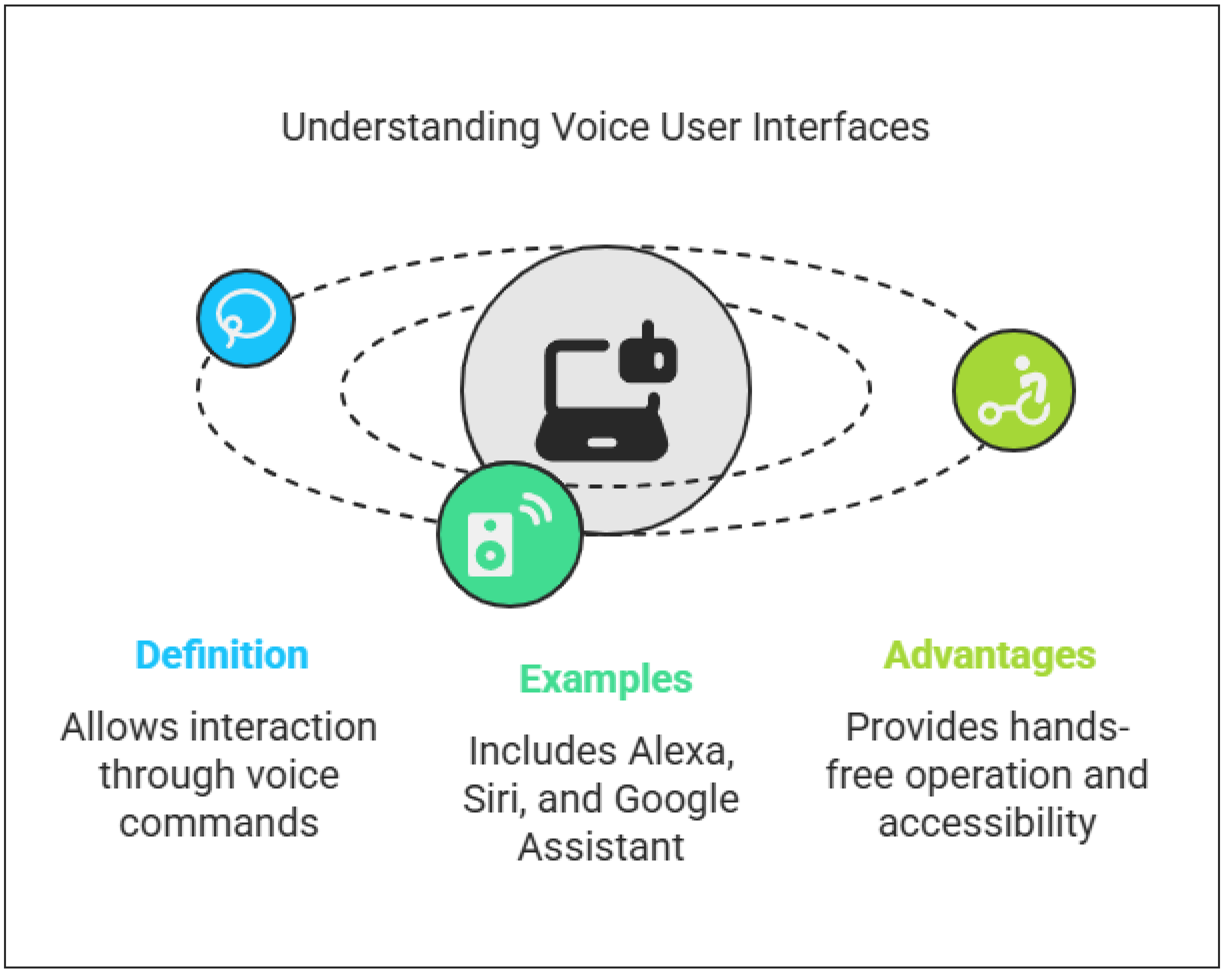
Advantages:

- Intuitive to use, especially for beginners.
- Enables effective management of multiple tasks simultaneously through multi-window and multitasking capabilities.


Understanding GUI
Definition
A visual-based interface using icons and menus
Examples
Operating systems and software applications using GUI
Advantages
Intuitive use and effective task management

Voice User Interface (VUI):

- Definition: Allows users to interact with a system through voice or speech commands.
- Examples:
  - Voice-activated assistants like Amazon Alexa, Apple Siri, and Google Assistant.
  - Voice-driven applications in smartphones and smart home devices.
- Advantages:
  - Hands-free operation makes it accessible and convenient.
  - Enhances accessibility for users with visual impairments or physical disabilities.
  - Useful in situations where using hands is impractical or unsafe, such as while driving.

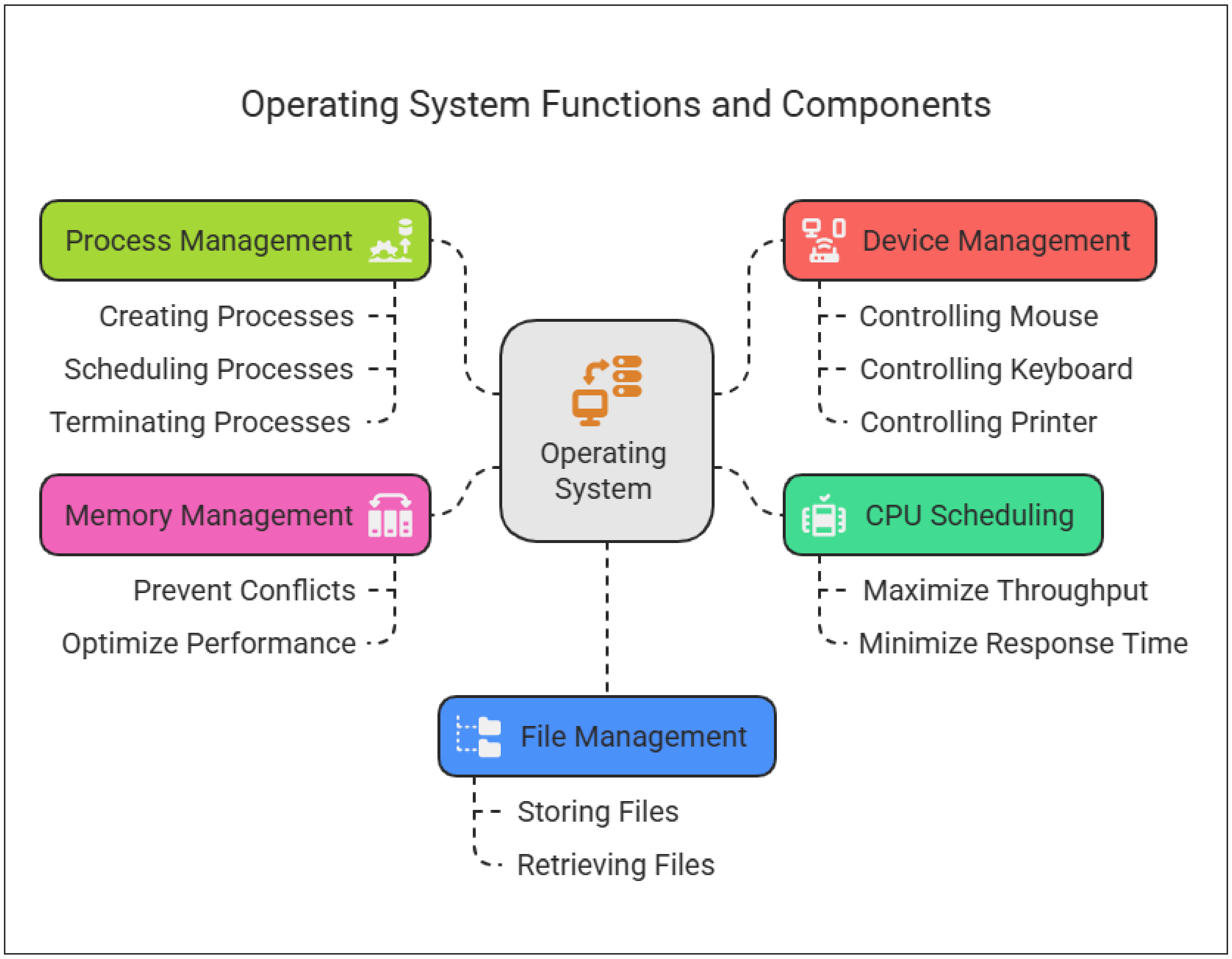

Understanding Voice User Interfaces
Definition
Allows interaction through voice commands
Examples
Includes Alexa, Siri, and Google Assistant
Advantages
Provides hands-free operation and accessibility

OS does

- CPU Scheduling - This determines the order in which processes are executed by the CPU, the primary goal of CPU scheduling is to maximize system throughput and minimize response time and to do allocation of CPU time among the competing processes.
- Process Management - This is a core component of OS, which is responsible for creating, scheduling, termination and managing processes and threads.
- Memory Management - This involves the efficient usage of computers memory, it ensures that programs and processes have access to memory they need while preventing conflict and optimizing system performance.
- File Management - This functionality involves, organizing, storing, retrieving and manipulation of files and directories on the storage devices such as Hard Drive, SSD or any network storage.

- Device Management - This involves controlling and coordinating access to hardware devices connected to the computer system, the devices which can be included are mouse, keyboard, printer, scanner or any other I/O devices.

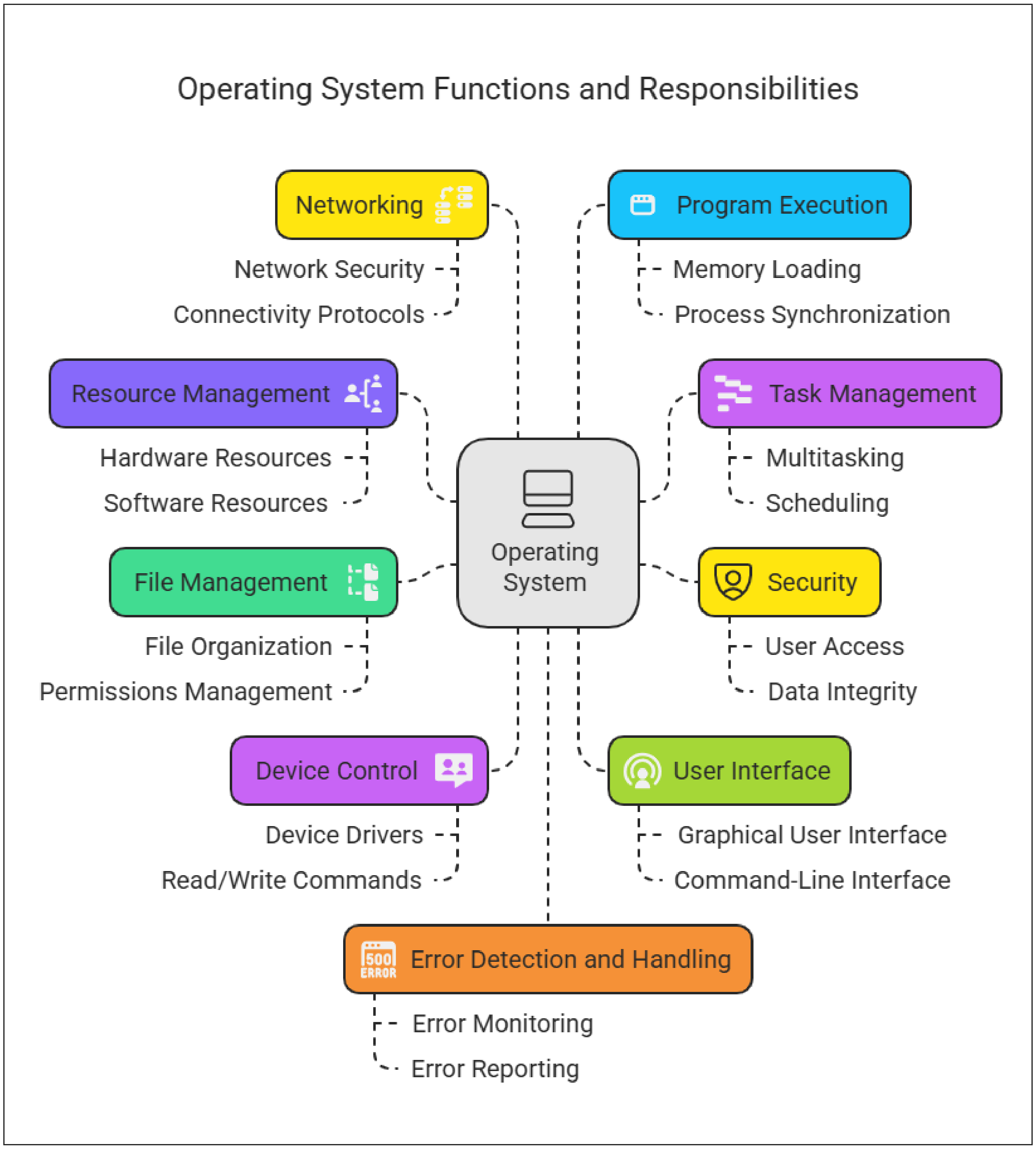
The contributions done by developers/ programmers are usually done on Linux Kernel such as to add features, finding and fixing bugs.



Here are the key functions of an operating system:

1. Resource Management:
    - Hardware Resources: The OS manages and allocates CPU time, memory space, disk space, and I/O devices. It ensures that different programs and users running concurrently do not interfere with each other.
    - Software Resources: The OS handles the execution of programs, provides necessary services to applications, and efficiently manages system resources among all running applications.
2. Task Management:

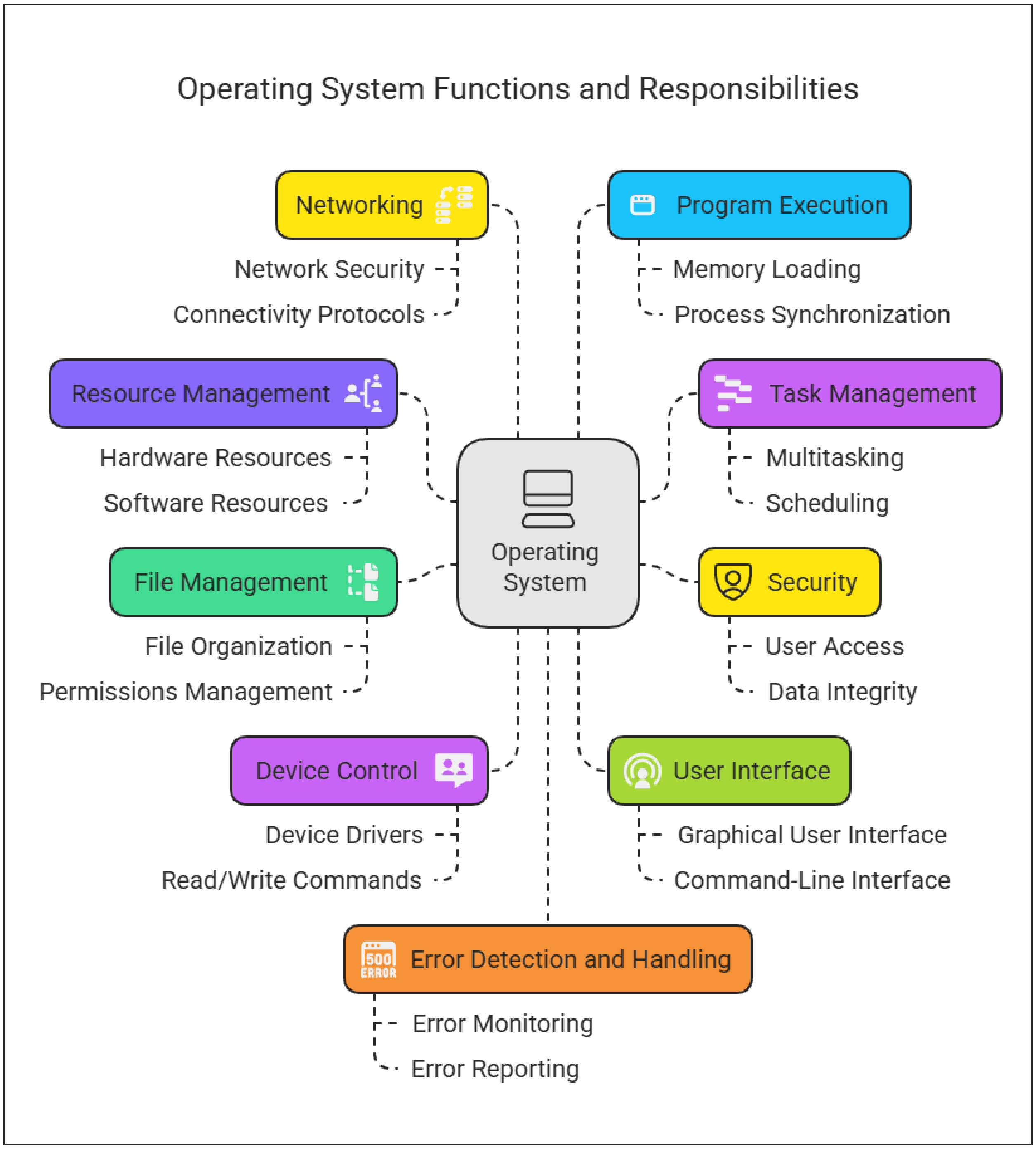
   - The OS is responsible for multitasking, which involves managing and scheduling the execution of multiple tasks. It keeps track of processor status, program status, and system functions.
3. Security:
   - An operating system provides a secure environment within which system resources and user data are protected. This includes managing user access to programs and data, protecting against unauthorized access, and ensuring data integrity.
4. File Management:
   - The OS manages files on various storage devices, organizes files in directories, and provides functions to create, move, delete, and access files. It also manages permissions and access rights for files and directories.
5. Device Control:
   - The OS manages all device communication via respective drivers. It translates the user's read/write commands into device-specific calls, acting as a communicator between the hardware and the software.
6. User Interface:
   - Operating systems provide interfaces through which users interact with computers. This can be a graphical user interface (GUI) like in Windows or macOS, a command-line interface (CLI) such as in Linux, or touch interfaces as seen in mobile operating systems like Android and iOS.
7. Networking:
   - The OS handles networking capabilities, allowing different computing devices to communicate and share resources over a network. It manages the data exchange, network security, and connectivity protocols.
8. Program Execution:
   - The OS loads programs into memory, sets up the execution stack, and handles the execution of applications. It provides mechanisms for process synchronization and inter-process communication.
9. Error Detection and Handling:
   - The operating system continuously monitors the system for possible errors. It provides error-reporting methods to the users and takes appropriate actions to recover from system errors.



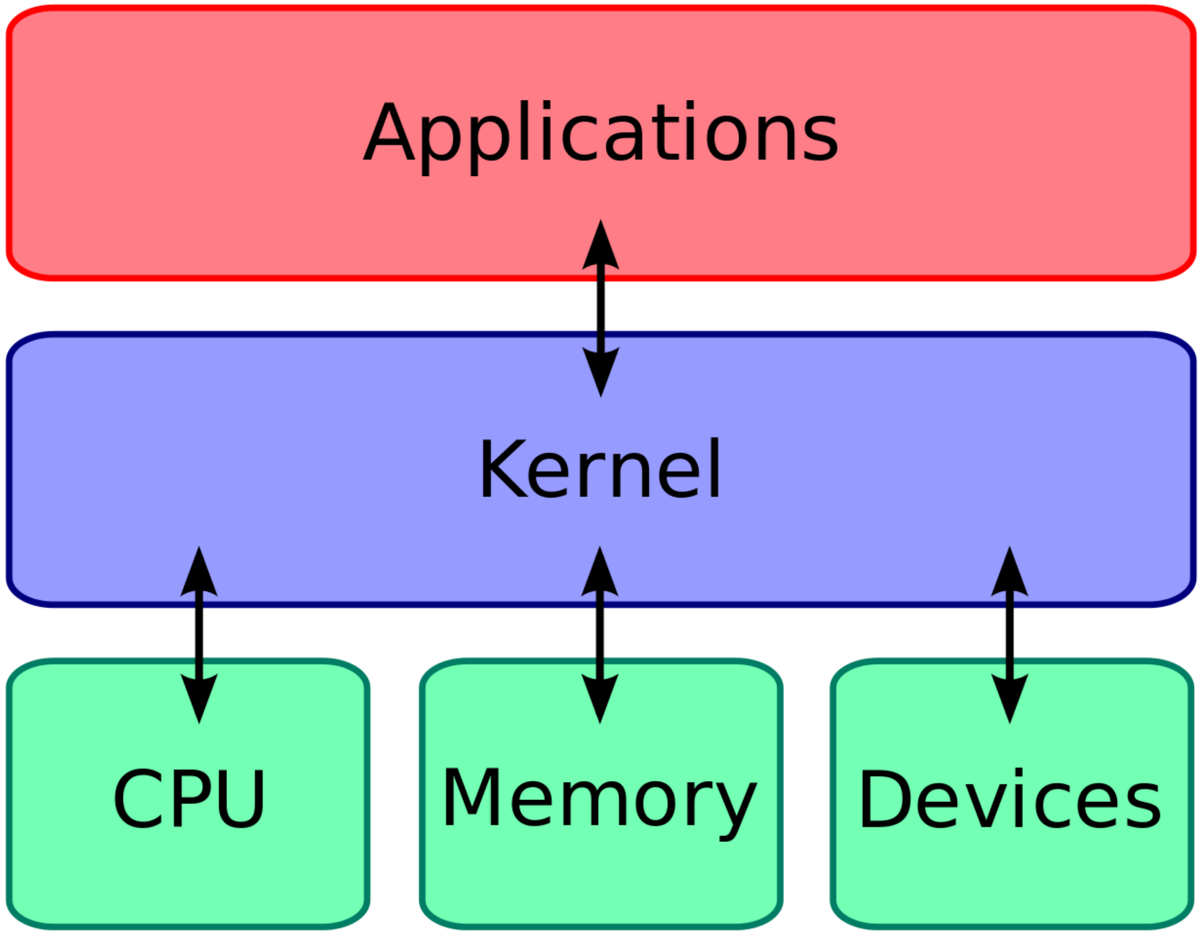
Here are the key functions of an operating system:

1. Resource Management:
    - Hardware Resources: The OS manages and allocates CPU time, memory space, disk space, and I/O devices. It ensures that different programs and users running concurrently do not interfere with each other.

    - Software Resources: The OS handles the execution of programs, provides necessary services to applications, and efficiently manages system resources among all running applications.
2. Task Management:
    - The OS is responsible for multitasking, which involves managing and scheduling the execution of multiple tasks. It keeps track of processor status, program status, and system functions.
3. Security:
    - An operating system provides a secure environment within which system resources and user data are protected. This includes managing user access to programs and data, protecting against unauthorized access, and ensuring data integrity.
4. File Management:
    - The OS manages files on various storage devices, organizes files in directories, and provides functions to create, move, delete, and access files. It also manages permissions and access rights for files and directories.
5. Device Control:
    - The OS manages all device communication via respective drivers. It translates the user's read/write commands into device-specific calls, acting as a communicator between the hardware and the software.
6. User Interface:
    - Operating systems provide interfaces through which users interact with computers. This can be a graphical user interface (GUI) like in Windows or macOS, a command-line interface (CLI) such as in Linux, or touch interfaces as seen in mobile operating systems like Android and iOS.
7. Networking:
    - The OS handles networking capabilities, allowing different computing devices to communicate and share resources over a network. It manages the data exchange, network security, and connectivity protocols.
8. Program Execution:
    - The OS loads programs into memory, sets up the execution stack, and handles the execution of applications. It provides mechanisms for process synchronization and inter-process communication.
9. Error Detection and Handling:
    - The operating system continuously monitors the system for possible errors. It provides error-reporting methods to the users and takes appropriate actions to recover from system errors.



Kernel
This is a core part to interact between hardware and software. It is a core part of OS, that interacts directly with hardware
Kernel is responsible for managing CPU, memory, I/O devices that handles system calls, interrupt or any exceptions.



Memory Management:

- The kernel controls and manages system memory. It decides how much memory to allocate to processes and when to reclaim it. It also handles memory swapping between the physical RAM and the hard disk to optimize system performance.

Process Management:

- The kernel is responsible for process scheduling and lifecycle management. It controls process creation, execution, pausing, resuming, and termination. It manages CPU resources by deciding which processes get CPU time, in what order, and for how long.

Device Drivers:

- Device drivers are programs that allow the kernel to interact with hardware devices, like keyboards, mice, disk drives, network adapters, etc. The kernel uses these drivers to abstract the hardware details away from the software, providing a consistent interface to peripherals.

System Calls and Security:

- System calls are the mechanisms by which programs request services from the kernel, such as creating processes, handling files, or communicating over a network. The kernel also enforces security policies and isolates different system processes to prevent unauthorized access to data.

Input/Output Management:

- The kernel manages I/O operations and provides a buffer management system to control the flow of data to and from hardware devices. It optimizes these operations to improve system efficiency and responsiveness.

Networking:

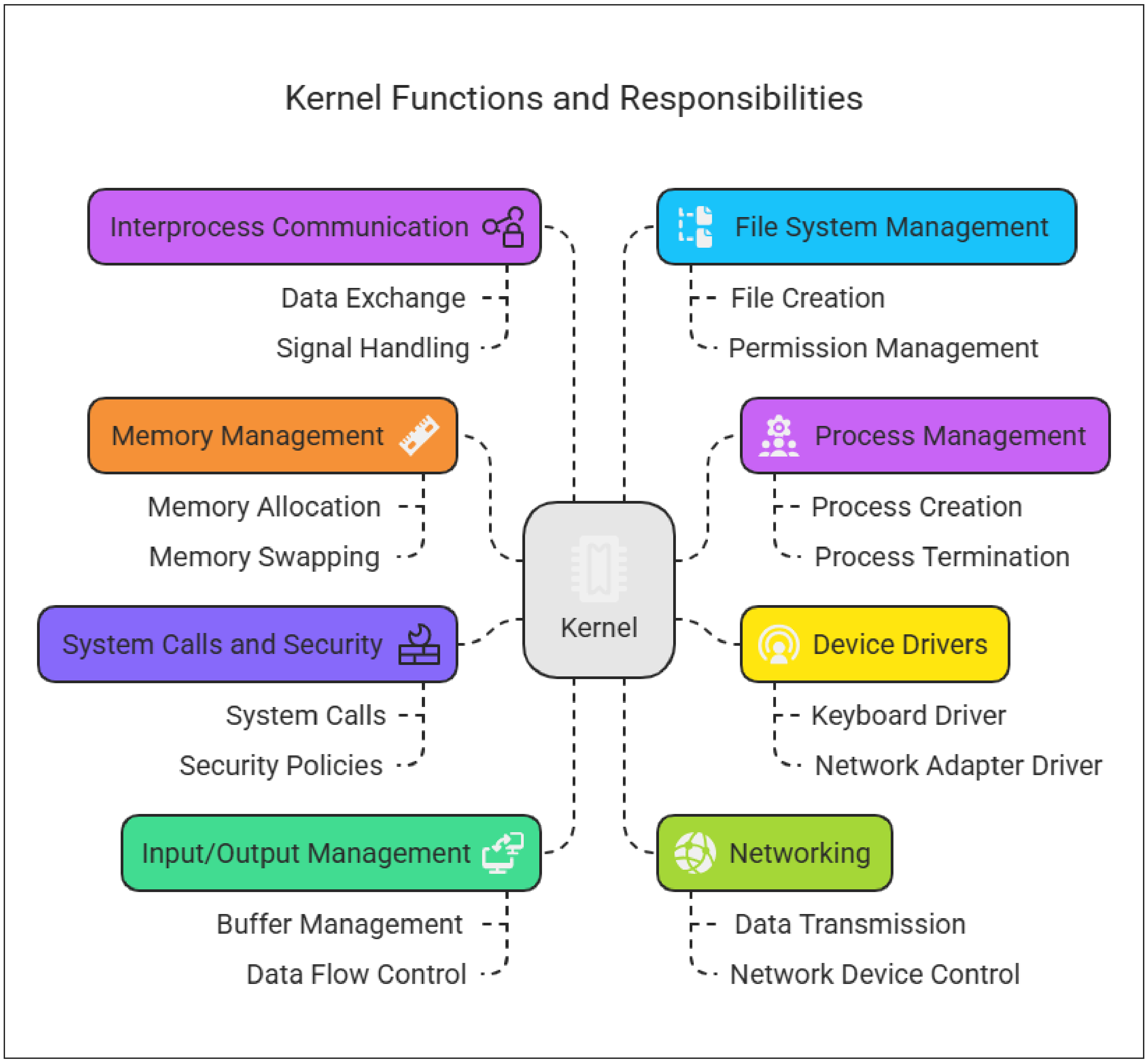
- The kernel handles the networking stack that allows the system to communicate over various network protocols, handling both the data transmission and the network device control.

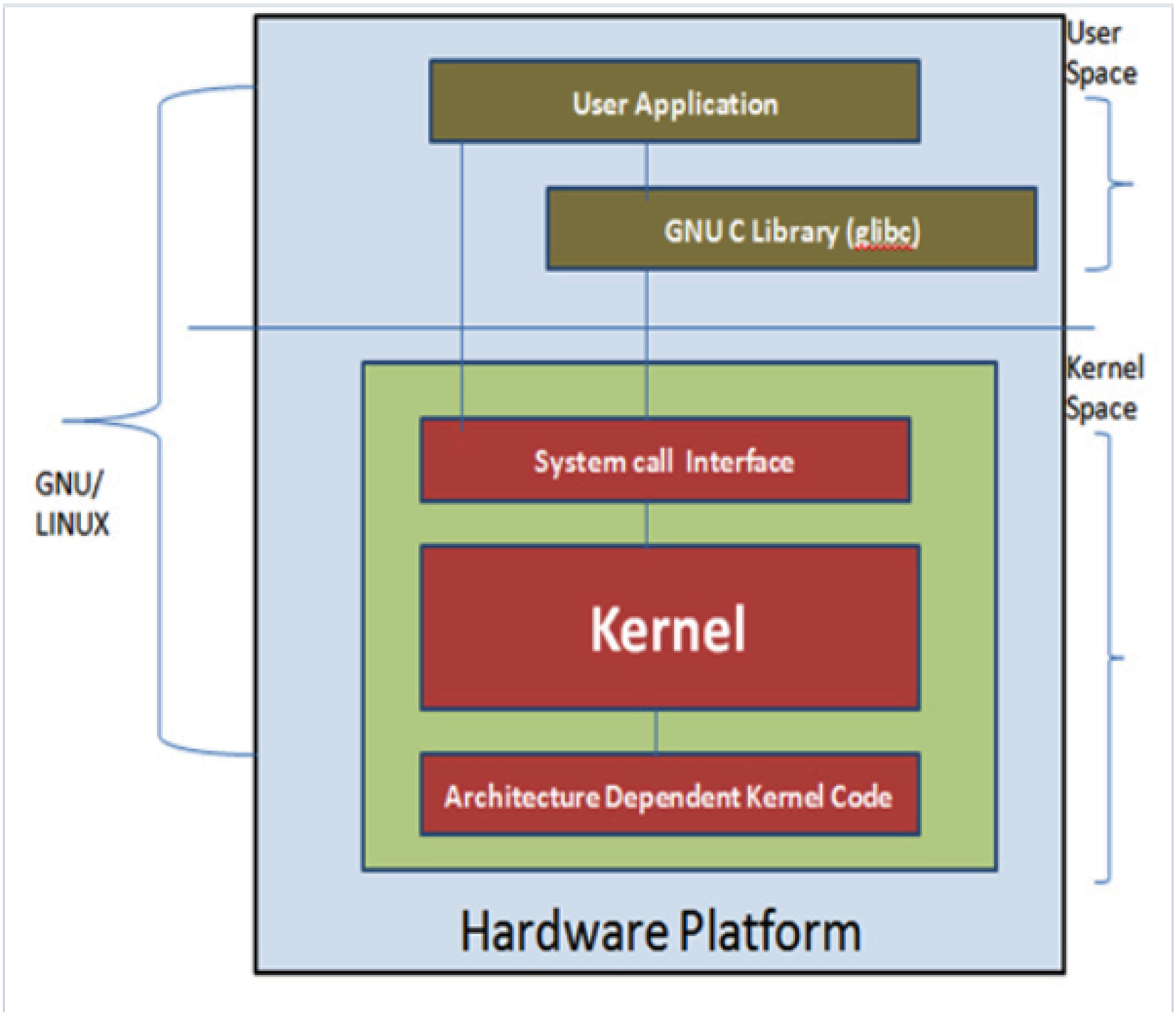
Interprocess Communication:

- The kernel facilitates communication between processes, allowing them to exchange data and signals securely and efficiently. This is crucial for applications that need to work together closely to perform their tasks.

File System Management:

- It manages file systems, which organize and store the files on various storage devices. The kernel provides functions to create, delete, read, and write files and to manage permissions and security settings.


Kernel Functions and Responsibilities
Interprocess Communication
Data Exchange
Signal Handling
File System Management
File Creation
Permission Management
Memory Management
Memory Allocation
Memory Swapping
Process Management
Process Creation
Process Termination
Kernel
System Calls and Security
System Calls
Security Policies
Device Drivers
Keyboard Driver
Network Adapter Driver
Input/Output Management
Buffer Management
Data Flow Control
Networking
Data Transmission
Network Device Control



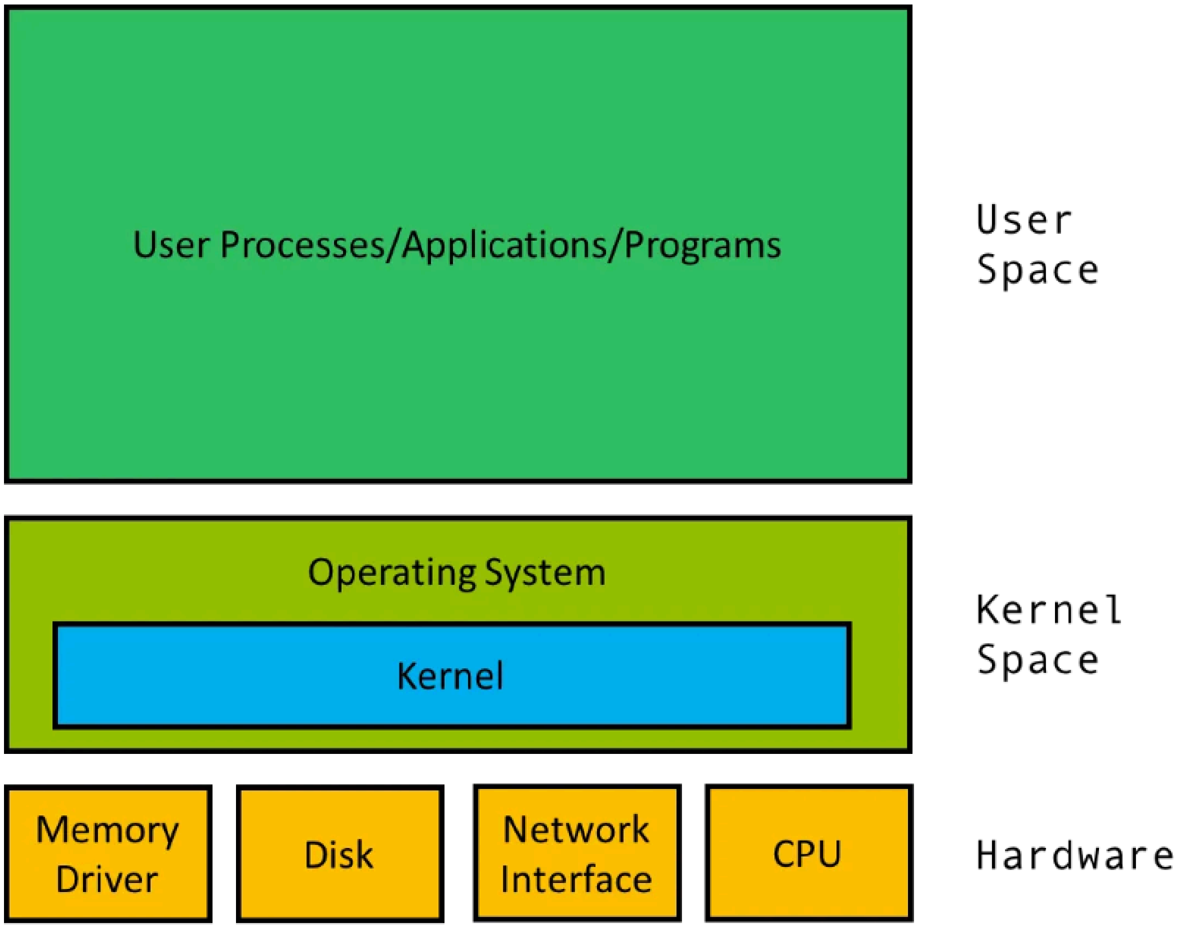
User Space

- Definition: User space (or user mode) is the region of system memory where user applications and processes run. This space is protected and isolated from the kernel space, meaning that code running here cannot directly interact with the hardware or interfere with kernel operations.

- Characteristics:
    - Security and Stability: Errors or crashes in user space do not affect the core of the operating system, which helps maintain overall system stability and security.
    - Limited Access: Applications in user space have limited access to system hardware and resources, mediated by the kernel through system calls.
    - Environment: User space provides the environment where third-party and user applications run, such as web browsers, games, and office software.

Kernel Space

- Definition: Kernel space (or kernel mode) is where the kernel (the core part of the operating system) operates. It has unrestricted access to the hardware and all system resources.
- Characteristics:
    - Full Access: Code running in kernel space can directly interact with hardware devices, manage memory, and perform critical system tasks.
    - Performance: Operations in kernel space are faster than those in user space because there's no need for the context switch typically required to access hardware.
    - Risk: Errors or crashes in kernel space can lead to system failures or security issues since the kernel has complete control over the system.



Interaction Between User Space and Kernel Space

- System Calls: The primary method of interaction between user space and kernel space is through system calls. Applications in user space request services from the kernel (like file access, starting a new process, or network communication) using these calls.

- APIs and Libraries: User applications typically use APIs and libraries (like the GNU C Library in Linux) that abstract these system calls into more manageable functions for programming.

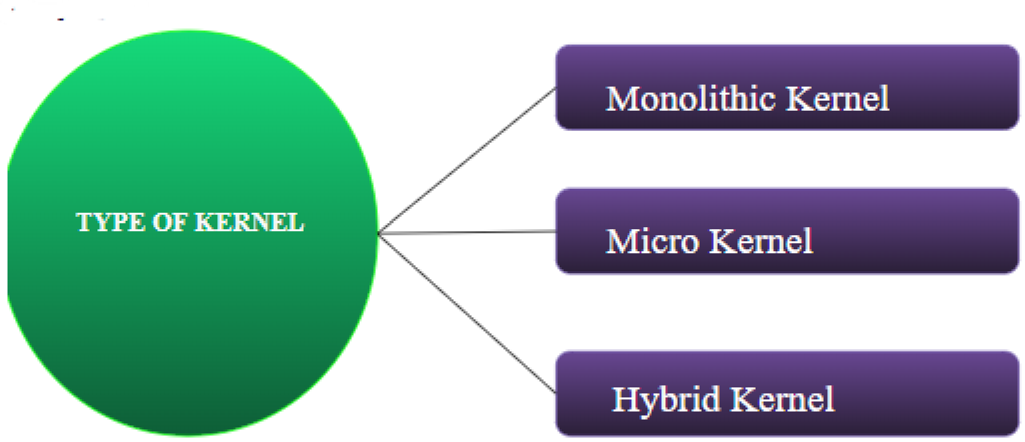
The separation between user space and kernel space is fundamental for system security and reliability. It ensures that malfunctioning programs or malicious software in user space cannot corrupt the kernel or other running applications, thereby protecting the integrity and stability of the operating system.

Kernel loads first into memory, when an OS is loaded and remains into memory until OS is shut down. The kernel has a process table that keeps track of all active processes.Kernel is written in C language.

The difference between Kernel and OS:

Kernel	OS
It is a core part of the OS.	It is an interface between hardware & software.
It is a software / low level program that manages hardware resources including memory, CPU & I/O devices, interrupt and exceptions.	OS not only includes Kernel but various system utilities, libraries and various user interfaces.
It is responsible for doing tasks like process management, memory management , scheduling, hardware management.	Is an interface that helps to interact between user interface.
It is a central point of control and resource management in OS.	It controls, manages system configuration files or tools.

Types of Kernel
There are 3 types of kernel



Monolithic Kernel

- These types of kernel where all OS services operate in kernel space, this is one of the oldest types of kernel where the entire OS is composed in a single large executable file that runs on kernel mode.
- This is more efficient kernel as it does not have to switch between different address spaces, when executing different tasks. This makes the process execution faster as there is no separate memory space for user and kernel.
- All operating system services run in kernel space, which can lead to better performance but may result in less stability due to the complexity of the kernel.
- Examples: Linux, Unix.

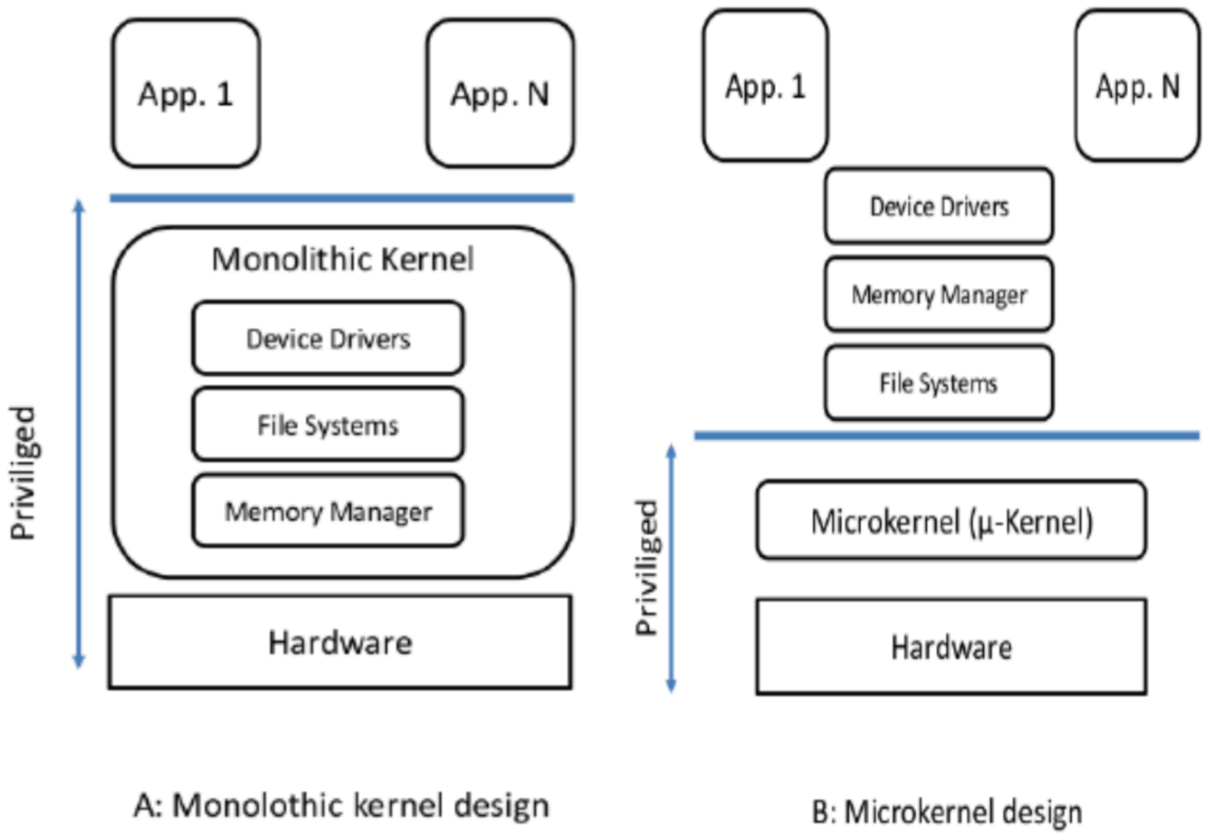
Advantages of Monolithic Kernel

- It is faster because it operates from kernel space.
- It is more secure than any other kernel because it can be protected easily from malicious code.

Disadvantages

- As the entire OS is composed of a single binary executable file that runs on kernel mode so if anything gets corrupted the whole system gets corrupted.

Examples : UNIX, LINUX, Open VMS



Micro-Kernel
It is a type of kernel which works between two spaces i.e, user space and kernel space. Micro Kernel works on modularity i.e system services, device drivers run on separate space i.e user space and all the process management, memory management will work under kernel space. The kernel is minimized, and most services run in user space. This approach can improve stability and security but might incur a performance penalty.
Examples: MINIX, QNX.

Advantages of Micro-Kernel
It is more stable than Monolithic Kernel, if any services get affected or corrupted it doesn't affect the whole system.
Examples: Blackberry QNX, MiNIX

Disadvantages
- As it is modular it requires more communication and synchronization between different spaces.
- It uses more system resources such as CPU, Memory than monolithic kernel as it has to maintain synchronization.

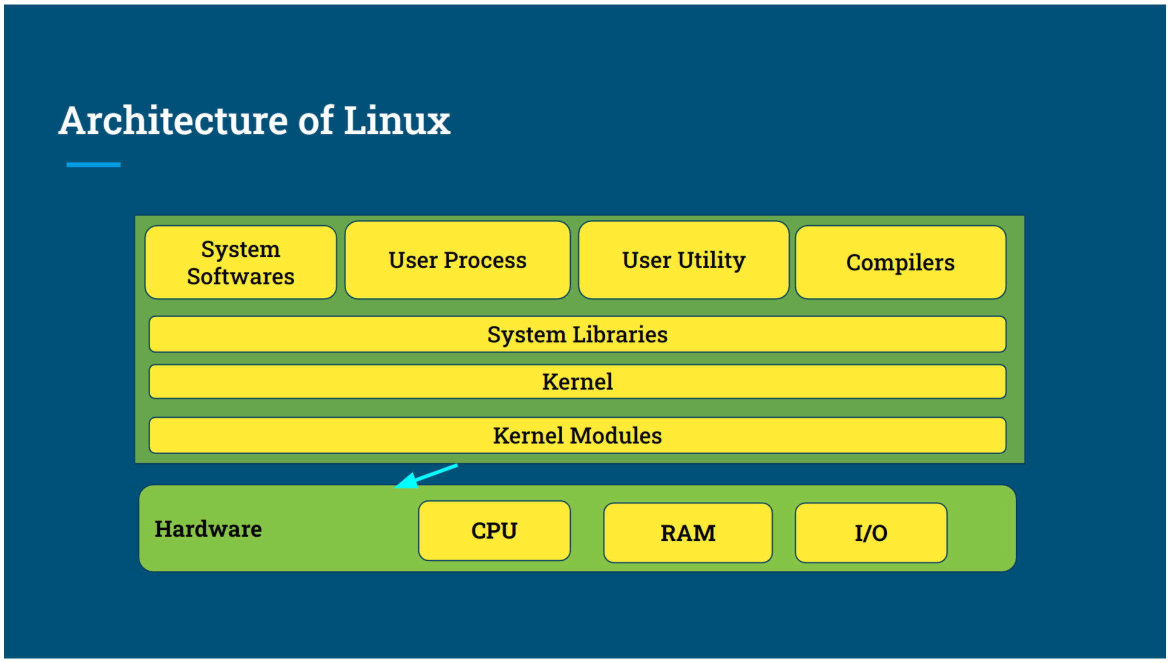
Hybrid Kernel
This is a combination of monolithic and micro kernel, it has speed of monolithic kernel while it has modularity and stability of micro kernel. All the essential services like process management, file management, CPU management, CPU scheduling, interrupts are kept in Kernel mode while services like device drivers are placed under User mode.
Combines aspects of both monolithic and microkernel architectures, trying to balance performance and modularity.
Examples: Windows NT, macOS.

Examples : MS Windows, MAC OS, Netware.

Difference between Micro and Monolithic Kernel

Micro Kernel	Monolithic Kernel
User services and kernel services are kept in separate spaces	Both kernel and user services are kept in same space
Smaller than Monolithic Kernel	They are larger than micro-kernel
It is easier to add new functionality as it works in modularity	It is difficult to add new functionality as everything is there in one space
Failure of one component doesn’t affect the whole system	Failure of any one component affects the whole system
Execution speed is slower	Execution speed is faster
Ex : Blackberry QNX	Ex : UNIX

Architecture of linux:



Hardware

CPU - It is a central processing unit which is responsible for executing and controlling any process or task on the system.
RAM - It is said as Random Access Memory that helps to store data temporarily
I/O Devices - Input devices are those devices which help you to enter data or commands into the system for example - Keyboard, Mouse, Scanner.
Output devices are those devices which help you to display the output from the system for example - Monitor, Printer, Speaker

Kernel Modules

It is also known as device driver or also referred to as loadable kernel .
It is called as loadable and unloadable kernel because a piece of code can dynamically be loaded into linux kernel without rebooting your system. These are basically written into C / Assembly language
Examples - Graphic Card, Network Adapter.
Kernel modules are stored in files with extension of '.ko' and are loaded into kernel using 'INSMOD' or using 'MODPROBE' commands.
The main subdirectory for kernel modules is found under '/lib/modules'.

Kernel

It is a core part of the OS that exists between the hardware and software, whenever a system boots kernel is the first part that gets memory and exists till the system shuts down. It is responsible for managing CPU memory, I/O devices that handles system calls, interrupts and any exceptions.

System Libraries

They are prewritten codes that provide functionality to programs and applications running on OS. It is a set of functions for user level applications to interact with OS Kernel.
Examples - C Standard Library (LibC) this library is used to perform operations like I/O, Memory management and string manipulation,
Math Library (LibM) provides a wide range of mathematical functions that includes Trigonometry, Logs.
Database Connectivity (Libpq, LibMySQLClient) These libraries are used to interact with various databases on your system.
System Call Wrappers this library is used to make system call to interact between user level program and kernel
Example - printf

User Utility

It provides various functionality tools to user and system administrator to interact and perform various tasks on Linux
Example - Shell (Bash, CSH, ZSH), Shell command is used to interact with system by entering some commands
Text Editor this is used to create and edit a text file.
Example - Vim, Nano, Touch

File & Directory Management, this helps you to manage file and directories including managing files by commands like cp, rm, ls, mkdir

Process Management allows you to view and manage the process with commands like ps,kill,top.

Package Manager this helps you to install and update system software, Examples - apt, yum, pacman.

File Compression this helps you to compress and decompress any file. Examples - Zip, Unzip, Tar

Remote Desktop this utility allows you to remotely access any other system Examples - SSH PUTTY

Networking Tools this helps you check the network using ping,ss commands

User Processes

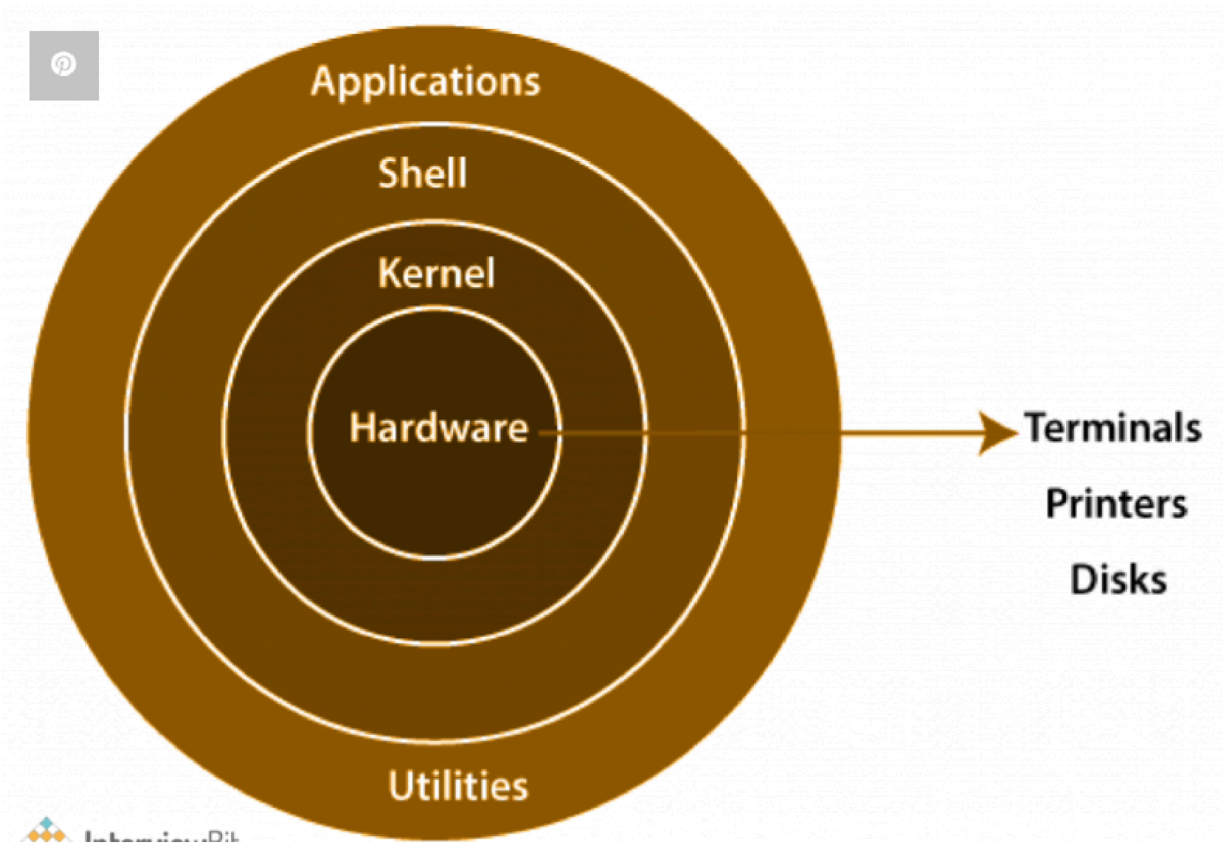
These are those programs that run in user space which are protected and isolated areas of memory and execute in separate space. They are initiated by user to interact with kernel and system resource through system call following are the different modules through which user processes work :

- User Application - User processes consist of user application which can either be graphical application or command line utility to interact with the system.
- User Space - User processes that operate within user space, it is a protected environment where processes are given limited and direct access to hardware resources.
- Process Creation - Processes are created by users from another parent process fork() ,exec() are system calls that are used to create and replace any processes.
- Resource Management - Processes interact with kernel to manage various resources like CPU Memory, I/O.
- System Calls - User processes interact with kernel by making system calls these calls request services / operation from kernel, few system calls are like creating new processes, allocating memory, doing I/O operations, creating new files and reading files.

System Software

It refers to collection of software and utilities that are required to manage operation on OS Examples -

- Init - System is responsible for initializing, managing system services and handling system shutdown process, this is responsible for initializing OS after it loads into memory.
- Bootloader - This is responsible for loading OS into memory, it is the first program that is executed whenever you turn on the system.



User space this is a layer where all the user level applications such as GUI, CLI falls. This layer helps to interact with the kernel through system calls to perform various tasks.

Kernel space provides an interface between hardware and software that manages system resources.

Shell

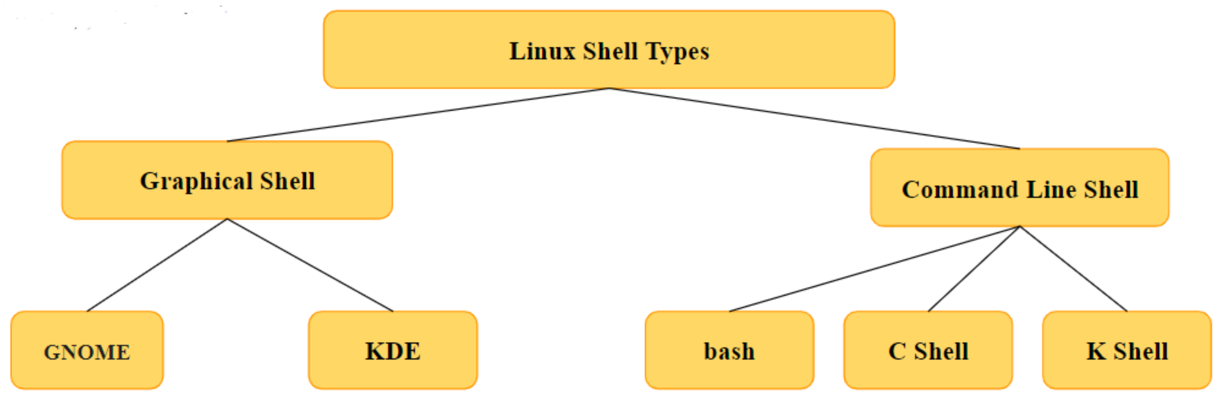
It is a command line interface that allows you to interact with kernel and user. It is a program that interprets user input and executes the commands or we can say it is a program that executes any other program.
Following are the tasks that are performed by shell :

- User can execute the command
- Shell provides I/O redirection which allows the user to redirect I/O from the command to and fro from the file.
- This manages system and environment variables using shell ,users can modify system variables.

- Shell allows the user to write a script that can be executed as a single unit.(Script is a collection of commands).

Shell is divided into two parts

- Graphical - This shell specifies the manipulation of a program using a graphical interface that provides operations like moving, closing, resizing or switching between applications.
- Command Line - It is a program that provides command line interface for interacting with OS. It allows users to enter any command on prompt and executing.



Path

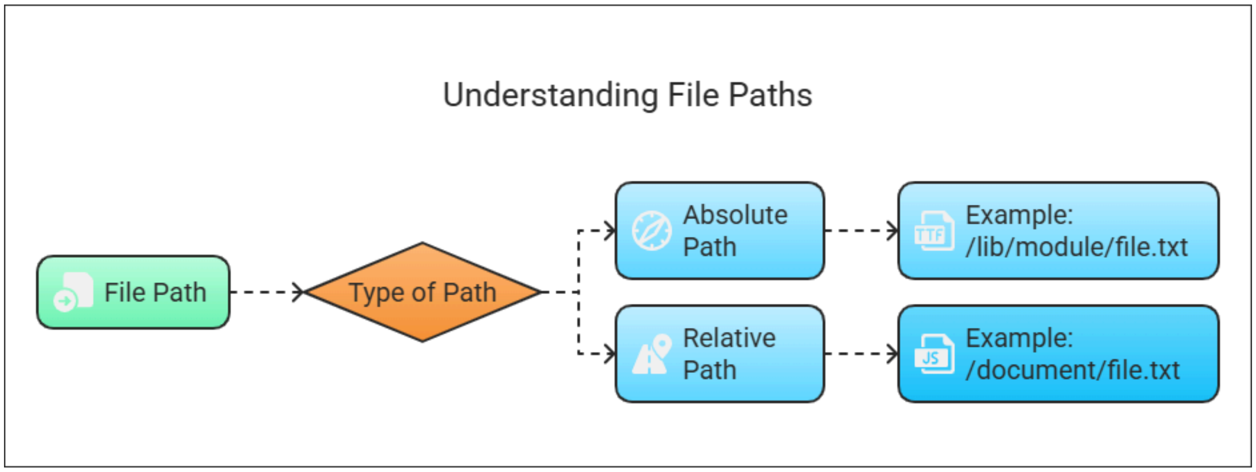
It is a unique location to a file / folder in a file system. A path of a file can be a combination of '/' and alphanumeric characters.
Example : /lib/module/file.txt (where lib is a parent directory and module is a sub directory inside lib and file.txt is file inside module)

There are two types of path

- Absolute Path - This specify location of file and directory from root directory in other way we can say absolute is a complete path from start of actual file system from root(/) directory.
  Example - /lib/module/file.txt

- Relative Path - It is defined as path related to present working directory, it starts from your current directory .

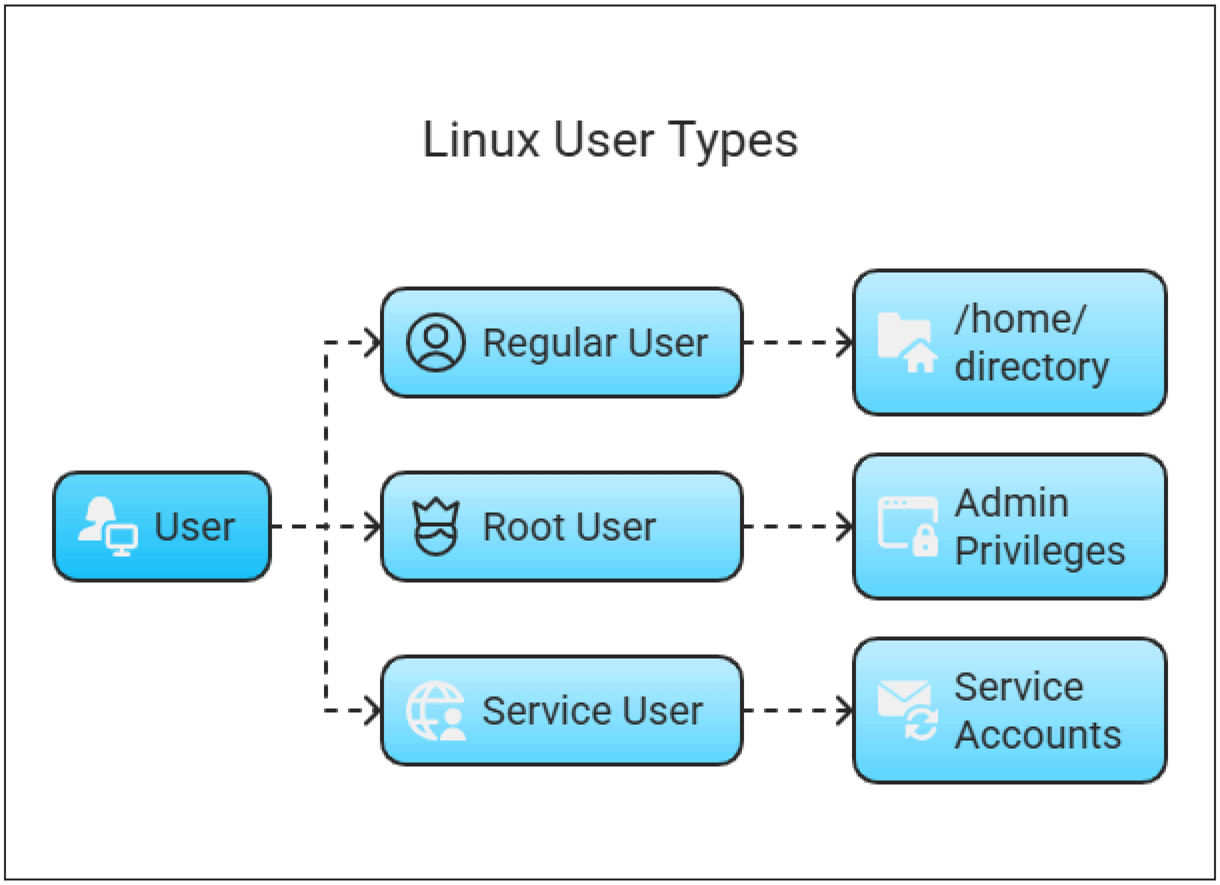
Example - Suppose your file is located at ‘ /home/usr/document/ ‘, your file is located at document directory in relative path it will be referred as ‘ /document/file.txt’



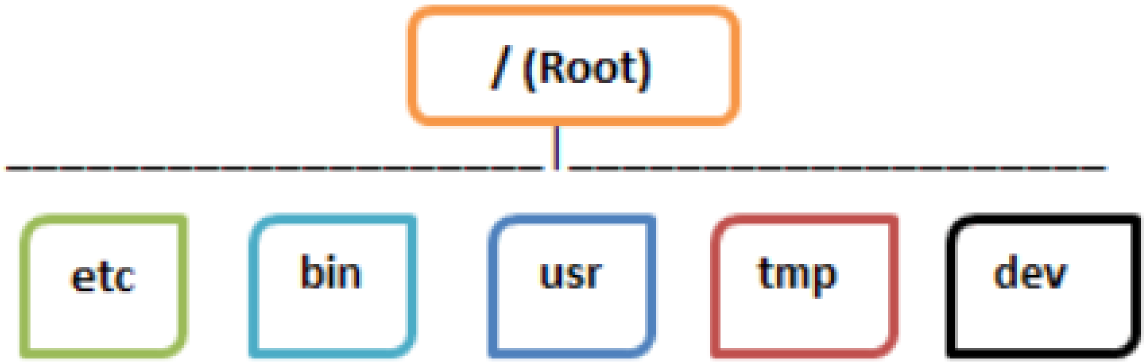
Users in Linux

User is referred as individual who is interacting with the OS to perform various tasks, there are 3 types of users

- Regular User: These are those users which are created during installation of Linux in this all the files and folders are stored in /home/ directory and they can not have access to directories of any other user. Their symbols is ‘$’ and can also be ‘%’
- Root User: They are the super users that have all the admin privileges. Its symbol is ‘#’
- Service User: This is widely used as server OS and services like e-mail and other applications have their own services account.

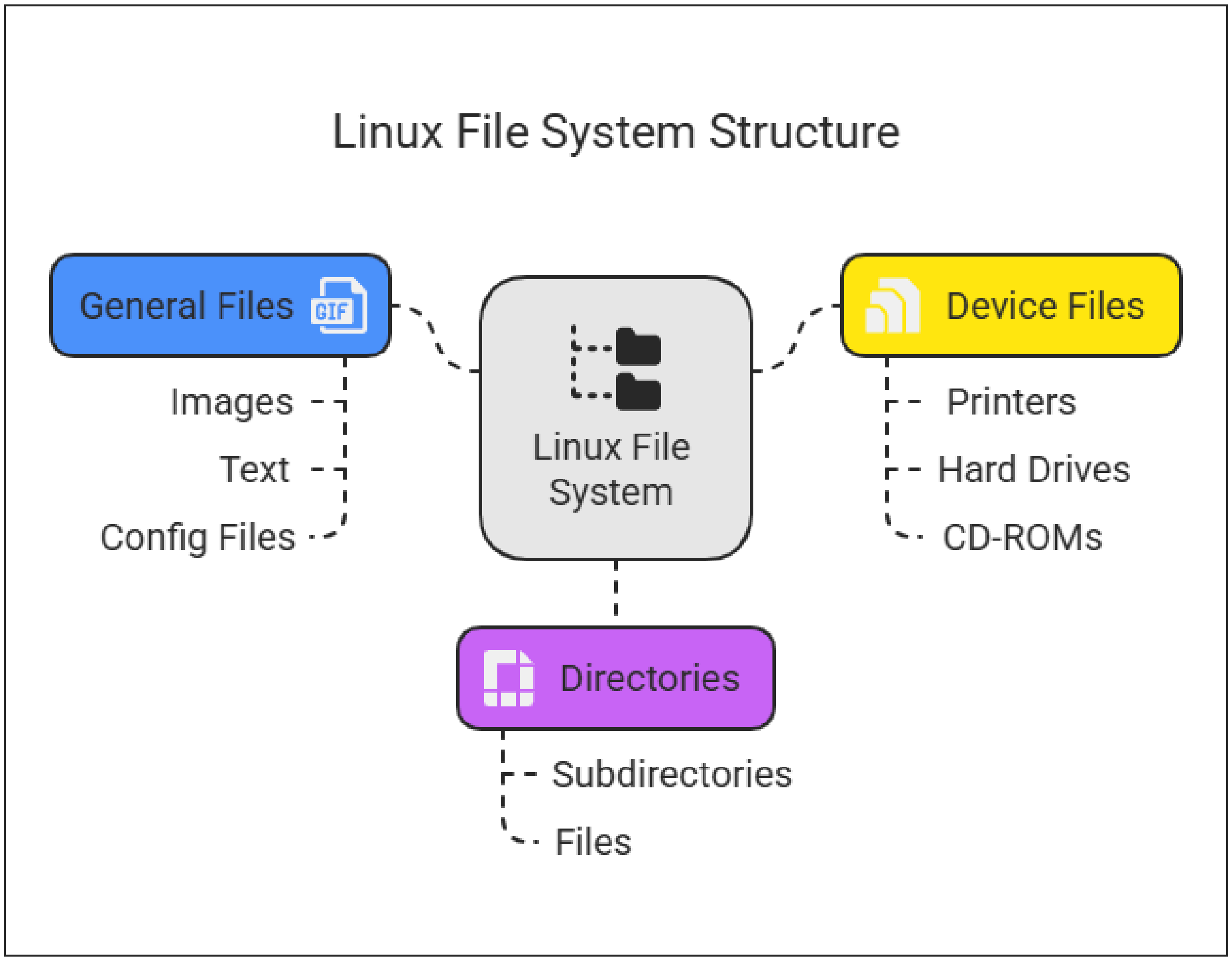


File system in Linux



The files in Linux are stored in tree structure where root is considered as the start of the file system . This file system is further divided into 3 parts

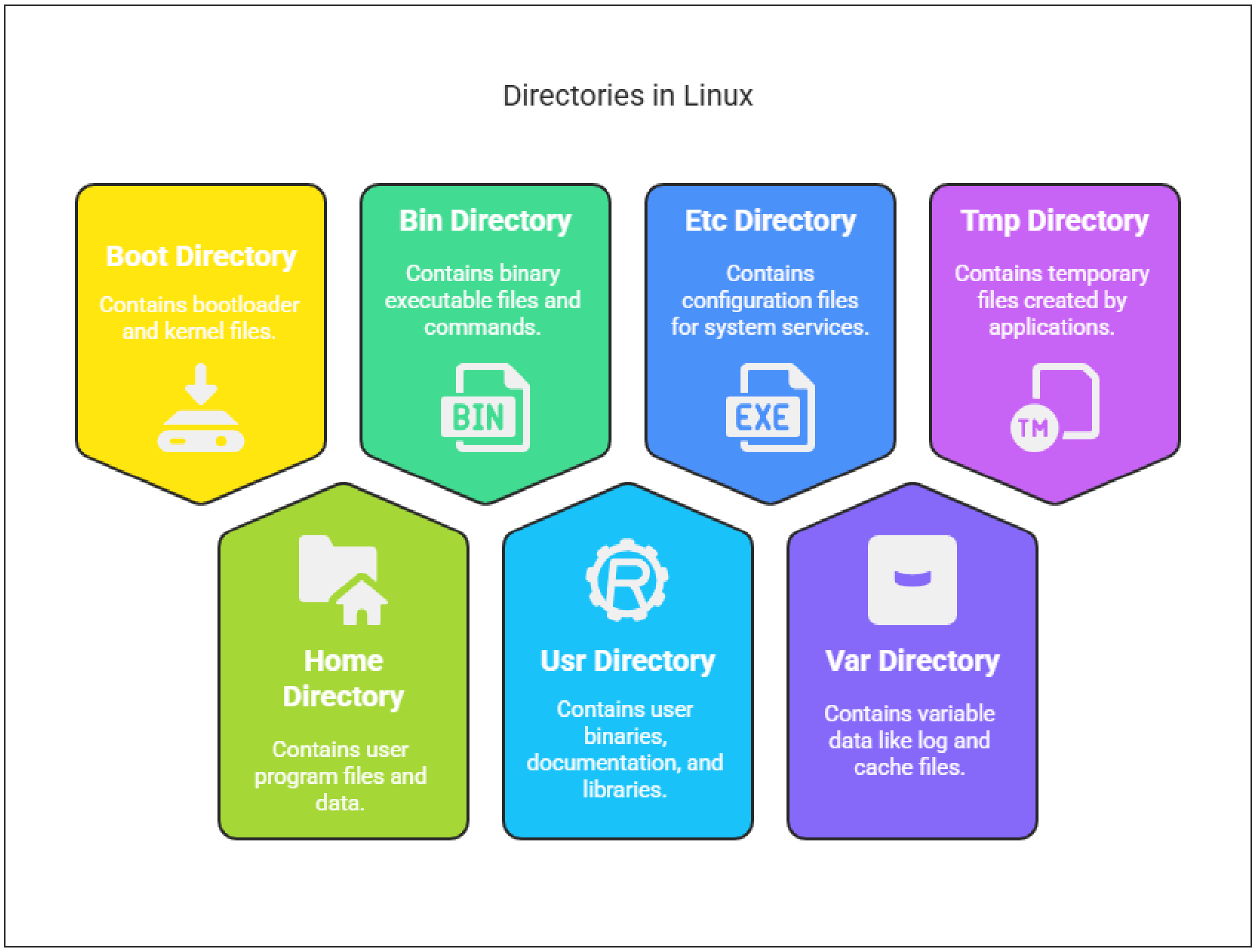
- General - It is also referred to as ordinary files they contain images,text, config files they can store files in ASCII or Binary format. These are the most commonly used file types by linux user.
- Directories - These are special types of files that contain directories and files.
- Device - These are special files that represent physical and virtual devices in the system such as printer, Hard drive, CD-ROM.



Directories

- /boot - This contains bootloader and kernel files, it also contains all the files that are needed during the boot of the system.
- /home - This contains all the program files that are created by the user. It also contains user data, config files, user documents (all the personal files of a user).
- /bin(Binary Binaries) - This contains all the binary executable files, it also contains all the user commands in binary format like cd, pwd, mv, rm.
- /usr - It is also referred as Unix System Resources this contains all the user binaries, documentations, libraries, installed softwares and read only program data

- /etc - It is referred to as Editable Text Configuration that contains all the config files which are used by system services; it also contains system startup and system shutdown script that is used to start any program.
- /var - It is referred to as Variables, contains all the variable data like log files generated by the system, cache files generated for applications.
- /tmp - It is referred to as temporary, contains temporary files created by various applications typically these are cleared when the system reboots.


Directories in Linux
Boot Directory
Contains bootloader and kernel files.
Bin Directory
Contains binary executable files and commands.
BIN
Etc Directory
Contains configuration files for system services.
EXE
Tmp Directory
Contains temporary files created by applications.
TM
Home Directory
Contains user program files and data.
Usr Directory
Contains user binaries, documentation, and libraries.
Var Directory
Contains variable data like log and cache files.

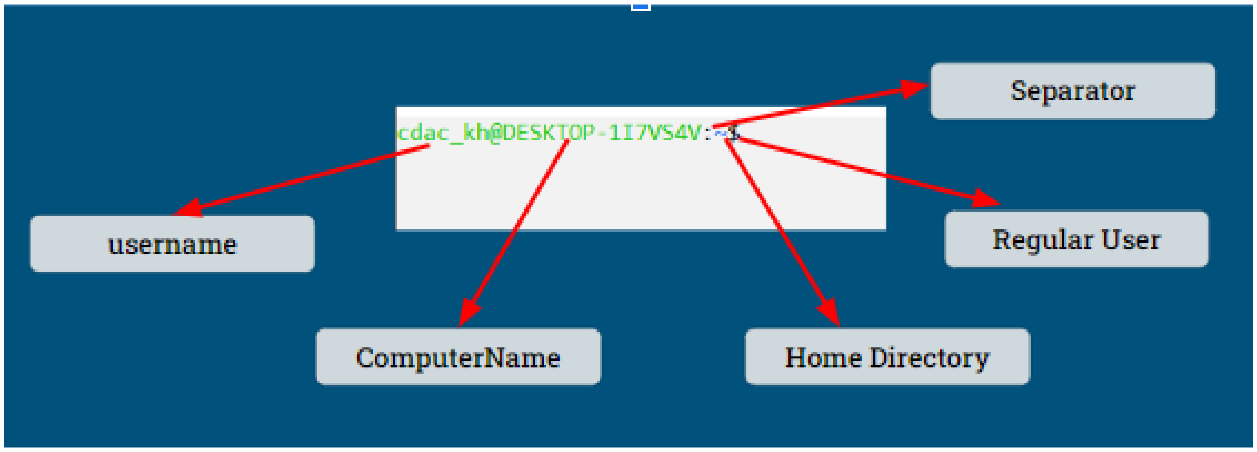
Difference between Windows and Linux

Windows	Linux
Windows is Closed Source	Linux is Open Source
Windows has different Data Drivers	Linux has a tree like structure

Peripheral Devices in windows are like printer, CD-ROM	While here they are considered as files
There are 5 different kinds of users	There are 3 different kinds of users

---

Commands:



':' This is a separator
'~' It shows the user working directories, if we change the directory the sign will vanish
'$' Suggests that you're working as a regular user
'#' this mean you're working as root user
'/' this sign represents the sign of root
Username can be anything you have given to your system
Hostname / Computername this helps to identify computer over any network

Non Root user default prompt is '$'
Default prompt for root is '#'

System Information:

1. Uname : This is used to check OS.

Use Cases:
Cross-platform Scripts: Determining if a script is running on a Linux system.

```
if [ "$(uname)" = "Linux" ]; then
  # Linux-specific commands
else
  # Other OS commands
fi
```

2. Uname -r : Is used to check the version of the OS . Displays the kernel release version (e.g., 5.15.0-56-generic).

Use Cases:
Compatibility Checks: Verifying if the kernel version supports specific features.

```
                    if [[ "$(uname -r)" > "5.0.0" ]]; then
echo "Kernel supports the required features"
else
echo "Kernel too old, please upgrade"
Fi
```

Driver Installation: Some hardware drivers require specific kernel versions.
Bug Reporting: Providing kernel version information when reporting issues.

3. Uname -a: Is used to display all the system information. Displays all available system information, including kernel name, version, machine architecture, and operating system. Displays all available system information, including:
    - Kernel name
    - Kernel release and version
    - Machine hardware name (architecture)
    - Operating system

4. Uname -s: Is used to check OS same as uname. Shows only the kernel name (e.g., Linux).

This is used to display system information. The output typically includes:
- The kernel name.
- The network node hostname.
- The kernel release.
- The kernel version.
- The machine hardware name (architecture).
- The processor type (if available).
- The hardware platform (if available).

- The operating system.

5. Uname -v: Is used to display OS/Kernel versions. Shows the kernel version, including build date and other details.

Use Cases:

1. Detailed Debugging: Providing complete build information when troubleshooting kernel-related issues.
2. System Auditing: Documenting the exact kernel build for security audits.

6. uname -m: Is used to display machine hardware. Displays machine hardware architecture (e.g., x86_64 for 64-bit, i686 for 32-bit).
Use Cases:
Software Installation: Downloading the correct architecture-specific packages.

```
ARCH=$(uname -m)
case $ARCH in
  x86_64)
    # Install 64-bit software
    ;;
  i686)
    # Install 32-bit software
    ;;
  aarch64)
    # Install ARM 64-bit software
    ;;
esac
```

Compatibility Checks: Ensuring software supports the current architecture.

7. uname -a : The uname -a command displays comprehensive system information on Unix-like operating systems (such as Linux, macOS, BSD, etc.). The -a flag stands for "all" and shows all available information in a single line.

When you run this command, it typically displays:

1. Kernel name (Linux, Darwin, etc.)
2. Network node hostname
3. Kernel release version
4. Kernel version
5. Machine hardware name (architecture)
6. Processor type
7. Hardware platform
8. Operating system

Use Cases:

1. Quick System Overview: Getting comprehensive system information with a single command.
2. System Documentation: Capturing complete system details for documentation.
3. Bug Reports: Providing all system information when reporting issues.

8. uname -n → Prints the hostname of the machine.
9. cat /etc/os-release : Provides detailed OS information, including:

   a. Distribution name
   b. Version
   c. OS ID.

Use Cases:
Distribution-specific Scripts: Ensuring compatibility with specific Linux distributions.

```
        OS_ID=$(grep -oP '(?<=^ID=).+' /etc/os-release | tr -d '"')
if [ "$OS_ID" = "ubuntu" ]; then
   # Ubuntu-specific commands
   apt update && apt upgrade -y
elif [ "$OS_ID" = "centos" ]; then
   # CentOS-specific commands
   yum update -y
fi
```

Version Checking: Verifying OS version for software installations.

```
VERSION_ID=$(grep -oP '(?<=^VERSION_ID=).+' /etc/os-release | tr -d '"')
if [[ "$VERSION_ID" < "20.04" ]]; then
  echo "This software requires Ubuntu 20.04 or newer"
  exit 1
fi
```

System Documentation: Recording the exact OS details for documentation.

10. hostnamectl : Displays hostname and OS details, including:
    d. System architecture
    e. Kernel version
    f. Operating system

Use Cases:

1. System Documentation: Getting formatted output of both hostname and OS details.
2. OS Verification: Confirming the exact OS version and architecture.
3. Server Inventory: Collecting data for server inventory management.

11. HostName: It gives you the hostname of your system.

Use Cases:

1. Network Identification: Confirming the current host's name on the network.
2. Multi-server Scripts: Using hostname to determine which server a script is running on.

```
        case $(hostname) in
  web-server*)
    # Web server specific tasks
    ;;
  db-server*)
    # Database server specific tasks
    ;;
esac
```

12. Hostname -I: It gives you the IP address of your system.

Use Cases:

Server Setup: Determining IP addresses for configuration files.

```
                SERVER_IP=$(hostname -I | awk '{print $1}')
echo "server.address=$SERVER_IP" >> application.properties
```

1. Network Troubleshooting: Checking which IP addresses are assigned to the system.
2. Dynamic Configurations: Using in scripts to automatically configure services based on IP.

13. Hostname -f: It gives you the qualified domain name .
Use Cases:
SSL Certificate Setup: Getting the FQDN for SSL certificate generation.

```
    FQDN=$(hostname -f)
openssl req -new -key server.key -out server.csr -subj "/CN=$FQDN"
```

1. Web Server Configuration: Setting up virtual hosts in Apache or Nginx.
2. Email Server Setup: Configuring mail servers that require FQDN.

14. How to set new Hostname: hostnamectl set-hostname newhostname.
Description: Changes the system hostname.

```
    sudo hostnamectl set-hostname new-hostname
```

Use Cases:
Server Provisioning: Setting hostname during server setup.

```
# Set hostname based on role and identifier
sudo hostnamectl set-hostname web-server-01
```

1. System Administration: Updating hostname after role changes.
2. Cloud Instance Setup: Setting proper hostnames for cloud instances.

CPU Information Commands

These commands provide details about the processor, its architecture, cores, and model.

1. lscpu : Displays detailed CPU information, including:
    - Number of cores
    - CPU model
    - CPU speed
    - Virtualization support

   Use Cases:
    1. Hardware Inventory: Documenting detailed CPU specifications.
    2. Performance Benchmarking: Understanding CPU capabilities before testing.
    3. Virtualization Setup: Checking if CPU supports virtualization features.

```
if lscpu | grep -q "Virtualization: VT-x"; then
   echo "CPU supports virtualization, proceeding with VM setup"
   # VM setup commands
else
   echo "CPU does not support virtualization"
   exit 1
fi
```

4.Processor Architecture Verification: Confirming if software is compatible.

```
        if lscpu | grep -q "Architecture: x86_64"; then
   # Install 64-bit version
else
   # Install 32-bit version
fi
```

2. cat /proc/cpuinfo → Shows detailed CPU specifications for each core.Shows detailed information about each CPU core.

Use Cases:

Detailed CPU Analysis: Examining specific features of each CPU core.

```
# Check if CPU supports a specific instruction set
if grep -q "avx2" /proc/cpuinfo; then
   echo "CPU supports AVX2 instructions"
else
   echo "CPU does not support AVX2"
fi
```

CPU Feature Detection: Checking for specific CPU capabilities.

```
# Check for virtualization support
if grep -q -E 'vmx|svm' /proc/cpuinfo; then
   echo "Virtualization supported"
else
   echo "Virtualization not supported"
fi
```

Cache Analysis: Examining cache sizes for performance tuning.

```
# Extract cache size
CACHE_SIZE=$(grep "cache size" /proc/cpuinfo | head -1 | awk '{print $4}')
echo "Cache Size: $CACHE_SIZE KB”
```

CPU Model Identification: Getting exact CPU model information.

3. nproc → Outputs the number of available CPU cores.Shows the number of processing units (cores) available.
   Use Cases:

Parallel Processing: Determining how many parallel tasks to run.

```
# Use optimal number of threads for compilation
make -j$(nproc)
```

Resource Allocation: Setting appropriate thread counts for applications.

Performance Tuning: Scaling services based on available CPU cores.

4. dmidecode

Description: Displays DMI (Desktop Management Interface) table contents, providing detailed hardware information.

Syntax:

- sudo dmidecode -t system Shows detailed system hardware information (requires root).
    - Displays BIOS version, motherboard details, and system serial number.
- sudo dmidecode -t processor Provides a detailed processor report (requires root access).
- sudo dmidecode -t memory

Use Cases:

- Hardware Inventory: Getting detailed system hardware information.
- Memory Configuration: Analyzing memory slot usage and maximum capacity.
- hwinfo : Displays detailed hardware information (if installed).
    - Lists CPU, RAM, storage, and other hardware specs.

Memory (RAM) Information Commands

These commands help in checking total RAM, available memory, and memory usage statistics.

- free: It is used to display information about the system that is how much memory is available, allocated.

Use Cases:

- Memory Usage Monitoring: Checking current memory utilization.
- Resource Planning: Determining if more memory might be needed.

- free -h : Displays RAM and swap usage in a human-readable format (MB, GB). Gives you the info in human readable format (The sizes are specified in MB,KB)
    - Shows total, used, free, shared, buffers, and cached memory.
    - The -h flag ensures that output is easy to read (e.g., 4.2G instead of 4200000 KB).
    - System Administration: Quick health check of memory usage.

```
        # Simple memory monitoring script
MEM_AVAIL=$(free -h | awk '/^Mem:/ {print $7}')
echo "Available memory: $MEM_AVAIL"
```

Performance Monitoring: Regularly checking memory usage for bottlenecks.

```
# Memory alert script
AVAIL_MB=$(free -m | awk '/^Mem:/ {print $7}')
if [ $AVAIL_MB -lt 1000 ]; then
   echo "Critical: Less than 1GB memory available ($AVAIL_MB MB)"
fi
```

Application Deployment: Ensuring sufficient memory before deploying applications.

- cat /proc/meminfo : Provides detailed memory statistics, including:
    - Total memory (MemTotal) – Total amount of RAM installed.
    - Free memory (MemFree) – Unused memory available.
    - Buffers and cache – Memory used for disk buffering and caching..
    - Detailed Memory Analysis: Getting specific memory metrics for troubleshooting.

```
        # Check memory committed ratio
COMMITTED=$(grep "Committed_AS" /proc/meminfo | awk '{print $2}')
TOTAL=$(grep "MemTotal" /proc/meminfo | awk '{print $2}')
RATIO=$(echo "scale=2; $COMMITTED/$TOTAL" | bc)
echo "Memory commitment ratio: $RATIO"
```

Memory Leak Detection: Monitoring specific memory counters over time.
Performance Tuning: Analyzing kernel memory allocations.

- vmstat -s : Summarizes memory, swap, and paging statistics.
    - Shows details on used, free memory, swap usage, and system activity.

- Useful for performance monitoring.
- System Analysis: Getting aggregated memory and CPU statistics.

```
# Check swapping activity
SWAP_IN=$(vmstat -s | grep "pages swapped in" | awk '{print $1}')
SWAP_OUT=$(vmstat -s | grep "pages swapped out" | awk '{print $1}')

if [ $SWAP_IN -gt 0 ] || [ $SWAP_OUT -gt 0 ]; then
   echo "Warning: Swapping detected!"
fi
```

- Performance Monitoring: Tracking context switches and interrupts.
- Capacity Planning: Analyzing memory usage patterns.

Disk & Storage Information Commands

These commands help in checking disk space, partitions, and mounted filesystems.

- df: Shows disk space usage for all mounted filesystems.

  Use Cases:
- Disk Space Monitoring: Checking available space on all filesystems.
- Storage Management: Identifying large or full partitions.

- df -h : Displays disk space usage for all mounted file systems in a human-readable format (MB, GB, TB).

  - Shows the total size, used space, available space, and percentage of usage.
  - System Administration: Quick filesystem space check.
  - Backup Planning: Determining if there's enough space for backups.
  - User Notifications: Alerting users about low disk space.

- lsblk : Lists block devices (hard drives, SSDs, USBs, partitions) in a tree format.

  - Displays disk names, partition sizes, mount points, and types of storage.

Use Cases:

- Storage Configuration: Identifying all storage devices and their partitions.
- Partition Management: Verifying partition structure before making changes.

        - New Disk Setup: Identifying newly added disks for formatting.

- fdisk -l : Shows detailed partition table information of all disks (requires root).
    - Displays partition structure, total disk size, partition type, and filesystem information.
- blkid : Displays UUIDs (Unique IDs) and file system types of partitions.
    - Helps in identifying and mounting storage devices.
- mount | column -t : Lists all mounted file systems, including their mount points and filesystem types.

Use Cases:

- File System Auditing: Checking mount options for security compliance.
- Troubleshooting: Verifying correct mount options for performance issues.
- System Configuration: Documenting current mount setup

- du -sh /path/to/directory : Displays disk usage of a specific directory.
    - The -s flag provides a summary of disk usage.
    - The -h flag makes the output human-readable.

Use Cases:

1. Storage Cleanup: Finding large directories consuming disk space.
2. Quota Management: Checking user directory sizes.
3. Backup Sizing: Determining how much space a backup will require.

- lspci:Lists PCI devices connected to the system.

Use Cases:

1. Hardware Inventory: Identifying all PCI devices in the system.
2. Driver Troubleshooting: Getting exact hardware model information for driver issues.
3. System Compatibility: Verifying hardware compatibility before software installation.

- lspci -v or lspci -vv

Description: Shows detailed or very detailed PCI device information.
Syntax:
lspci -v
Displays PCI devices, including:

- Graphics card

    - Network card
    - Other peripherals

Use Cases:

1. Advanced Troubleshooting: Getting detailed information about hardware devices.
2. Driver Configuration: Examining current driver assignments and capabilities.
3. Device IRQ Analysis: Checking interrupt assignments for performance tuning.

- lsusb: Lists USB devices connected to the system. Displays device names, manufacturer, and bus information.

Use Cases:

- Device Troubleshooting: Identifying connected USB devices.
- Hardware Inventory: Documenting all connected USB peripherals.
- Device Identification: Getting USB device IDs for driver configuration.

Network Information Commands

These commands help in retrieving network details, IP addresses, and active connections.

- ip a / ip addr show : Shows all network interfaces and their assigned IP addresses. Displays all network interfaces on the system with their configurations.
    - Displays information on both wired (eth0) and wireless (wlan0) interfaces.
    - Shows the current IPv4 and IPv6 addresses.

Information Provided:

- Interface names (lo, eth0, wlan0)
- Link state (UP, DOWN)
- MAC address (link/ether)
- IPv4 addresses (inet) with subnet mask in CIDR notation
- IPv6 addresses (inet6) with subnet mask
- Interface flags (BROADCAST, MULTICAST, UP, etc.)
- MTU (Maximum Transmission Unit) value
- Scope (host, global, link)
- Address lifetime (valid_lft, preferred_lft)

Use Cases:

- Network Troubleshooting
- IP Address Extraction: # Extract primary IP address
        - PRIMARY_IP=$(ip -4 addr show eth0 | grep -oP '(?<=inet\s)\d+(\.\d+){3}')
        - echo "Primary IP: $PRIMARY_IP”
- Network Configuration Validation.

- ifconfig : Displays network configuration, such as IP address, MAC address, and subnet mask.
  Traditional command to display and configure network interface parameters.
    - Note: This command is deprecated and may not be installed by default on newer Linux distributions. Use ip a instead.

    - Note: Deprecated, ip a is now preferred.

ifconfig
ifconfig eth0 # For a specific interface
Information Provided:

- Interface names (eth0, lo)
- Interface flags (UP, BROADCAST, RUNNING, etc.)
- IPv4 address (inet) with explicit netmask and broadcast address
- IPv6 address (inet6) with prefix length
- MAC address (ether)
- Packet statistics (RX/TX packets, bytes, errors)
- MTU value

Use Cases:

- Legacy Script Compatibility:
- Network Statistics Monitoring:
- MAC Address Identification:

- ip route : Shows the routing table of the system.Displays the kernel's routing table, showing how network traffic is directed.

ip route
ip route show

    - Displays default gateway, interface details, and active routes.

Information Provided:

- Default gateway (default via IP)
- Network destinations (IP/CIDR)
- Interface for each route (dev eth0)
- Routing protocols (dhcp, kernel, static)
- Scope (link, host, global)
- Source address (src) for outgoing packets
- Route metrics (priority values)

Use Cases:

- Default Gateway Verification.
- Routing Troubleshooting.

- Multiple Network Configuration.
- Source-based Routing Check.

- netstat -r : Displays the kernel routing table, showing available routes and network gateway information.
  Displays the kernel routing table in a traditional format.

Note: netstat is considered deprecated in favor of ss and ip route.

  - netstat -r
  - netstat -rn # -n shows numerical addresses instead of resolving hostnames

Information Provided:

- Destination networks
- Gateway addresses
- Subnet masks (Genmask)
- Flags (U=up, G=gateway, H=host route)
- Interface names (Iface)
- Additional metrics (MSS, Window, irtt)

Use Cases:

- Legacy System Compatibility
- Network Connectivity Verification.
- Interface Route Distribution.

- ss -tulnp : Shows open ports and active network connections.Shows detailed socket statistics, particularly useful for viewing open ports and active connections.

  - ss -tulnp # t=TCP, u=UDP, l=listening, n=numeric, p=processes
  - ss -ta   # Show all TCP connections
  - ss -ua   # Show all UDP connections
  - ss -s   # Show summary statistics

  - -t → Lists TCP connections.
  - -u → Lists UDP connections.
  - -l → Shows only listening sockets.
  - -n → Displays port numbers instead of service names.
  - -p → Shows the process using the port.

Information Provided:

- Socket type (tcp, udp)
- Socket state (LISTEN, ESTABLISHED, UNCONN, etc.)
- Local address and port
- Remote/peer address and port
- Process information (name, PID, file descriptor)

- Queue sizes (Recv-Q, Send-Q)

Use Cases:

- Security Auditing.
- Service Verification.
- Connection Monitoring.
- Connection Limiting: # Count connections per remote IP

```
ss -tn | grep ESTAB | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr | head -5
```

- hostname -I : Prints the system's IP address.

Information Provided:

- All IP addresses (IPv4 and IPv6) assigned to any interface
- Excludes loopback addresses (127.0.0.1, ::1)
- Space-separated list

- cat /etc/hosts : Displays local hostname mappings (useful for troubleshooting).

    Information Provided:

    - Local hostname mappings
    - Loopback address mappings
    - Custom hostname to IP mappings
    - IPv6 loopback and multicast entries

Use Cases:

1. DNS Bypass Troubleshooting
2. Network Testing

- ip neigh (or arp)Shows the ARP table (neighbor cache) containing mappings of IP addresses to MAC addresses.

Syntax:

```
ip neigh
arp -a
```

Use Cases:

- Network mapping
- Detecting rogue devices
- Troubleshooting IP conflicts
- Ping :Tests network connectivity to a specific host.

```
ping -c 4 192.168.1.1  # Send 4 ping packets
```

Use Cases:

- Basic connectivity testing
- Network latency measurement
- Checking if a host is online
- traceroute / tracepath : Shows the network path that packets take to reach a destination.

```
traceroute google.com
tracepath google.com  # Alternative that doesn't require root
```

Use Cases:

- Identifying network bottlenecks
- Troubleshooting routing issues
- Understanding network topology

- dig or nslookup:DNS lookup utility for querying domain information.

```
dig example.com
nslookup example.com  # Simpler alternative
```

Use Cases:

- DNS troubleshooting
- Checking domain records
- Verifying DNS propagation

System Uptime & Load Commands

These commands help in checking system uptime, load average, and performance.

- uptime : Shows system uptime and load average. Displays the system uptime, current time, number of logged-in users, and load average.

    - Displays how long the system has been running.
    - Shows current number of logged-in users.
    - Shows CPU load average over 1, 5, and 15 minutes.

It is used to display how long a system has been running, along with other information such as current time, number of users logged in, system load average. This command is generally used for diagnosing system performance
12:34:56 - Current time
up 10 days, 3:48 - System has been running for 10 days and 3 hours 48 minutes
2 users - Number of users currently logged in
load average: 0.01, 0.05, 0.05 - Load averages for the last 1, 5, and 15 minutes

Use Case:

This command is useful for quickly checking how long the system has been running and understanding the system load.
Example Output:

```
12:34:56 up 10 days, 3:48,  2 users,  load average: 0.01, 0.05, 0.05
```

- 12:34:56: Current time.

    - up 10 days, 3:48: The system has been running for 10 days and 3 hours.
    - 2 users: Number of users currently logged in.
    - load average: 0.01, 0.05, 0.05: Load average for the past 1, 5, and 15 minutes.
- top : Displays real-time CPU and memory usage, along with running processes .Displays real-time information about CPU usage, memory usage, and running processes.

    - Helps in monitoring CPU and memory utilization, sorting by resource usage.

Use Case:
Used for real-time monitoring of your system's resource usage and identifying processes consuming the most resources.

    - CPU Usage: Shows percentages of CPU time spent in user space, system space, and idle time.
    - Memory Usage: Provides the total memory, used, and free memory.

- last: This displays information about the last logged in user with their username, log in time, system shutdown, reboot time and terminal session.Displays information about the last logged-in users, including the username, login time, and session durations.

Use Case:
Used for audit purposes, to check who accessed the system, and when they logged in or out.
Key Fields:

USER: The username of the user.

    - TTY: The terminal line from which the user logged in.
    - IP ADDRESS/ HOSTNAME: The remote host IP address or hostname from which the user accessed the system (if the user logged in remotely).
    - LOGIN TIME: The start time of the user's session.
    - LOGOUT TIME: The time when the user session ended, or the duration of the session if the user is still logged in.
    - DURATION: The total time the user was logged in during that session.

- htop : Interactive process monitoring tool (if installed).A more user-friendly and interactive version of top that provides real-time system statistics, including CPU, memory, processes, and much more.

    - Use Case:
      It is helpful for monitoring system resources interactively with a graphical interface and sorting by CPU and memory usage. (It needs to be installed.)

    - Similar to top, but with a more user-friendly interface

- vmstat 1 5 :Monitors CPU, memory, and disk usage every 1 second for 5 times. Monitors system performance statistics such as CPU, memory, and disk usage, updating every 1 second for 5 iterations.

- Use Case:
  Useful for monitoring system performance over a short period and identifying bottlenecks or resource consumption issues.

- sar – System Activity Reporter
  The sar command collects and reports system activity data, such as CPU usage, memory usage, disk activity, and more.

  ```
  sar -u 1 5  # CPU usage statistics every second for 5 iterations
  ```

  - Use Case:
    Used for performance monitoring over time, especially in environments where long-term statistics are needed.

- dstat – Versatile Resource Monitoring

dstat is a versatile resource monitoring tool that can display information about CPU usage, disk activity, network activity, memory usage, and more in real-time.

  - Use Case:
    Provides comprehensive system resource information in one place, often used for troubleshooting and monitoring.

    ```
    dstat
    ```

- iostat – CPU and I/O Statistics

The iostat command provides statistics on CPU usage and I/O operations, helping monitor system performance and identify bottlenecks.

Use Case:
Helps in analyzing disk I/O performance and identifying slow or overloaded disks.

```
iostat -x 1 5
```

- mpstat – CPU Usage Statistics :mpstat provides detailed CPU statistics, including the usage percentages of each CPU core.

Use Case:
Ideal for multi-core systems, helps monitor the workload distribution across CPU cores.

Example Command:

```
mpstat -P ALL 1 5
```

---

1. whoami

Whoami is a command that is used to display the current user ID and username of the user who is currently logged in, this will print the username of the user who is running the command. This command is useful when you want to check the currently logged in user onto the system.

Use Cases:
Script Authentication: Used in shell scripts to verify the current user before performing privileged operations.

```
if [ "$(whoami)" != "root" ]; then
   echo "This script must be run as root"
   exit 1
fi
```

1. Troubleshooting Login Issues: When dealing with permission problems, checking which user you're logged in as.
2. Multi-user Systems: When switching between users with su or sudo, confirming the current user context.

2. echo $USER :Alternative method to display the current username using the environment variable.

Syntax:
```
echo $USER
```
Use Cases:
Shell Scripts: Often preferred in scripts because it reads from environment variables.

```
# Create a backup directory for the current user
```

1. mkdir -p /backups/$USER/$(date +%Y-%m-%d)

2. Custom Prompts: Used in customizing shell prompts to include username.

   export PS1="$USER@\h:\w\$ "

Users: It is used to display the list of users currently logged in to the system.
The users command simply lists the names of users currently logged in to the system. This command only displays the usernames without additional details.

Use Case:
Used for quickly checking which users are currently logged in, particularly useful for multi-user environments.

w: It is a command used to display the information about currently logged in users, this includes the information related to username, log in time, idle time, system load and what operation they are performing.
USER: The username of the logged-in user.
TTY: The terminal type that the user is logged in from.
FROM: The hostname or IP address from where the user is accessing the system.
LOGIN@: The time when the user logged in.
IDLE: The idle time of the terminal, i.e., how long since any input has been detected on that terminal.
JCPU: The time used by all processes attached to the tty.
PCPU: The time used by the current process, named in the WHAT field.
WHAT: The command line of the process that the user is currently running.
Use Case:
The w command is useful for getting a detailed snapshot of who is logged in and what they are doing on the system, as well as identifying the system's load and user activity.

who: This displays information about users who have logged in with their username, terminal session and their log in time.
NAME: The username of the user logged into the session.
LINE: The terminal or tty (teletypewriter) from which the user is logged in.
TIME: The time when the user logged in.
COMMENT: Often shows the hostname or IP address from where the user is logging in.
The who command displays information about all users currently logged in to the system, including their username, terminal session, login time, and in some cases, the remote host (IP

address or hostname) they are logging in from.

Use Case:
Useful for viewing a list of users logged into the system with details about their terminal session, the time they logged in, and sometimes their remote hostname or IP.
System monitoring: Administrators use these to check who's logged in
Security auditing: Identifying unexpected logins
Resource management: Seeing who might be using system resources
Troubleshooting: Checking if someone is running intensive processes
Server maintenance: Ensuring users are logged out before maintenance

- who -q : Displays only the names and number of users currently logged in.The who -q command displays a quick summary of users currently logged in, including their names and the total number of users.

Use Case:
 Used when you need to get a quick count of the logged-in users along with their names, without any extra details.

- who -r : Shows the current runlevel of the system.The who -r command shows the current runlevel of the system. A runlevel defines the state of the system (e.g., whether it's in single-user mode, multi-user mode, etc.).

Use Case:
Used to check the current system runlevel, which is especially helpful for understanding the system’s state (whether it's in multi-user mode or a single-user mode).

- who -a : Displays all information, combining multiple other options for a comprehensive view.The who -a command combines multiple who options to give a comprehensive view of all information about users currently logged in.

Use Case:
 Used for detailed user session information, including login time, terminal session, remote host, idle time, and more.

- who -b : Shows the last system boot time.The who -b command shows the last system boot time.

Use Case:

Helpful for knowing when the system was last rebooted.

- who -H : Includes column headers in the output. The who -H command includes column headers in the output of the who command for a more structured and readable output.

Use Case:

Useful for generating clearer output when parsing the results programmatically or when you want to easily identify the meaning of each column.

- who -u : Shows the user list with idle time, adding an idle column to the standard display.The who -u command shows a list of users currently logged in along with their idle time, which represents how long it’s been since their last activity.

Use Case:

Useful for determining how active or inactive users are on the system, particularly in multi-user environments.

Date: It is a command that is used to display Date and Time.The date command without any options simply displays the current date and time according to the system's time zone.

Use Case:

This is commonly used for checking the current date and time.

- date: This will give you current date and time
- date +%y-%m-%d: This displays date in YYYY-MM-DD format.You can use format specifiers to display the date in a custom format. %Y-%m-%d will display the date as Year-Month-Day in four-digit year, two-digit month, and two-digit day format.

Use Case:

Useful when you need the date in a standardized or easily sortable format.

- date +%T: This will display time in HHMMSS format. Displays the current time in HH:MM:SS format, where HH is the hour, MM is the minute, and SS is the second.

Use Case:

This command is useful when you need to display just the time without the date.

- date -u: This will display date according to UTC Time zone.The -u flag displays the current date and time in UTC (Universal Time Coordinated) rather than the local time zone.

Use Case:

Useful when working with systems in different time zones or for setting up servers that require UTC time.

- date -u +'%Y-%M-%D %H:%M:%S': This displays UTC time in a given format.This command will display the UTC date and time in a specific format like YYYY-MM-DD HH:MM:SS.

Use Case:

Helpful when you want UTC time formatted in a specific way.

- date +'%A' : This displays the current day. The %A format specifier displays the name of the current day (e.g., Monday, Tuesday, etc.).

Use Case:

Useful for tasks where you need to determine the current weekday.

- date MMDDhhmmccYY.ss: Where MM denotes Month, DD denotes Date, hh denotes hours, mm denotes minutes, cc denotes century, YY denotes year and .ss are the seconds(this is optional)- This command is used to set new date and time for the system

This command allows you to set the system date and time manually using a specific format.

- MM: Month (01 to 12)
- DD: Day of the month (01 to 31)
- hh: Hour (00 to 23)
- mm: Minute (00 to 59)
- cc: Century (for example, 20 for the year 2024)
- YY: Year (last two digits, 24 for 2024)
- .ss: Seconds (optional)

Use Case:

This is typically used by system administrators to manually set the system clock.

```
sudo date 031012342024.59
```

- sudo date +%Y%m%d%H%M.%S -s ‘new date and time’: This command is used to set new date and time

This command allows you to set the system date and time using a specific format:

```
sudo date +%Y%m%d%H%M.%S -s '2024-03-10 12:34:59'
```

- %Y: Year (4 digits)
- %m: Month (2 digits)
- %d: Day (2 digits)
- %H: Hour (2 digits)
- %M: Minute (2 digits)
- %S: Seconds (2 digits)

The -s flag is used to specify the new date and time.

Use Case:
Typically used for setting the system date and time in production environments.

- date +"%H:%M:%S"Displays the current time in the format 12:34:56 (Hour:Minute).

Displays the current time in HH:MM:SS format, similar to the date +%T command, but without the colon separator.

Use Case:
Another way to display the current time in a specific format.

- date +"%A, %B %d, %Y":Displays a more readable format like Monday, September 04, 2024

This displays the date in a more readable format such as: Day, Month Date, Year.

Use Case:
Useful when you need a human-readable date format for reports or logs.

Useful Format Specifiers

- %Y - Year (e.g., 2023)
- %m - Month (01 to 12)
- %d - Day of the month (01 to 31)
- %H - Hour (00 to 23)
- %M - Minute (00 to 59)

- %S - Second (00 to 59)
- %A - Weekday name (e.g., Monday)
- %B - Full month name (e.g., September)

Calendar

- cal/ncal: Is used to display the calendar of a particular year.

The cal command is used to display the calendar for a particular month or year in a simple text-based format.ncal is an alternative to cal that displays the calendar in a more visually appealing format with the current date highlighted.

Use Case:

Use ncal if you want a more user-friendly calendar layout.

Use Case:

It's useful when you want to view a calendar in the terminal quickly without opening any graphical calendar application.

- cal -y: It displays a calendar for the year entered.

The cal -y option displays the calendar for the entire year.

Use Case:

Use this command when you want to see all months of a specific year at once.

- cal -3: It displays the previous month, current month and next month.

The cal -3 option displays the calendar for the previous month, current month, and next month.

```
cal -3
```

Use Case:

Useful for quickly seeing the three consecutive months, which can be helpful for planning or scheduling events.

- cal -3 m y: It displays a calendar for a specific month of the year along with the previous and coming month.

The cal -3 m y option allows you to display the calendar for a specific month (m) of a specific year (y), along with the previous month and next month.

```
cal -3 3 2024
```

Use Case:
Use this when you want to see a particular month and the two surrounding months for a specific year.

- cal m y: It displays a calendar for a particular month in a year.

The cal m y command displays the calendar for a specific month (m) of a specific year (y).

```
cal 3 2024
```

Use Case:
Use this when you need to display the calendar for a specific month in a specific year.

Accessing File System

1. pwd– Print Working Directory: It is used to display the present working directory. It is also known as print working directory, this gives you the absolute path of the current working directory.
   The pwd command stands for Print Working Directory. It simply shows you the current directory you're in, in terms of the absolute path.

When to Use it:

- When you’re unsure of your current directory location within the file system.

- Before performing any actions like moving or creating files, to confirm you’re in the correct directory.

2. /bin/’any command’ –Display Command Version: This will display the version of any command
   This command helps you check the version of a program or command installed on your system.

When to Use it:

- Useful for checking which version of a command or software you're running, especially when troubleshooting or checking compatibility.

- You might need this when ensuring a specific version of a tool (e.g., for testing DBDA or TEST compatibility).

```
/bin/mysql --version
```

3. mkdir ‘directory name’ – Create a New Directory: This command is used to create new directories
   The mkdir command stands for make directory. It’s used to create new directories

(folders) in your file system.

When to Use it:

- Whenever you need to organize files into a new folder (e.g., creating a folder for database backups, test scripts, or log files).

Example: Let's say you're working on a database project called DBDA and need to create a folder for test data:

```
mkdir /home/user/dbda/test_data
```

. (Dot) - Current Directory
The single dot . represents the current directory. It is a reference to the directory that the user is currently in.

.. (Dot Dot) - Parent Directory

The double dot .. represents the parent directory of the current directory. This is the directory one level up in the hierarchy.

4. cd -Change Directory: This is used to change the directory, it is also used to move to home directory
   The cd command is used to change directories (move around in the file system).

When to Use it:

- When you need to navigate between directories, especially when working on different projects (e.g., moving from a DBDA directory to a TEST directory).
- cd ‘directory name’: – Move into a specific directory: This command is used to change the current directory.

- Cd .. : Move to the parent directory (previous directory):This command will move to the previous working directory. If you’re in /home/user/dbda/test_data, running cd .. will bring you to /home/user/dbda.

- Cd -: Move back to the last directory you were in: Will print last working directory. If you were working in /home/user/dbda/test_data and then navigated to /home/user/dbda/backups, cd - will bring you back to /home/user/dbda/test_data.

- Cd ~:Go to your home directory: Will move to home working directory.Takes you to /home/user (assuming user is your username).

- Cd /: Move to the root directory of the file system:Will move to the system working directory.
  This brings you to the root directory, the topmost level of the file system (e.g., /home, /bin, /etc, etc.).

- Cd ~'user':Go directly to another user's home directory: Will move to the user directory.

cd ~cdac
Takes you to the home directory of john, assuming it's /home/cdac.

Creating Files in Linux

1. touch- Create an Empty File: It is used to create an empty file and update the access, modifying the timestamp of the file. It can create multiple files using touch 'file1' 'file2' 'file3'...
   If the file doesn't exist then touch will create a new file with new timestamps and it will be updated
   The touch command is used to create an empty file. If the file already exists, it updates the file's timestamps (i.e., modifies the access and modification times).

When to Use It:
- Creating placeholder files or temporary files for testing purposes.

- When you want to update the timestamp of an existing file without changing its content.

Example: You want to create empty files for your DBDA project like db_schema.txt, test_results.txt, and backup.txt:

```
touch db_schema.txt test_results.txt backup.txt
```

2. Echo– Print and Create Files with Content: It is used to print and display the content on the terminal when it is used with the redirect operator (>) it will create a file adding text to the file.

The echo command is used to display text to the terminal. When used with the > operator, it writes that text to a file, effectively creating or overwriting the file.

When to Use It:
- Useful for quickly adding content to files or for creating new files with content in one go.

Example 1: Create a file with some text: You want to create a file test_info.txt and write "DBDA Test Report" to it:
Syntax : Echo “Hello World” > file1.txt

If you want to add content to an existing file without overriding the content then we will use ‘>>’ operator
Syntax : Echo “Hello World” >> file1.txt
Note: The > operator overwrites any existing content. If the file already exists, the content will be replaced.

Example 2: Append text to an existing file: If you want to add additional information to test_info.txt without overwriting the existing content, use the >> operator:
echo "Test executed on: $(date)" >> test.txt

3. Printf– Create and Update Files: It is similar to echo which creates new files and updates pre-existing files.

printf works similarly to echo, but it offers more formatting options. It's used to print formatted text and can be used to create files as well.

When to Use It:

- When you need more control over formatting (e.g., including newlines, tabs, or specific formats).

- For tasks where escape characters (like \n for newline) are needed.

Syntax : printf “Hello World” > file1.txt
Similar Syntax if you do not want to override
In printf we can use escape characters while in echo we can not use escape characters.
printf "Test Execution Date: %s\nTest ID: %d\nStatus: %s\n" "$(date)" 101 "Passed" > test_report.txt
printf "Additional Test Information: %s\n" "All checks passed." >> test_report.txt

4. Nano– Simple Text Editor: It is used to create files and its syntax is : nano ‘filename’ , to save the file we press ‘Ctrl + o’.
   Exit - ‘Ctrl +x’
   Cut / Delete from current line - ‘Ctrl + k’
   To Search - ‘Ctrl + w’
   To Paste / Uncut - ‘Ctrl + u’
   nano is a simple, user-friendly terminal text editor used to create and edit text files. It is

great for quick file edits or creating files directly from the terminal.

When to Use It:

- When you need to create or modify files directly from the terminal in an interactive way.

5. vi/vim – Advanced Text Editor : It is used to perform tasks like creating, editing, saving and navigation. vi (or vim, an enhanced version of vi) is a powerful text editor used for creating and editing files. It operates in modes (normal, insert, command-line mode), which makes it more flexible for advanced editing.

When to Use It:

- When working with larger files, coding, or performing advanced file edits. It's ideal for configuration files or complex scripts for DBDA or TEST

To insert - Press 'i'
To move to Escape mode: Press 'Esc'
Move to beginning of the file in escape mode: Press 'gg' or '/G'
Move cursor to left: 'h'
Move cursor to right: 'l'
Move cursor up : 'j'
Move cursor down: 'k'
Search forward : '/search'
Search Backwork: '?search'
Delete: 'dd'
Undo: 'u'
Redo: 'Ctrl + r'
Copy: 'yy'
Paste: 'p'
Quit: ':q!'
Write and Quit: ':wq'

Modal Interface: vi operates in various modes, primarily:

- Normal Mode: For navigating and editing text, where keyboard inputs perform editing and navigation functions.
- Insert Mode: For inserting text. Accessed from Normal Mode by pressing i, a, o, etc.
- Command-Line Mode: For entering commands that can search text, replace strings, save files, etc. Accessed from Normal Mode by pressing :.

Esc mode refers to the mode you enter when you press the Esc (Escape) key. The Esc key is used to exit other modes and return to Normal Mode.

Basic Commands in vi

- Entering Insert Mode:
  - i: Insert before the cursor.
  - a: Append after the cursor.
  - o: Open a new line below the current line.
- Exiting Insert Mode:
  - Esc: Return to Normal Mode.
- Saving and Exiting:
  - :w: Save the file but keep it open.
  - :wq or ZZ: Save the file and quit vi.
  - :q: Quit if there are no changes.
  - :q!: Quit and ignore any changes.
- Cut, Copy, and Paste:
  - dd: Cut (delete) a line.
  - yy: Copy (yank) a line.
  - p: Paste below the cursor.
  - P: Paste above the cursor.
- Undo and Redo:
  - u: Undo the last operation.
  - Ctrl + r: Redo the last undo.
- Moving Around:
  - h, j, k, l: Move left, down, up, right, respectively.
  - 0: Move to the beginning of the line.
  - $: Move to the end of the line.
  - G: Move to the end of the file.
  - gg: Move to the beginning of the file.

Search and Replace with Confirmation:

- Command: :%s/old/new/gc

Autocompletion:

- Command: Ctrl-n and Ctrl-p (in insert mode)

Faster Scrolling:

- Command: Ctrl-u and Ctrl-d

Editing Multiple Files:

- Command: :args file1 file2 file3 then use :next, :prev, :first, :last to navigate.

Diff and Merge:

- Command: vimdiff file1 file2

Block Selection and Editing:

- Visual Block Mode: Press Ctrl-v to enter visual block mode.

---

ls: It is used to show the list of files and directories. The ls command shows a list of files and directories in the current directory.

When to Use It:
Whenever you want to quickly see what files or directories are in the current working directory.

- ls -l :Detailed List of Files and Directories: This shows the files modification size, time, file, folder name and their owners with the permissions.
    - The -l flag provides a long listing format that includes detailed information such as:
    - Permissions
    - Number of links
    - Owner and group
    - Size
    - Last modification date and time
    - File or directory name
        - When to Use It:
        - When you want to see detailed information about the files or directories, such as ownership, permissions, and modification times.
- ls -a:List All Files, Including Hidden Ones: This will list out all the hidden files (starts with . and ..).
    - The -a flag lists all files, including hidden files (those that start with a dot . like .git or .bashrc).

        - When to Use It:
        - When you need to see hidden files that are often used by the system or applications (like configuration files).
- ls -it:List Files Sorted by Modification Date: This shows the list of files and directories with modified date in ascending order. The -it flag lists files in ascending order by their

modification time. The most recently modified files appear at the bottom.

        - When to Use It:
        - When you want to see files sorted by when they were last modified, useful for version control or test results that change over time.
- ls -S: List Files Sorted by Size: This shows list of files and directories in descending order.The -S flag lists files sorted by their size, with the largest files listed first.

        - When to Use It:
        - When you want to organize files by size, which could be helpful for finding large log files or database dumps in a DBDA project.

- ls -n: List Files with Numeric User/Group ID :This gives the user, username, group id of any file or directory.
    - The -n flag shows the numeric user ID (UID) and group ID (GID), instead of user/group names.

        - When to Use It:
        - When you need to see the UID and GID rather than the actual names for users or groups.

- ls ~: List Files in the Home Directory: This gives you the list of files and directories at home.
  The ~ symbol refers to the home directory of the current user. ls ~ lists all files and directories in your home directory.

When to Use It:

- When you want to quickly list files in your home directory, especially if you want to manage files related to your project setup, configurations, or scripts.

- ls *: This gives you a list of directories and sub directories.

- ls -i: List Files with Inodes: This gives you an inode of the directories.(inode is an index node that is a unique number which holds metadata of file, attributes of file like size, owner of file, creation of file, permission, location, file type and any other link attached to the file).

The -i flag shows the inode number of each file. The inode is a unique number that stores the metadata of a file (like size, owner, permissions, etc.).

When to Use It:

- Useful when you need to track or find files by their inode number for low-level file management tasks, especially when debugging file system issues.

- ls -R: List Files Recursively: This will list you all the directories in tree format.
  The -R flag lists all files in the current directory and subdirectories, showing the directory structure in a tree-like format.

When to Use It:

- When you need to see all files and directories within a folder and its subdirectories, such as when working with complex directory structures in a DBDA project folder.

- ls -r: List Files in Reverse Order: This will show you files and directories in reverse order.

The -r flag reverses the order of the file listing, showing files from last to first.

When to Use It:

- When you want to view files in the reverse order, such as the oldest files first.

- ls -d */:List Only Directories: This gives you a list of directories.The -d flag with the */ pattern lists only directories, not files.

When to Use It:

- When you want to see just the directories and ignore regular files, such as when organizing project directories or managing DBDA subfolders.

- ls -l /temp: List Files in a Specific Directory: This gives you the list of temporary files.The -l flag lists detailed information about files in a specific directory. /temp is used here as an example for the temporary files directory.

When to Use It:

- When you need to list the contents of a specific directory like temporary files in /temp.

- ls -lS: List Files Sorted by Size: This gives you a list of files and directories in order of their sizes.The -lS flag sorts the files in the directory by size, with the largest files shown first.

When to Use It:

- Useful for finding and managing large files, such as when cleaning up your TEST logs or analyzing DBDA data dumps.

Removing Files and Directories in Linux: rm and rmdir Commands

1. rm ‘filename’:Remove a Specific File: It is used to remove directories, files and content within the directories.

The rm command is used to remove a file from the system. This is a permanent deletion, and it doesn’t go to the trash or recycle bin.

```
rm test_info.txt
```

When to Use It:

- When you want to delete specific files, such as temporary data files or outdated logs in your DBDA projects.
- Be careful, as this command doesn't move files to a trash can—once they're deleted, they’re gone!

- rmdir ‘directoryname’: Remove an Empty Directory: Removes the specific directory name.

The rmdir command is used to remove an empty directory. It won't work if the directory contains any files or subdirectories.

```
rmdir old_logs
```

When to Use It:

- When you need to remove empty directories, such as after cleaning up temporary directories in your DBDA backups.

- rm -r ‘directoryname’ : Remove a Directory and Its Contents: This deletes the directory along with its contents
  The -r (recursive) flag allows you to delete a directory and all of its contents, including files and subdirectories. This is very powerful, so use it with caution.

When to Use It:

- When you need to delete non-empty directories and everything within them.
- Ideal for removing old project directories or large logs that you no longer need.

Force deletion without prompts (use carefully): If you want to force remove a directory without getting any prompts (even for non-empty directories):

```
rm -rf old_dbda_test_data
```

The -f option forces the removal of files without asking for confirmation (even for write-protected files).

Copying Files and Directories in Linux: cp Command:

The cp command is used to copy files and directories from one location to another in Linux

1. cp:Basic File Copy : It is used to copy files and directories from one location to another
   Syntax - cp 'source' 'destination'

The basic syntax of cp is used to copy a file from the source location to the destination.

- When to Use It:

When you need to duplicate a file or copy it to another directory, like copying a test script or configuration file from one folder to another.
Example 1: Copy a file from one directory to another You have a file test_report.txt in the TEST directory and want to copy it to the backup folder:
cp test_report.txt /home/user/backup/
Example 2: Copy a file and rename it during the copy You want to copy db_schema.txt from the DBDA project folder to another folder and rename it as backup_db_schema.txt:
cp db_schema.txt /home/user/backup/backup_db_schema.txt.

- cp -n 'source' 'destination' Prevent Overwriting Existing Files: does not override an already existing file.

The -n (no-clobber) option prevents overwriting an existing file at the destination. If a file already exists with the same name, it will not be copied again.

- When to Use It:
  When you want to avoid overwriting existing files and only copy the file if it doesn't already exist in the destination.

Example: You want to copy test_info.txt from the TEST directory to the backup folder, but only if test_info.txt doesn't already exist in the backup folder:

```
cp -n test_info.txt /home/user/backup/
```

If test_info.txt already exists in /home/user/backup/, the file will not be overwritten. Otherwise, it will be copied.

- cp -r 'source' 'destination' : Copy Directories Recursively: copies directories from one place to another.

The -r (recursive) option allows you to copy directories along with their contents, including subdirectories and files.

- When to Use It:

When you need to copy a whole directory and its subdirectories, such as when you're duplicating project folders or backup directories.

Example 1: Copy a directory and its contents You want to copy the test_data/ directory (and everything inside it) to the /home/user/backup/ directory:

```
cp -r test_data/ /home/user/backup/
```

- cp -a ‘source’ ‘destination’:– Copy and Preserve File Attributes: preserves the file attributes

The -a (archive) option copies files and directories while preserving their attributes, such as permissions, timestamps, and symbolic links.

- When to Use It:
  When you need to copy files or directories but preserve their original metadata. This is useful when backing up important project files or ensuring that copied files maintain their permissions and attributes.

Example 1: Copy a directory with attributes preserved You want to copy the entire db_backup/ directory while preserving its permissions, ownership, and timestamps:

```
cp -a db_backup/ /home/user/backup/
```

Outcome:
The db_backup/ directory is copied, along with all its contents and metadata (permissions, timestamps, etc.).

Moving Files and Directories in Linux: mv Command
The mv command in Linux is used to move files or directories from one location to another. It can also be used to rename files and directories.

1. mv:Basic File Move: It is used to move file from one location to another
   Syntax - mv ‘source’ ‘destination’

The mv command is used to move a file from one location (the source) to another (the destination). It can also be used to rename files if the destination is a new filename.

- When to Use It:
  When you want to move a file to a different location within your file system, such as organizing your DBDA project files.
  Renaming files or directories in one command.

- mv ‘file.txt’ ‘destination’ - This is moving files to a destination.

Example 1: Move a file to a different directory You have a file test_report.txt in your TEST project folder, and you want to move it to the backup/ directory:

```
mv test_report.txt /home/user/backup/
```

- mv ‘file.txt’ ‘destination/new.txt’: Move and Rename a File - This command moves and renames the file.

When you specify a new filename as the destination, the mv command will move the file and rename it at the same time.

- When to Use It:

When you want to move a file to a different location and also rename it in one step.

Example 1: Move and rename a file You have db_schema.txt in the DBDA project folder, and you want to move it to the backup/ folder while renaming it to db_backup.txt:

```
mv db_schema.txt /home/user/backup/db_backup.txt
```

The file db_schema.txt is moved to /home/user/backup/ and is renamed to db_backup.txt.

- mv dir1 ‘destination’ : Move a Directory: This moves the directory.

The mv command can also be used to move directories to new locations, just like files.

- When to Use It:
  When you need to move a whole directory (e.g., project directories, backup folders, or test data directories) to another location.

Example 1: Move a directory You have a directory test_data/ in your TEST project folder, and you want to move it to the backup/ folder:

```
mv test_data/ /home/user/backup/
```

The test_data/ directory, along with all its contents, is moved to /home/user/backup/.
Extra:

Overwrite Warning: By default, mv will overwrite files at the destination without any warning if a file with the same name exists. To avoid this, you can use the -i (interactive) flag, which will prompt you before overwriting:
mv -i test_report.txt /home/user/backup/
Difference between CP and MV command

CP	MV
It is used to copy a file and directory	It is used to move file and directory to a new location
Copies the file but doesn't delete the original file	Mv command deletes the original file while moving

1. Rename: We can rename a file using two methods
   A. rename: This is used to rename files in bulk by specifying match to match and replace pattern.
   B. mv: This is used to rename one file at a time.
   Syntax: rename 's/"old pattern"/ "new pattern"' "filename to be renamed"
   S stands for substitute
   / stands for delimiter used to separate
   g stands for global, if you want to replace all the occurrences globally
   ^ stands for beginning of the character
   $ indicates end of the line
   Example: rename 's/\.txt$/ sh/' *txt

Metacharacter	Name	Description	Example
.	Dot	Matches any single character except newline characters	a.b matches aab, abb, acb, etc., but not ab
^	Caret	Anchors the match to the start of the string	^abc matches abc at the beginning of a string, but not xyzabc
$	Dollar Sign	Anchors the match to the end of the string	abc$ matches abc at the end of a string, but not abcxyz
*	Asterisk	Matches zero or more occurrences of the preceding character or group	a*b matches b, ab, aab, aaab, etc.

+	Plus	Matches one or more occurrences of the preceding character or group	a+b matches ab, aab, aaab, etc., but not b
?	Question Mark	Matches zero or one occurrence of the preceding character or group	colou?r matches both color and colour
{n,m}	Braces	Matches between n and m occurrences of the preceding character or group	a{2,4} matches aa, aaa, or aaaa, but not a

Character Classes
Character classes define a set of characters that can be matched at a specific position in the pattern.

- [abc]: Matches any one of the characters a, b, or c.
    - Example: [aeiou] matches any vowel in a string.
- [^abc]: Matches any character except for a, b, or c.
    - Example: [^a-z] matches any non-lowercase letter.
- [0-9]: Matches any digit (equivalent to \d in many regex engines).
    - Example: [0-9]{3} matches any 3-digit number.
- [A-Za-z]: Matches any uppercase or lowercase letter.

Predefined Character Classes

- \d: Matches any digit (equivalent to [0-9]).
    - Example: \d{2,4} matches any number with 2 to 4 digits.
- \D: Matches any non-digit character.
    - Example: \D+ matches one or more non-digit characters.

- \w: Matches any word character (alphanumeric plus underscore: [A-Za-z0-9_]).
    - Example: \w+ matches one or more word characters.
- \W: Matches any non-word character.
    - Example: \W+ matches one or more non-word characters.
- \s: Matches any whitespace character (spaces, tabs, newlines).
    - Example: \s+ matches one or more whitespace characters.
- \S: Matches any non-whitespace character.
- \b: Matches a word boundary.
    - Example: \bword\b matches the word word but not sword or wording.
- \B: Matches a non-word boundary.

Grouping and Capturing

You can group parts of a pattern and capture them for later use.

- (abc): Groups the characters abc together as a single unit.
    - Example: (abc)+ matches one or more occurrences of abc.
- \1, \2, etc.: Refers to the captured group (also called backreference).
    - Example: (abc)\1 matches abcabc.

The alternation operator | allows you to match one pattern or another.

- a|b: Matches either a or b.
    - Example: cat|dog matches either cat or dog.

Escape Sequences

Some characters are special in regular expressions and need to be escaped using a backslash (\) if you want to match them literally.

- \.: Matches a literal dot (.), because . is a special character in regex.
- *: Matches a literal asterisk (*).
- \?: Matches a literal question mark (?).

Regex Quantifiers

Quantifiers control how many times an element should be matched.

- *: Zero or more occurrences of the preceding element.
- +: One or more occurrences of the preceding element.
- ?: Zero or one occurrence of the preceding element.
- {n}: Exactly n occurrences.
- {n,}: At least n occurrences.
- {n,m}: Between n and m occurrences.

Difference between rename and mv

rename	mv
It uses regular expression	This doesn't use any regular expression
We can rename bulk of files in a single command	We can only rename one file at a time
It doesn’t change the location of the file	It changes the location of the file

Change file extension: .txt → .md

```
rename 's/\.txt$/.md/' *.txt
```

notes.txt → notes.md

Add a prefix to all files

rename 's/^/new_/' *
report.doc → new_report.doc

Add a suffix before extension

rename 's/\.txt$/_backup.txt/' *.txt
data.txt → data_backup.txt

Replace spaces with underscores

rename 's/\s+/_/g' *
my file name.txt → my_file_name.txt

Convert all lowercase filenames to UPPERCASE

rename 'y/a-z/A-Z/' *
file.txt → FILE.TXT

Remove all digits from filenames

rename 's/\d+//g' *
data123.csv → data.csv

Remove all special characters (non-word characters)

rename 's/\W+//g' *
hello-world@2024!.txt → helloworld2024txt

Rename all .jpeg to .jpg

rename 's/\.jpeg$/.jpg/' *.jpeg
image.jpeg → image.jpg

Replace a specific word in filenames

```
rename 's/draft/final/' *
project_draft.doc → project_final.doc
```

Add today’s date to filenames

```
rename 's/^(.*)\.txt$/$1_$(date +%Y-%m-%d).txt/' *.txt
report.txt → report_2025-03-24.txt
```

Always use -n flag first to preview: `rename -n 's/_/-/g' *`

- Keep a log of renames in scripts if possible.
- Use versioned backups if renaming important files.

Remove spaces from filenames (replace with underscore):

```
rename 's/ /_/g' *.txt
```

Changes: my report.txt → my_report.txt

Add today’s date as a suffix to all .csv files:

```
rename "s/\.csv$/_$(date +%Y-%m-%d).csv/" *.csv
```

Changes (if today is 2025-03-24):
sales.csv → sales_2025-03-24.csv
data.csv → data_2025-03-24.csv

- Replace a specific word in filenames:

Replace the word draft with final in all .docx files:

```
rename 's/draft/final/' *.docx
```

Changes: project_draft.docx → project_final.docx

- Convert all filenames to lowercase:

```
rename 'y/A-Z/a-z/' *
```

Changes:
Report.TXT → report.txt
IMAGE.PNG → image.png

- Convert all filenames to UPPERCASE:

```
rename 'y/a-z/A-Z/' *
```

Changes:
file1.txt → FILE1.TXT
notes.md → NOTES.MD

- \d – Match digits : Rename files like file123.txt to file_123.txt

```
rename 's/(\d+)/_\1/' *.txt
```

  - \d+ → means match one or more digits
  - () → creates a capture group so we can refer to it as \1
  - _\1 → adds an underscore before the digits

g in s/(\d+)/_\1/g stands for "global"

Option	Meaning	Result
s/pattern/replacement/	Replace first match only	1st match
s/pattern/replacement/g	Replace all matches	Global

- \D – Match non-digits : Remove all non-digit characters from filenames:

```
rename 's/\D+//g' *.txt
```

What it does:

- \D+ matches one or more non-digit characters.
- Replaces them with nothing → removes all non-digits.
- g applies it globally across the whole filename.

Example Before:

file123name.txt
report_2024_version.txt
log-07.txt

After running the command:

```
123.txt
2024.txt
07.txt
```

- \w – Match word characters : Replace all word characters with X:

```
rename 's/\w/X/g' *.txt
```

What it does:

- \w → matches any word character: [A-Za-z0-9_]
- X → replaces each word character with X
- g → applies the replacement globally (i.e. to all matches in each filename)
- *.txt → applies only to .txt files in the current directory

Example Before:

```
data123.txt
test_file.txt
```

After:

```
XXXXXXX.txt
XXXXXXXXX.txt
```

- \W – Match non-word characters : Remove all special characters (non-word):

```
rename 's/\W//g' *
```

What it does:

- \W → matches any non-word character, i.e., anything that's not [A-Za-z0-9_]
  This includes: spaces, hyphens, dots (.), @, #, etc.
- '' (empty) → replaces each non-word character with nothing
- g → does it globally (across the entire filename)

Example Before:

```
my file.txt
hello-world@2024!.txt
Report_2024.txt
```

After Rename:

```
myfiletxt
helloworld2024txt
report_2024txt
```

- \s – Match whitespace :Replace spaces in filenames with underscores:

```
rename 's/\s+/_/g' *
```

What it does:

- \s+ → matches one or more whitespace characters (space, tab, etc.)
- _ → replaces them with an underscore
- g → applies it globally (to all whitespace in the filename)

Example Before:

```
my file name.txt
test   data.csv
 spaced out .log
```

After Rename:

```
my_file_name.txt
test_data.csv
_spaced_out_.log
```

- \S – Match non-whitespace :Keep only whitespace (delete other characters):

```
rename 's/\S+//g' *
```

What it does:

- \S+ → matches one or more non-whitespace characters
- '' → replaces them with nothing
- g → replaces all matches in the filename

- \b – Word boundary : Add test_ before files that have the exact word "report":

```
rename 's/\breport\b/test_report/' *
```

What it does:

- \b → word boundary anchor (ensures report is a separate word)
- 'report' → matches the exact word "report"

- 'test_report' → replaces it with this
- * → applies to all files in the current directory

report_2024.txt → test_report_2024.txt

Does not affect projectreport.txt or reporting.txt.

Example Before:

```
report.txt
weekly_report.txt
finalreport.txt
reporting_notes.txt
```

After Rename:

```
test_report.txt
weekly_report.txt       unchanged
finalreport.txt         unchanged
reporting_notes.txt     unchanged
```

- \B – Non-word boundary: Replace word only when inside another word:

```
rename 's/\Bword/BLOCK/' *
```

What it does:

- \B → non-word boundary, matches only when "word" is not at the start of a word.
- word → looks for the literal text "word"
  So, it replaces "word" only if it's inside another word (not at the beginning)
- BLOCK → replacement
- * → applies to all filenames in the current directory

sword.txt → sBLOCK.txt

Does not change word.txt

Example Before:

```
word.txt
sword.txt
keyword_report.txt
```

wording.txt

After Rename:

word.txt	✅ unchanged
sBLOCK.txt	✅ "word" is inside, not at start
keyBLOCK_report.txt	✅
BLOCKing.txt	✅ (from wording.txt)

Difference between rename and mv

rename	mv
It uses regular expression	This doesn't use any regular expression
We can rename bulk of files in a single command	We can only rename one file at a time
It doesn’t change the location of the file	It changes the location of the file

---

Rsync: A Powerful File Synchronization and Transfer Tool

The rsync command is widely used for efficiently transferring and synchronizing files and directories, both locally and remotely. It’s especially useful for keeping backups or syncing files between remote systems or different directories on the same machine.
It is a command that is used to transfer and synchronize files and directories between two networks, these are specially used to keep backups or to remotely synchronize files.
Rsync gives

Syntax: rsync 'source' 'destination’

The basic syntax of rsync copies files or directories from a source to a destination. This can be done locally (on the same machine) or remotely (between different systems).

- When to Use It:
  Use rsync when you need to copy files from one place to another, especially when you want to sync files or keep backups.

Example 1: Basic Local File Sync You want to sync the contents of db_backup/ directory to the backup/ directory on your local machine:

```
rsync db_backup/ backup/
```

Outcome:The contents of db_backup/ are copied to the backup/ directory. Only changed files are transferred, making it efficient for large datasets.

- Rsync -av ‘source’ ‘destination’ :Sync a Directory Locally with Archive Mode: Sync a directory locally.

The -a flag stands for archive mode, which preserves the file attributes such as permissions, timestamps, symbolic links, and recursive copying of directories. The -v flag makes rsync run in verbose mode, providing detailed output during the synchronization.

- When to Use It:
- Use rsync -av when you need to sync directories locally, ensuring that all file attributes are preserved.

Example 1: Sync a local directory with all attributes preserved You want to sync your test_logs/ directory from your TEST project and preserve everything (permissions, timestamps):

```
rsync -av test_logs/ /home/user/backup/test_logs/
```

Outcome: The test_logs/ directory is copied to /home/user/backup/test_logs/ while preserving all its file attributes, including timestamps and permissions.

- Rsync -av -e ssh ‘source’ user@remotehost:’destination’: Sync Files Remotely Over SSH

The -e ssh option tells rsync to use SSH (Secure Shell) for encrypted communication while transferring files. This command is ideal for syncing files between remote systems securely over the internet.

- When to Use It:
  Use this command when you need to sync files between remote systems or between your local machine and a remote server. It ensures secure file transfer over SSH.

Example 1: Sync files remotely to a server You want to sync your test_data/ directory from your local TEST project to a remote server with the user user on remotehost:

```
rsync -av -e ssh test_data/
user@remotehost:/home/user/remote_backup/test_data/
```

Outcome:The test_data/ directory is copied from your local machine to the remote_backup/test_data/ directory on the remote server (remotehost), with all file attributes preserved.

Example 2: Sync a database backup directory remotely You want to sync the DBDA project's db_backup/ directory from your local machine to a remote server for backup:

```
rsync -av -e ssh db_backup/ user@remotehost:/home/user/db_backup/
```

Outcome:The db_backup/ directory is securely synced to the remote server's db_backup/ directory.

- -z (Compression):
  Compresses file data during transfer, which can be helpful when syncing large files or directories over a network with limited bandwidth.

  ```
  rsync -avz db_backup/ user@remotehost:/home/user/db_backup/
  ```

- --delete (Delete files not in source):
  Removes files from the destination that are no longer present in the source. This is useful for maintaining an exact mirror of the source directory.

  ```
  rsync -av --delete test_data/ user@remotehost:/home/user/remote_test_data/
  ```

Difference between rsync and mv

rsync	mv
It is used to copy and sync files and directories	It moves files and directories
It can transfer or copy files locally or remotely	It can only copy files locally
It also provides detailed information during the transfer	It doesn't provide full information
It is used to sync, take backup and transfer files remotely/locally	It moves the files and directories
It copies the differences and update	It overrides the existing files

---

Counting Words, Lines, and Characters with wc Command

The wc (word count) command is a handy tool for counting the number of words, lines, and characters in a file. Whether you're analyzing log files from scripts or reviewing data backups in DBDA, this command helps you quickly gather statistics about your files.

- wc:wc 'filename' – Count Words, Lines, and Characters in a File: It is used to count the number of words,lines and characters in a file.

Syntax : wc 'filename'
wc test_report.txt

- The wc command without any options counts:
    Lines: The number of lines in the file.

Words: The number of words in the file.
Characters: The number of characters in the file.

- When to Use It:

When you want to quickly know how large a file is in terms of lines, words, and characters, especially for project files, test logs, or reports.

- wc -l: Count Number of Lines in a File: Returns number of lines.

The -l option returns the number of lines in a file.

```
wc -l test_log.txt
```

  - When to Use It:
    When you need to quickly know how many lines are in a file, such as counting the log entries or lines of code in a test script.

- wc -w: Count Number of Words in a File: Returns number of words.

The -w option counts the number of words in a file.

```
wc -w test_script.sh
```

- When to Use It:
  Useful for analyzing textual content or counting words in reports, documentation, or scripts.

- wc -m:Count Number of Characters in a File: Returns number of characters.

The -m option counts the number of characters in a file.

```
wc -m data.csv
```

  - When to Use It:

When you need to analyze the character count of a file, such as when checking the size of database records or log entries.

- wc -L: Count the Length of the Longest Line: Returns the longest line.

The -L option returns the length of the longest line in a file.

- When to Use It:

When you want to know the maximum line length in a file, useful for finding lines with unusually large content, like long database entries or log messages.
wc -L test_logs.txt

- wc -l *.txt :Count Lines in Multiple Files: Count Lines in Multiple Files.

This command counts the number of lines in all .txt files in the current directory.

wc -l *.txt

- When to Use It:
  When you need to count the total number of lines across multiple files, such as logs or test results.

- cat file.txt | wc -l:Count the Number of Lines Using Command Sequence: command sequence counts the number of lines in file.txt.

This command sequence uses cat to display the contents of file.txt and pipes the output to wc -l, which counts the number of lines in the file.
cat file.txt | wc -l

- When to Use It:
  When you want to count the lines in a file and prefer using a command sequence or piping output between commands.

Grep – Global Regular Expression Print

Grep: It stands for Global Expression Regular Print, this command checks for specified patterns in the entire file. This means it will check the entire file for a particular specified pattern passed to it.
The grep command is used to search for specific patterns in files. It supports powerful regular expressions, making it ideal for filtering and extracting text.

Basic Syntax : grep [options] 'pattern' filename

- grep -n ‘pattern’ ‘filename’:Show line numbers where pattern is matched: Returns number of lines where pattern has been found

grep -n "Hello" filename.txt

Use Case: Quickly locate where the matching text occurs in large files.

- grep -c ‘Hello’ ‘filename’:Count number of matching lines: Returns number of patterns matched.

```
grep -c "Hello" filename.txt
```

Use Case: Get how many times a pattern appears (e.g., error count in logs).

- grep -v ‘Hello’ ‘filename’ : Show lines that do not match the pattern: Will display number of lines where patterns didn't match.

```
grep -v "Hello" filename.txt
```

Use Case: Exclude noise like commented lines or known log messages.

- grep -e ‘Hello’ ‘filename’: Use expression (for multiple patterns) : Will display matches in case sensitive manner.

```
grep -e "Hello" -e "World" filename.txt
```

Use Case: Filter for multiple strings in a single pass.

- grep -i ‘Hello’ ‘filename’:Case-insensitive match: Will display matches in case insensitive manner.

```
grep -i "hello" filename.txt
```

Use Case:When you're unsure of the case used in files (e.g., usernames, config keys).

- grep -r ‘Hello’ ‘filename’: Recursive search in directories : Will find for matches recursively in directories and subdirectories.

```
grep -r "Hello" /path/to/dir
```

Use Case:Search through all files and subfolders, like in source code directories or logs.

- grep -o ‘Hello’ ‘filename’:Only show matched pattern, not the entire line: Will display matches without the lines.

```
grep -o "Hello" filename.txt
```

Use Case:Extract keywords or tokens without surrounding text (e.g., email IDs, IPs).

- grep -w "hello" filename.txt:Match the whole word only (not substrings): Matches only lines where "hello" stands as a whole word, not as a part of another word.

```
grep -w "hello" filename.txt
```

Use Case: Accurate matching — e.g., looking for variable main, not mainPage.

- grep -n "Hello" filename: Displays the matching lines and their line numbers.

- grep "search_string" file1.txt file2.txt:Search in multiple files: Searches for "search_string" across multiple files.
  ```
  grep "search_string" file1.txt file2.txt
  ```

Use Case:Search across logs, config files, or multiple reports

Context Matching with -A, -B, -C
-B (before), -A (after), -C (context)

- grep -A 3 "search_string": grep -A N — Show N lines After the match: filename.txt shows 3 lines after each match.
  ```
  grep -A 2 "error" log.txt
  ```

Use Case: View follow-up logs after an error.

- grep -B 2 "search_string" filename.txt:grep -B N — Show N lines Before the match: shows 2 lines before each match.
  ```
  grep -B 2 "error" log.txt
  ```

Use Case: View logs/events that led to the error.

- grep -C 1 "search_string" filename.grep -C N — Show N lines Before & After match (Context):txt shows 1 line before and after each match.
  ```
  grep -C 1 "error" log.txt
  ```

- grep “^H” f1.txt: Match lines starting with H: Show all that matches H at first character

```
grep "^H" filename.txt
```

Use Case: Find headings, config sections, or patterns at the beginning of lines.

| (pipe)
‘|’ :The | (pipe) in Linux is one of the most powerful tools in the shell. It is used to connect two commands where one command’s output will act as input for another command.

Syntax: command1 | command2
Examples : cat ‘filename’ | grep ‘pattern’ , ps aux | wc -l, ls | grep ‘pattern’

View only the first few lines of long output

```
ls -l | head -n 5
```

Count how many .txt files are in a folder

```
ls *.txt | wc -l
```

Find running processes related to Chrome

```
ps aux | grep chrome
```

Sort a list of files by size and show the largest 3

```
ls -lhS | head -n 3
```

Search for a word in a file and count matches

```
grep 'error' logfile.txt | wc -l
```

List all files and only show those with 'report' in the name

```
ls -l | grep report
```

Chain multiple filters

```
cat data.txt | grep 'failed' | sort | uniq
```

Count how many users

```
who | wc -l
```

Display current processes, filter, and count them

```
ps aux | grep java | wc -l
```

Show only directories from ls

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

List only hidden files in the home directory

```
ls -a ~ | grep ^\.
```

Show the last 10 error logs from a file

```
grep "ERROR" app.log | tail -n 10
```

Get the total size of .log files

```
ls -l *.log | awk '{sum += $5} END {print sum}'
```

Sort a word list alphabetically and remove duplicates

```
cat words.txt | sort | uniq
```

Count the number of files in a directory

```
ls | wc -l
```

List all files, sort them alphabetically

```
ls | sort
```

View the last 5 lines of files containing a keyword

```
grep "success" log.txt | tail -n 5
```

Find most used commands from your history

```
history | awk '{print $2}' | sort | uniq -c | sort -nr | head
```

List only directories

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

What is SED?

SED (Stream Editor) is a powerful stream editor for filtering and transforming text. It allows you to:

- Edit files without opening them in an editor
- Find and replace text using patterns
- Delete, insert, or modify lines
- Process large files efficiently
- Automate text transformations in scripts

SED works by reading input line by line, applying commands, and outputting the result.

Installing SED

```
# SED is usually pre-installed on Ubuntu
sed --version

# If not installed
sudo apt update
sudo apt install sed
```

Basic SED Syntax

```
sed [options] 'command' file
sed [options] 'address/pattern/replacement/flags' file
sed [options] -e 'command1' -e 'command2' file
sed [options] -f script_file file
```

Common Structure:

```
sed 's/old_text/new_text/g' filename
#    │ │        │        │ │
#    │ │        │        │ └── flags (g = global)
#    │ │        │        └───── new text
#    │ │        └─────────────────── old text/pattern
#    │ │
#   │  └──────────────────────────────── delimiter
#   └────────────────────────────────────── substitute command
```

Basic Text Substitution

1. Simple Find and Replace

```
# Replace first occurrence per line
sed 's/old/new/' file.txt

# Replace all occurrences (global)
sed 's/old/new/g' file.txt

# Replace and save to new file
sed 's/old/new/g' input.txt > output.txt

# Replace in-place (modify original file)
sed -i 's/old/new/g' file.txt

# Create backup before in-place edit
sed -i.bak 's/old/new/g' file.txt
```

2. Using Different Delimiters

```
# Default delimiter (/)
sed 's/old/new/g' file.txt

# Using different delimiters (useful for paths)
sed 's|/old/path|/new/path|g' file.txt
sed 's#old#new#g' file.txt
sed 's@old@new@g' file.txt
```

3. Case Sensitivity

```
# Case-sensitive (default)
sed 's/Hello/Hi/g' file.txt

# Case-insensitive
sed 's/hello/Hi/gi' file.txt
sed 's/HELLO/Hi/gi' file.txt
```

---

1. Line Numbers

```
# Edit specific line
sed '3s/old/new/' file.txt              # Line 3 only
sed '1,5s/old/new/' file.txt            # Lines 1 to 5
sed '2,$s/old/new/' file.txt            # Line 2 to end
sed '1~2s/old/new/' file.txt            # Every 2nd line starting from 1

# Multiple line ranges
sed -e '1,3s/old/new/' -e '10,15s/foo/bar/' file.txt
```

2. Pattern Matching

```
# Lines containing pattern
sed '/pattern/s/old/new/' file.txt

# Lines NOT containing pattern
sed '/pattern/!s/old/new/' file.txt

# Between two patterns
sed '/start_pattern/,/end_pattern/s/old/new/' file.txt
```

3. Combining Addresses

```
# Line number AND pattern
sed '5,/pattern/s/old/new/' file.txt

# First occurrence of pattern
sed '0,/pattern/s/old/new/' file.txt
```

---

1. Delete Lines

```
# Delete specific lines
sed '3d' file.txt                    # Delete line 3
sed '1,5d' file.txt                  # Delete lines 1-5
sed '$d' file.txt                    # Delete last line

# Delete lines matching pattern
sed '/pattern/d' file.txt            # Delete lines containing "pattern"
sed '/^$/d' file.txt                 # Delete empty lines
sed '/^\s*$/d' file.txt              # Delete blank lines (with whitespace)
```

2. Insert and Append

```
# Insert before line
sed '3i\New line before line 3' file.txt

# Append after line
sed '3a\New line after line 3' file.txt

# Insert at beginning
sed '1i\Header line' file.txt

# Append at end
sed '$a\Footer line' file.txt
```

3. Print Operations

```
# Print specific lines (suppress others with -n)
sed -n '1,5p' file.txt                  # Print lines 1-5
sed -n '/pattern/p' file.txt            # Print lines with pattern

# Print and delete
sed -n '1,5p; 1,5d' file.txt            # Print then delete lines 1-5
```

---

1. Basic Patterns

```
# Any character
sed 's/a.c/ABC/g' file.txt              # Match a[any]c

# Start/end of line
sed 's/^/PREFIX: /' file.txt            # Add prefix to each line
sed 's/$/ SUFFIX/' file.txt             # Add suffix to each line

# Word boundaries
sed 's/\bword\b/WORD/g' file.txt        # Match whole word only
```

2. Character Classes

```
# Digits
sed 's/[0-9]/X/g' file.txt              # Replace any digit with X
sed 's/[0-9]\+/NUMBER/g' file.txt       # Replace sequences of digits

# Letters
```

```
sed 's/[a-zA-Z]/X/g' file.txt          # Replace any letter
sed 's/[A-Z]/lower/g' file.txt         # Replace uppercase letters

# Whitespace
sed 's/\s\+/ /g' file.txt              # Replace multiple spaces with one
sed 's/^\s*//' file.txt                # Remove leading whitespace
sed 's/\s*$//' file.txt                # Remove trailing whitespace
```

3. Quantifiers

```
# Zero or more
sed 's/a*/X/g' file.txt                # Match zero or more 'a'

# One or more
sed 's/a\+/X/g' file.txt               # Match one or more 'a'

# Specific counts
sed 's/a\{3\}/XXX/g' file.txt          # Match exactly 3 'a's
sed 's/a\{2,5\}/X/g' file.txt          # Match 2 to 5 'a's
```

Practical Examples

1. File Processing

```
# Remove comments from config file
sed 's/#.*$//' config.conf

# Remove empty lines and comments
sed '/^#/d; /^$/d' config.conf

# Add line numbers
sed = file.txt | sed 'N;s/\n/\t/'

# Convert Windows line endings to Unix
sed 's/\r$//' windows_file.txt > unix_file.txt

# Convert tabs to spaces
sed 's/\t/    /g' file.txt
```

2. Log File Processing

```
# Extract specific date range
```

```
sed -n '/2024-01-01/,/2024-01-31/p' logfile.log

# Remove timestamps
sed 's/^[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9] [0-9][0-9]:[0-9][0-9]:[0-9][0-9] //' log.txt

# Extract IP addresses
sed -n 's/.*\([0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\).*/\1/p' access.log

# Count error lines
sed -n '/ERROR/p' logfile.log | wc -l
```

3. Data Transformation

```
# CSV manipulation - swap columns 1 and 2
sed 's/\([^,]*\),\([^,]*\),\(.*\)/\2,\1,\3/' data.csv

# Convert comma to tab
sed 's/,/\t/g' data.csv

# Remove quotes from CSV
sed 's/"//g' data.csv

# Extract email addresses
sed -n 's/.*\([a-zA-Z0-9._%+-]\+@[a-zA-Z0-9.-]\+\.[a-zA-Z]\{2,\}\).*/\1/p' file.txt
```

Advanced SED Features

1. Hold Space (Advanced)

```
# Reverse file line order
sed '1!G;h;$!d' file.txt

# Print every other line
sed 'n;d' file.txt

# Duplicate each line
sed 'p' file.txt

# Join lines with comma
sed ':a;N;$!ba;s/\n/,/g' file.txt
```

2. Multiple Commands

```
# Chain commands with semicolon
sed 's/old/new/g; s/foo/bar/g' file.txt

# Use -e for multiple expressions
sed -e 's/old/new/g' -e 's/foo/bar/g' file.txt

# Use script file
cat > script.sed << EOF
s/old/new/g
s/foo/bar/g
/^$/d
EOF
sed -f script.sed file.txt
```

3. Labels and Branching

```
# Label and branch (advanced)
sed ':a;s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta' numbers.txt  # Add commas to numbers

# Conditional execution
sed '/pattern/{s/old/new/g;}' file.txt
```

What is UNIQ?

UNIQ is a command-line utility that reports or omits repeated lines. It can:

- Remove duplicate consecutive lines from input
- Count occurrences of duplicate lines
- Show only unique or only duplicate lines
- Compare lines ignoring case or specific fields
- Process large text files efficiently

Important: UNIQ only works on consecutive duplicate lines, so input is usually sorted first.

Installing UNIQ

```
# UNIQ is part of GNU coreutils (pre-installed on Ubuntu)
uniq --version

# If not available (rare)
```

```
sudo apt update
sudo apt install coreutils
```

Basic UNIQ Syntax

```
uniq [options] [input_file] [output_file]

# Common usage patterns
uniq file.txt                  # Remove consecutive duplicates
sort file.txt | uniq           # Remove all duplicates
sort file.txt | uniq -c        # Count occurrences
sort file.txt | uniq -d        # Show only duplicates
```

Remove Consecutive Duplicates

```
# Sample file with consecutive duplicates
cat > sample.txt << EOF
apple
apple
banana
banana
banana
cherry
apple
apple
EOF

# Remove consecutive duplicates
uniq sample.txt
# Output:
# apple
# banana
# cherry
# apple

# Note: Last "apple" remains because it's not consecutive with first ones
```

Remove ALL Duplicates (with sort)

```
# Remove all duplicates (sorted output)
sort sample.txt | uniq
# Output:
# apple
# banana
# cherry

# Keep original order while removing duplicates (using awk)
awk '!seen[$0]++' sample.txt
# Output:
# apple
# banana
# cherry
```

Basic File Processing

```
# Process file and save to new file
uniq input.txt output.txt

# Process multiple files
cat file1.txt file2.txt | sort | uniq

# Use with pipes
sort data.txt | uniq | head -10
```

---

1. Count Duplicates (-c)

```
# Count occurrences of each line
sort sample.txt | uniq -c
# Output:
#       3 apple
#       3 banana
#       1 cherry

# Sort by count (most frequent first)
sort sample.txt | uniq -c | sort -nr
# Output:
#       3 banana
#       3 apple
#       1 cherry
```

2. Show Only Duplicates (-d)

```
# Show only lines that appear more than once
sort sample.txt | uniq -d
# Output:
# apple
# banana

# Count duplicates only
sort sample.txt | uniq -cd
# Output:
#       3 apple
#       3 banana
```

3. Show Only Unique Lines (-u)

```
# Show only lines that appear exactly once
sort sample.txt | uniq -u
# Output:
# cherry

# Count unique lines
sort sample.txt | uniq -cu
# Output:
#       1 cherry
```

---

1. Case Insensitive (-i)

```
# Sample file with mixed case
cat > mixed_case.txt << EOF
Apple
apple
APPLE
Banana
banana
Cherry
EOF
```

```
# Case-insensitive duplicate removal
sort mixed_case.txt | uniq -i
# Output:
# Apple
# Banana
# Cherry

# Count case-insensitive
sort mixed_case.txt | uniq -ci
# Output:
#       3 Apple
#       2 Banana
#       1 Cherry
```

2. Skip Fields (-f) and Characters (-s)

```
# Sample file with structured data
cat > structured.txt << EOF
1 apple red
2 apple green
3 banana yellow
4 banana yellow
5 cherry red
EOF

# Skip first field (ignore numbers)
sort structured.txt | uniq -f 1
# Output:
# 1 apple red
# 5 cherry red

# Skip first 2 characters
sort structured.txt | uniq -s 2
# Output based on characters after position 2

# Combine: skip 1 field and be case insensitive
sort structured.txt | uniq -f 1 -i
```

3. Check Only First N Characters (-w)

```
# Sample file
```

```
cat > prefixes.txt << EOF
apple123
apple456
banana789
cherry000
EOF

# Compare only first 5 characters
sort prefixes.txt | uniq -w 5
# Output:
# apple123
# banana789
# cherry000

# Count based on first 6 characters
sort prefixes.txt | uniq -cw 6
# Output:
#       2 apple123
#       1 banana789
#       1 cherry000
```

Real-World Examples

1. Log File Analysis

```
# Extract unique IP addresses from access log
awk '{print $1}' access.log | sort | uniq

# Count requests per IP
awk '{print $1}' access.log | sort | uniq -c | sort -nr

# Top 10 most frequent IPs
awk '{print $1}' access.log | sort | uniq -c | sort -nr | head -10

# Find IPs with only one request
awk '{print $1}' access.log | sort | uniq -u
```

2. User Data Processing

```
# Find duplicate usernames in /etc/passwd
cut -d: -f1 /etc/passwd | sort | uniq -d
```

```
# Count unique shells used
cut -d: -f7 /etc/passwd | sort | uniq -c

# Find users with duplicate UIDs
cut -d: -f3 /etc/passwd | sort | uniq -d
```

Text Analysis

```
# Word frequency analysis
tr ' ' '\n' < document.txt | tr '[:upper:]' '[:lower:]' | sort | uniq -c | sort -nr

# Find unique words only
tr ' ' '\n' < document.txt | sort | uniq -u

# Count unique lines in multiple files
cat *.txt | sort | uniq -c
```

Sort

What is SORT?

SORT is a powerful command-line utility that arranges lines of text in a specified order. It can:

- Sort lines alphabetically or numerically
- Sort by specific fields or columns
- Handle different data types (numbers, dates, IP addresses)
- Remove duplicates while sorting
- Merge pre-sorted files efficiently
- Work with large files using external sorting

SORT is essential for data processing, log analysis, and preparing data for other tools like uniq.

Installing SORT

```
# SORT is part of GNU coreutils (pre-installed on Ubuntu)
sort --version

# If not available (very rare)
sudo apt update
sudo apt install coreutils
```

Basic SORT Syntax

```
sort [options] [file...]

# Basic usage patterns
sort file.txt                    # Alphabetical sort
sort -n numbers.txt              # Numerical sort
sort -r file.txt                 # Reverse sort
sort -u file.txt                 # Sort and remove duplicates
```

Basic Sorting Examples

1. Alphabetical Sorting

```
# Sample data
cat > names.txt << EOF
Charlie
Alice
Bob
David
alice
EOF

# Default alphabetical sort
sort names.txt
# Output:
# Alice
# Bob
# Charlie
# David
# alice

# Case-insensitive sort
sort -f names.txt
# Output:
# alice
# Alice
# Bob
# Charlie
# David
```

2. Numerical Sorting

```bash
# Sample numbers
cat > numbers.txt << EOF
100
20
3
1000
5
EOF

# Wrong: Alphabetical sort of numbers
sort numbers.txt
# Output: 100, 1000, 20, 3, 5 (lexicographic)

# Correct: Numerical sort
sort -n numbers.txt
# Output: 3, 5, 20, 100, 1000
```

3. Reverse Sorting

```
# Reverse alphabetical
sort -r names.txt

# Reverse numerical
sort -nr numbers.txt

# Reverse with case-insensitive
sort -rf names.txt
```

---

1. Basic Options

Option	Description	Example
-n	Numerical sort	sort -n numbers.txt
-r	Reverse order	sort -r file.txt
-u	Unique (remove duplicates)	sort -u file.txt

-f	Case-insensitive	sort -f names.txt
-b	Ignore leading blanks	sort -b file.txt
-d	Dictionary order (alphanumeric + whitespace)	sort -d file.txt

2. Advanced Options

Option	Description	Example
-k	Sort by field/column	sort -k2 file.txt
-t	Field separator	sort -t',' -k2 file.csv
-o	Output file	sort file.txt -o sorted.txt
-m	Merge sorted files	sort -m file1.txt file2.txt
-c	Check if sorted	sort -c file.txt
-V	Version sort	sort -V versions.txt

Field-Based Sorting

1. Basic Field Sorting

```
# Sample data with fields
cat > data.txt << EOF
John 25 Engineer
Alice 30 Manager
Bob 20 Intern
Charlie 35 Director
EOF

# Sort by second field (age)
sort -k2 data.txt
# Output sorted by age as strings: 20, 25, 30, 35

# Sort by second field numerically
sort -k2n data.txt
# Output:
# Bob 20 Intern
# John 25 Engineer
# Alice 30 Manager
# Charlie 35 Director

# Sort by third field (job title)
sort -k3 data.txt
```

2. CSV and Delimited Data

```
# Sample CSV data
cat > employees.csv << EOF
Name,Age,Salary,Department
John,25,50000,Engineering
Alice,30,75000,Management
Bob,20,35000,Engineering
Charlie,35,90000,Executive
EOF

# Sort by age (second field, comma-separated)
sort -t',' -k2n employees.csv

# Sort by salary (third field, skip header)
tail -n +2 employees.csv | sort -t',' -k3n | head -1 employees.csv -

# Sort by department, then by salary
tail -n +2 employees.csv | sort -t',' -k4,4 -k3n
```

3. Complex Field Specifications

```
# Sample data with multiple fields
cat > complex_data.txt << EOF
2023-01-15 10:30 ERROR Server down
2023-01-15 09:45 INFO Backup complete
2023-01-15 10:30 WARN Low disk space
2023-01-14 23:59 ERROR Database error
EOF

# Sort by date (field 1), then time (field 2)
sort -k1,1 -k2,2 complex_data.txt

# Sort by log level (field 3), then by time (field 2)
sort -k3,3 -k2,2 complex_data.txt
```

```
# Sort by field range (fields 1-2 combined)
sort -k1,2 complex_data.txt
```

---

1. Numerical Variations

```
# Human-readable numbers (K, M, G suffixes)
cat > sizes.txt << EOF
100K
1.5M
2G
500K
1.2G
EOF
# Human-readable sort
sort -h sizes.txt
# Output: 100K, 500K, 1.5M, 1.2G, 2G
# General numeric sort (handles scientific notation)
cat > scientific.txt << EOF
1.5e10
2.3e5
1.2e12
5.7e8
EOF

sort -g scientific.txt
```

2. Version Sorting

```
# Software versions
cat > versions.txt << EOF
v1.10.2
v1.2.1
v1.10.10
v2.1.0
v1.9.5
EOF

# Version-aware sort
sort -V versions.txt
# Output: v1.2.1, v1.9.5, v1.10.2, v1.10.10, v2.1.0
# Natural sort (similar to version sort)
sort -n versions.txt  # Wrong approach
sort -V versions.txt  # Correct for versions
```

3. Date and Time Sorting

```
# ISO date format
cat > dates.txt << EOF
2023-12-01
2023-01-15
2023-12-15
2023-02-28
EOF

# Dates sort correctly with default sort
sort dates.txt
# Time sorting (if mixed with dates)
cat > datetime.txt << EOF
2023-01-15 14:30:00
2023-01-15 09:15:30
2023-01-14 23:59:59
2023-01-15 14:30:01
EOF
# Sort by date and time
sort datetime.txt
```

Find Command

find(imp) : It is used to search files and directories within specified directory hierarchy, it searches for files on criteria such as name, size, permission, timestamp.

Syntax : find ‘path’ -’name’ ‘filename’, this finds file with specific name.
find [path] [options] [expression]

- find . :Search in current directory: Lists all files and directories in the current directory.

```
find .
```

- find ‘path’ -type d: Find all directories this will find directories.Lists only directories under the given path.

```
find /path -type d
```

- find ‘path’ -type f: this will find files. Lists only regular files.

```
find /path -type f
```

- find /’filename’, this will search files in the entire system. searches from the root / down.

```
find / -name "filename"
```

- find ~’filename’: Search in home directory:This will search in the home directory.Searches in the current user's home directory.

```
find ~ -name "filename"
```

- find. ‘filename’:Search in current directory: This will search in current directory. Searches for the filename in the current directory and subdirectories.

```
find . -name "filename"
```

- find /home -type f -mtime -10: Find files modified in last 10 days: find in home directory as per modifications in last 10 days.

```
find /home -type f -mtime -10
```

Useful for checking recently edited files.

- find / -type f -atime -1:Find files accessed in the last 24 hours: -atime checks last access time (not modified).find files accessed in the last 24 hours

```
find / -type f -atime -1
```

- find /home -type f -perm -777: Find files with 777 permissions: find in home directory as per read,write and execute permission.

```
find /home -type f -perm 0777
```

Security check — list world-readable/writable/executable files.
Size-based Searching:

- Find files larger than 1 KB :find /home -type f -size +1k, find in home directory as per size of files exceeding 1KB.
- Find files larger than 100 MB: find / -size +100M, find files larger than 100MB.

Empty Files and Directories:

- Find empty directories:find /home -type d -empty, find in home directory for directories that are empty.
- Find empty files:find . -type f -empty, find in home directory for files that are empty.

- Find .txt files modified in the last 1 day: find path -type f -name "*.txt" -mtime -1: will show on the specified path, file extension txt that have been modified in the last 1 day.

Difference between find and grep

Find	Grep
It is used to find the files based on different attributes like name,size, permission, type	It is used to find pattern within the files
We don't use any kind of regular expression	We use regular expression to find the patterns

---

---

locate – Fast File & Directory Finder
Locate: This command as the name suggests will locate any file / directory.The locate command is used to quickly search for files and directories on a Linux system

Syntax: Locate ‘filename’/‘directoryname’ :

- Locate any file or directory by name: Finds all matches for filename.txt across the system.

locate filename.txt

- Locate a directory: Locate -r ‘‘directoryname’: It is used to locate directories .
  locate -r "/project_name$"

- Locate files owned by a specific user :Locate -u ‘username’, It is used to locate files and directories owned by specified users.
  Use Case: Security audits or checking file ownership.
- Locate recently updated files:Locate -e ‘filename’, It will locate all the update files .Shows only files that exist now (some old results may be stale in default locate output).
- Case-insensitive search : locate -i filename : -i option makes the search case insensitive.
  locate -i report.txt

- Limit number of results:locate -n 10 filename : n option limits the output to the specified number of results. In this case, it will show only the first 10 matches.
- Use regular expressions in search:locate --regex '.*\.log$'

locate --regex 'filename$' : --regex option allows you to use regular expressions to refine your search. This example finds files that end with "filename.

```
locate --regex '/home/.*/documents/.*report.*\.pdf$'
```

Difference between locate and find :

Locate	Find
It is used to search files and directories based on their location	It is used to search file name and permissions based on their different attributes
Comparatively faster than find	Find is slower than locate
Is able to find the files even after they are deleted	Find can't search for deleted files

---

Create a New User

Run the following command as root or with sudo:

```
sudo useradd -m cdac
```

- -m creates a home directory at /home/john

Set a Password for the User

```
sudo passwd cdac
```

Verify User Creation

```
id cdac
getent passwd cdac
```

Add the User to a Group

```
sudo usermod -aG developers mohan
```

Create user with specific UID

sudo useradd -m -u 1090 mohan

Create user with custom home directory
sudo useradd -m -d /home/customcdac mohan

Create user and add to a specific group
sudo useradd -m -G developers Mohan

Create user with comment (full name)
sudo useradd -m -c "CDAC Mumbai" Cdac

To remove a user from a group in Linux

Using gpasswd (Recommended)
Command Syntax
sudo gpasswd -d <username> <groupname>
Example
sudo gpasswd -d cdac developers
Manual Group File Editing

Open group file:
sudo nano /etc/group

# Delete user account only (keeps home directory)

sudo userdel username

# Delete user account AND home directory

sudo userdel -r username

# Force delete (even if user is logged in)

sudo userdel -f username

# Example

sudo userdel -r mohan

Managing Users & Groups

List All Users:

1. cat /etc/passwd
2. getent passwd

ACL (Access Control List)

ACL allows fine-grained permissions beyond the default u/g/o.

View ACL:

```
getfacl filename
```

View ACL for a specific user:

```
getfacl -u username filename
```

Set ACL for User:

```
setfacl -m u:username:rw filename
```

To check : getfacl filename

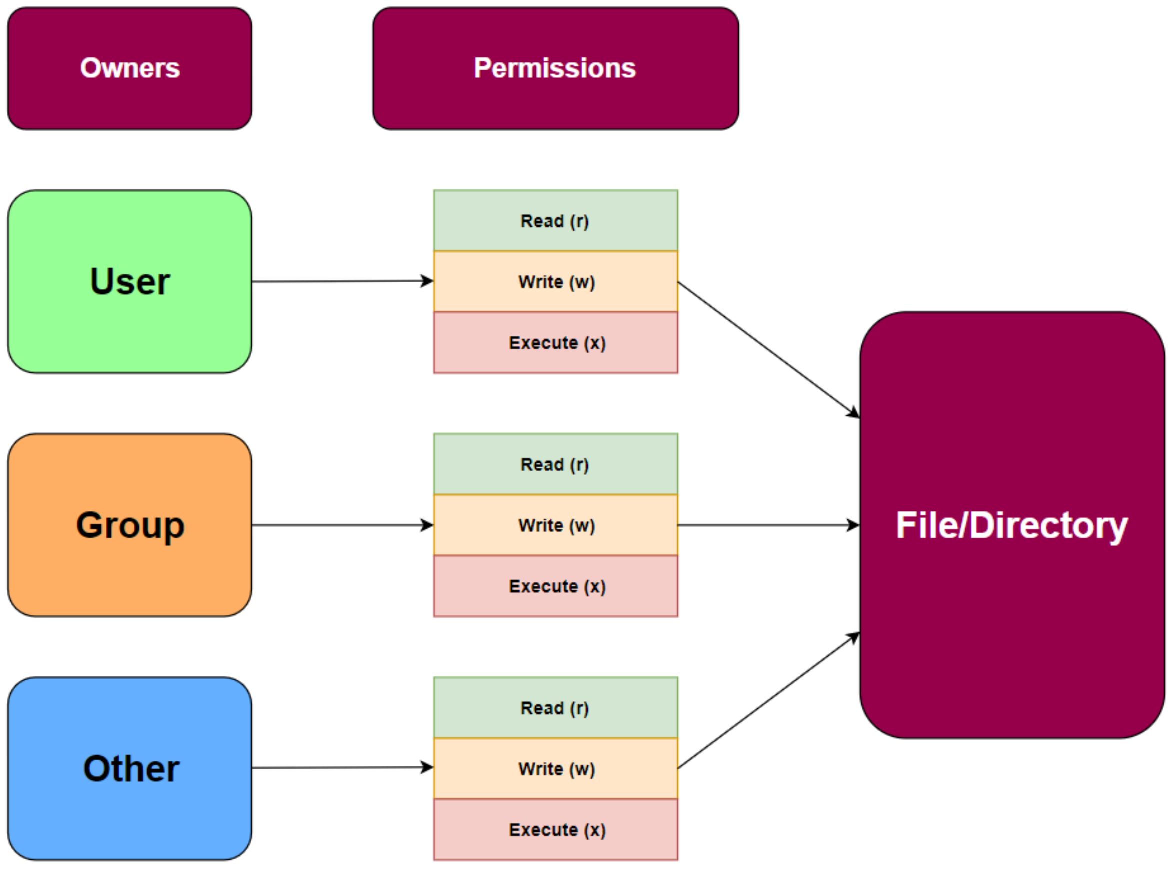
Set ACL for Group:

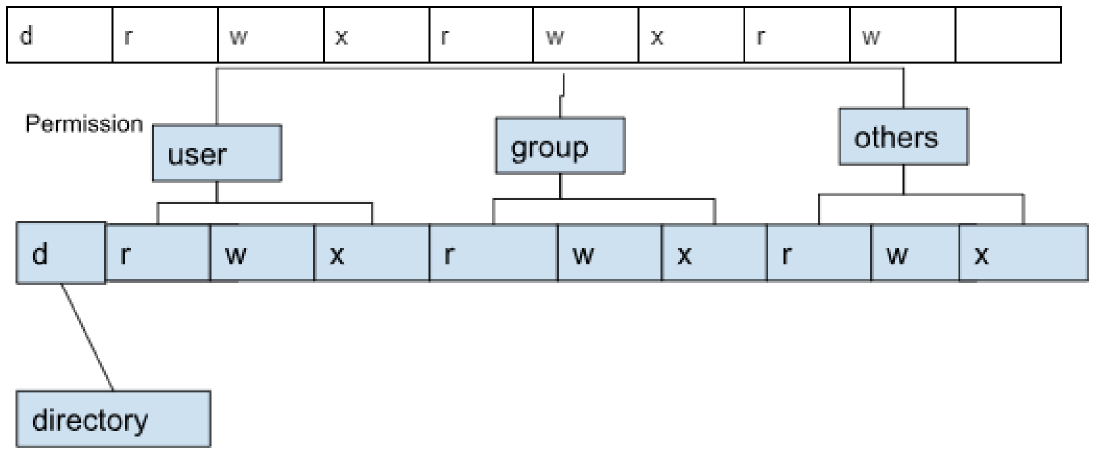
```
setfacl -m g:groupname:rx filename
```

Remove All ACL Entries:

```
setfacl -b filename
```

File Permission and owners





7	r	w	x
6	r	w	-
5	r	-	x
4	r	-	-
3	-	w	x
1	-	-	x
0	-	-	-

		read,write,and execute the file. Groups and others don't have access to any file.
666	rw- rw- rw-	Read , write permission is given to the owner , group and others. No one is having access to execute the file.
664	rw- rw- r- -	Read , write permission is given to the owner and group. Whereas , read only permission is given to others.
644	rw- r- - r - -	Read and write permission is given to the owner . Read only permission is given to group and others
640	rw- r- - - - -	Read, write permission is given to the owner. Read only permission is given to group and there is no permission given to others.
600	rw- - - - - - -	Only user has the read,write permission , Group and others have no permission.
400	r - - - - - - - -	Owner has the permission to just read. Groups and others have no permission.

Note : Important question for module end exam

In Linux , File authorization is divided into 2 parts:

1. Permission
2. Ownership

Linux File Ownership: This gives the information about file/directory owner(this means file belongs to which owner or group).
This is further divided into 3 parts:

1. User
2. Groups
3. Others

Users:
Users are those who own the file . By default those who create the file are the owner.These owners holds some attribute like
Id: this will print user id, group id, others id and these id have some predefined meaning.
I.e 0 means the ownership is with root and the 0 has been reserved for the root.

UserID: This is also called as user identity or uid , that is assigned to the user to identify the type of the users and to understand the system resources utilizations.
There are range for these id:
0: reserved for root.
1-99:reserved for predefined accounts.
100-999: reserved for system administrator and accounts.
1000-10000: reserved for application accounts.
Above 10000: reserved for user accounts.
We can change user permission using symbolic characters too

Syntax : chmod u=rx,g=r,o=x file1.txt

Another convection using +ve
chmod u+rwx,g+rx,o+w file1.txt
Similarly using -ve removes the permissions
```
                    chmod u-rw,g-w,o-w file1.txt
```

Groups: A group contains multiple users. All the users belonging to a group have the same set of permission access for the file.

Group permissions are owned by the group that owns the file/ directories.

Like Users , groups have some attribute called gid or group identity., that determine thes system resources for the group

There are range for these id:
0: reserved for root.
1-99:reserved for system and applications.
100-above: allocated to users groups.

Others: All users who are not:

- the file's owner

- part of the file's group

They are essentially general users with no special privileges on that file.

Permissions in Linux

Each file/directory has 3 types of permissions for:

- User (u)
- Group (g)
- Others (o)

Permissions:

- r – Read
- w – Write
- x – Execute

Change Permissions with chmod

Symbolic Method:

```
chmod u=rx,g=r,o=x file.txt
```

Set read+execute for user, read for group, execute for others.

Add Permissions:

```
chmod u+rwx,g+rx,o+w file.txt
```

Remove Permissions:

```
chmod u-rw,g-w,o-w file.txt
```

Check Current Permissions:

```
ls -l file.txt
```

Example output:

```
-rwxr-xr-- 1 user group 1234 Mar 24 12:00 file.txt
```

What is the purpose of chmod?

chmod is used to change file permissions (read, write, execute) for the user, group, or others.

Give full access to the owner, read/execute to group, no access to others

```
chmod 750 file.txt
```

Give read/write to everyone (not secure)

```
chmod 666 file.txt
```

---

3. Make a script executable by everyone

```
chmod a+x script.sh
```

---

4. Remove write permission from group and others

```
chmod go-w file.txt
```

---

5. Add execute permission to owner only

```
chmod u+x file.txt
```

---

6. Recursive permission change on a directory

```
chmod -R 755 /path/to/dir
```

The chgrp command is a fundamental Linux utility used to modify the group ownership of files and directories. Group ownership determines which user groups can access or modify a specific file or directory.

Command Syntax

```
chgrp [OPTIONS] GROUP FILE/DIRECTORY
```

1. Change Group of a Single File

```
sudo chgrp developers report.txt
```

- Changes the group of report.txt to the developers group
- Requires sudo for system files or when you don't own the file

2. Change Group of a Directory

```
sudo chgrp devteam /var/www/html
```

- Transfers group ownership of the entire /var/www/html directory to devteam

3. Recursive Group Change

```
sudo chgrp -R devteam /home/project
```

- The -R (recursive) flag changes group ownership for all files and subdirectories
- Useful for entire project folders or shared development spaces

Verifying Group Ownership

Use ls -l to view current group ownership:

```
ls -l report.txt
```

- Shows developers as the group owner

Strategic Use Cases

When to Use chgrp

- Team Collaboration: Enable shared file access across team members
- Project Organization: Manage file permissions based on team structures
- Security Management: Control access to sensitive project resources
- Development Workflows: Streamline permissions for shared repositories

What is the purpose of chown?

chown is used to change the ownership of a file or directory — i.e., assign a new user or group.

Change file owner to user john

```
sudo chown cdac file.txt
```

---

2. Change owner and group

```
sudo chown cdac:DataAnalyst file.txt
```

---

3. Change ownership of a directory recursively

```
sudo chown -R cdac:cdac /home/cdac/project
```

---

4. Change only group ownership

```
sudo chown :designers file.txt
```

---

5. Revert ownership back to root

```
sudo chown root:root file.txt
```

Can both commands be used together?

```
sudo chown cdac:dbda file.txt && chmod 740 file.txt
```

What is umask in Linux?

umask stands for User File Creation Mask. It determines the default permissions for newly created files and directories by subtracting its value from the system's full permission set.

Default Permission Reference:

Type	Default Max Permissions
Files	666 (rw-rw-rw-)
Directories	777 (rwxrwxrwx)

Note: Executable (x) permission is not given to files by default, unless explicitly set.

How umask Works
Formula:
Final Permission = Default Permission - umask

Example:

If umask = 022:

- For files → 666 - 022 = 644 → rw-r--r--
- For directories → 777 - 022 = 755 → rwxr-xr-x

Check Current umask

```
umask
```

Example Output:

```
0022
```

Set a Temporary umask

```
umask 007
```

- This would restrict others' access
  Files: 666 - 007 = 660 (rw-rw----)

Dirs: 777 - 007 = 770 (rwxrwx---)

Use Cases of umask

- Control default permissions for security
- Prevent unauthorized read/write by others
- Ensure application-created files have safe permissions
- Custom shell setups for multi-user systems

What is SSH?

SSH (Secure Shell) is a cryptographic network protocol that allows you to:

- Securely connect to remote machines over an unsecured network
- Execute commands on remote servers
- Transfer files securely
- Create secure tunnels for other protocols

All data is encrypted during transmission, making it safe even over public networks.

Installing SSH on Ubuntu

Install SSH Client (to connect to other machines)

```
sudo apt update
sudo apt install openssh-client
```

Install SSH Server (to allow others to connect to your machine)

```
sudo apt update
sudo apt install openssh-server

# Start and enable SSH service
sudo systemctl start ssh
sudo systemctl enable ssh

# Check if SSH is running
sudo systemctl status ssh
```

Connecting to Your Specific IP (10.0.2.15)

Basic Connection

```
ssh username@10.0.2.15
ssh john@10.0.2.15
ssh your_username@10.0.2.15
```

- You'll see a fingerprint warning (type yes to continue)
- Enter the password when prompted
- You're now connected to the remote machine!

Check What's on the Remote Machine

```
# After connecting via SSH
ls -la          pwd          whoami          uname -a
```

Custom Port

```
# If SSH server runs on port 2222 instead of default 22
ssh -p 2222 username@10.0.2.15
```

Troubleshooting Common Issues

Connection Refused

```
# Check if SSH server is running on target machine
sudo systemctl status ssh

# Check if port 22 is open
sudo ufw status
sudo ufw allow ssh
```

Permission Denied

```
# Ensure user exists on remote machine
# Check SSH server configuration
sudo nano /etc/ssh/sshd_config
# Restart SSH service after changes
sudo systemctl restart ssh
```

Host Key Verification Failed

```
# Remove old key and try again
ssh-keygen -R 10.0.2.15
```

SCP (Secure Copy Protocol) - Complete Guide

What is SCP?

SCP (Secure Copy Protocol) is a secure file transfer utility that uses SSH for data transfer. It provides:

- Encrypted file transfers between local and remote machines
- Authentication using SSH keys or passwords
- Preservation of file permissions and timestamps
- Recursive directory copying

Basic Syntax

```
scp [options] source destination
General Format:
scp [local_file] [user@remote_host]:[remote_path]       # Local to Remote
scp [user@remote_host]:[remote_file] [local_path]       # Remote to Local
```

Practical Examples with Your IP (10.0.2.15)

1. Copy File FROM Local TO Remote

```
# Copy a single file
scp /home/user/document.txt username@10.0.2.15:/home/username/

# Copy to specific directory
scp ./myfile.pdf username@10.0.2.15:/tmp/

# Copy and rename file
scp report.docx username@10.0.2.15:/home/username/final_report.docx
```

2. Copy File FROM Remote TO Local

```
# Copy file from remote to current directory
scp username@10.0.2.15:/home/username/remote_file.txt ./

# Copy to specific local directory
scp username@10.0.2.15:/var/log/syslog /home/user/Downloads/
```

```
# Copy and rename locally
scp username@10.0.2.15:/home/username/data.csv ./backup_data.csv
```

3. Copy Multiple Files

```
# Multiple files to remote
scp file1.txt file2.txt file3.txt username@10.0.2.15:/home/username/

# Using wildcards
scp *.jpg username@10.0.2.15:/home/username/pictures/

# Multiple files from remote
scp username@10.0.2.15:"/home/username/{file1,file2,file3}" ./
```

Directory Operations

1. Copy Entire Directory (Recursive)

```
# Copy directory to remote
scp -r /home/user/project/ username@10.0.2.15:/home/username/

# Copy directory from remote
scp -r username@10.0.2.15:/home/username/backup/ ./local_backup/

# Copy directory contents only
scp -r /home/user/project/* username@10.0.2.15:/home/username/new_project/
```

2. Preserve File Attributes

```
# Keep original permissions, timestamps, etc.
scp -p important_file.sh username@10.0.2.15:/home/username/

# Recursive with preserved attributes
scp -rp /home/user/webapp/ username@10.0.2.15:/var/www/
```

Common SCP Options

Option	Description	Example
-r	Recursive (for directories)	scp -r folder/ user@host:/path/

-p	Preserve file attributes	scp -p file.txt user@host:/path/
-v	Verbose (show progress)	scp -v file.txt user@host:/path/
-C	Enable compression	scp -C largefile.zip user@host:/path/
-P	Specify port	scp -P 2222 file.txt user@host:/path/
-i	Use specific SSH key	scp -i ~/.ssh/id_rsa file.txt user@host:/path/
-q	Quiet mode (no progress)	scp -q file.txt user@host:/path/

Real-World Ubuntu Examples

1. Backup System Files

```
# Backup important config files
scp /etc/nginx/nginx.conf username@10.0.2.15:/home/username/backups/

# Backup multiple config files
scp /etc/{hosts,fstab,passwd} username@10.0.2.15:/home/username/system_backup/

# Backup entire directory
scp -r /home/user/.config username@10.0.2.15:/home/username/config_backup/
```

2. Web Development Workflow

```
# Upload website files
scp -r /home/user/website/* username@10.0.2.15:/var/www/html/

# Download logs for analysis
scp username@10.0.2.15:/var/log/apache2/access.log ./logs/

# Upload database backup
scp database_backup.sql username@10.0.2.15:/tmp/
```

3. Development File Sync

```
# Upload source code
scp -r ./src/ username@10.0.2.15:/home/username/project/src/

# Download compiled binaries
scp username@10.0.2.15:/home/username/project/build/app ./
```

```
# Sync configuration files
scp .env username@10.0.2.15:/home/username/project/
```

Advanced SCP Usage

1. Using Custom SSH Port

```
# If SSH runs on port 2222
scp -P 2222 file.txt username@10.0.2.15:/home/username/
```

2. Using SSH Keys

```
# Use specific private key
scp -i ~/.ssh/id_rsa file.txt username@10.0.2.15:/home/username/
# Multiple options combined
scp -i ~/.ssh/id_rsa -P 2222 -r ./project/ username@10.0.2.15:/home/username/
```

3. Progress and Compression

```
# Show verbose progress with compression
scp -Cv large_file.tar.gz username@10.0.2.15:/home/username/

# Quiet mode for scripts
scp -q backup.tar username@10.0.2.15:/backups/
```

4. Copy Between Remote Hosts

```
# Copy from one remote host to another
scp user1@10.0.2.15:/path/file.txt user2@192.168.1.100:/path/

# Through your local machine (3rd party transfer)
scp -3 user1@10.0.2.15:/path/file.txt user2@192.168.1.100:/path/
```

What is ZIP?

ZIP is a compression and archiving utility that:

- Compresses files to save disk space and transfer time
- Archives multiple files into a single .zip file
- Maintains file permissions and directory structure
- Cross-platform compatible (Windows, Mac, Linux)
- Password protection support

```
# Install zip and unzip utilities
sudo apt update
sudo apt install zip unzip

# Check if already installed
zip --version
unzip --version
# Compress single file
zip document.zip report.pdf

# Compress multiple files
zip documents.zip file1.txt file2.txt file3.pdf

# Compress all files in current directory
zip all_files.zip *

# Compress specific file types
zip images.zip *.jpg *.png *.gif
# Compress entire directory
zip -r project.zip /home/user/my_project/

# Compress directory contents (not the directory itself)
cd /home/user/my_project/
zip -r ../project_contents.zip *

# Compress multiple directories
zip -r backup.zip Documents/ Pictures/ Downloads/
# Exclude specific files
zip -r project.zip my_project/ -x "*.log" "*.tmp"

# Exclude directories
zip -r source.zip src/ -x "src/node_modules/*" "src/.git/*"

# Exclude hidden files
zip -r clean_project.zip project/ -x "*/.*"
```

Common ZIP Options

Option	Description	Example

-r	Recursive (include subdirectories)	zip -r folder.zip folder/
-9	Maximum compression	zip -9 compressed.zip files*
-0	No compression (store only)	zip -0 fast.zip files*
-v	Verbose (show progress)	zip -rv backup.zip folder/
-q	Quiet mode	zip -rq silent.zip folder/
-u	Update existing archive	zip -u archive.zip newfile.txt
-d	Delete files from archive	zip -d archive.zip unwanted.txt
-e	Encrypt with password	zip -re secure.zip folder/
-x	Exclude files/patterns	zip -r all.zip * -x "*.tmp"

```
# Backup website files
zip -r website_backup_$(date +%Y%m%d).zip /var/www/html/

# Backup with exclusions
zip -r website_clean.zip /var/www/html/ -x "*/cache/*" "*/logs/*" "*.log"

# Source code backup (exclude build files)
zip -r myapp_source.zip myapp/ -x "*/node_modules/*" "*/build/*" "*/.git/*"

# Backup configuration files
zip -r system_configs.zip /etc/nginx/ /etc/apache2/ /etc/mysql/

# Log file archival
zip -r logs_$(date +%Y%m%d).zip /var/log/

# User home backup
zip -r user_backup.zip /home/username/ -x "*/.cache/*" "*/.tmp/*"
```

Extracting ZIP Files

```
# Extract to current directory
unzip archive.zip

# Extract to specific directory
unzip archive.zip -d /home/user/extracted/
```

```
# Extract without creating subdirectory
unzip -j archive.zip

# Extract specific files
unzip archive.zip "*.txt" "*.pdf"

# Extract specific directory
unzip archive.zip "documents/*"

# Extract with pattern
unzip archive.zip "config*"
# Overwrite existing files without prompt
unzip -o archive.zip

# Never overwrite existing files
unzip -n archive.zip

# List contents without extracting
unzip -l archive.zip

# Test archive integrity
unzip -t archive.zip

# Create split archive (100MB parts)
zip -r -s 100m large_backup.zip huge_folder/

# This creates: large_backup.z01, large_backup.z02, large_backup.zip

# Extract split archive
unzip large_backup.zip

# Zip command output
ls -la | zip -@ file_listing.zip

# Create zip from file list
cat file_list.txt | zip -@ from_list.zip
```

Shell Scripting:
A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

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

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell". Bourne shell is usually installed as /bin/sh on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.
we create a cdac.sh script. Note all the scripts would have the .sh extension
Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct.
#!/bin/sh : This tells the system that the commands that follow are to be executed by the Bourne shell.
Program 1:

Once we have our cdac.sh file created and we've specified the bash shebang on the very first line, we are ready to create our first Hello World bash script.

To do that, open the cdac.sh file again and add the following after the

```
#!/bin/bash

echo "Hello World!"
```

You will see a "Hello World" message on the screen.

Another way to run the script would be:

```
bash cdac.sh
```

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
By convention, Unix shell variables will have their names in UPPERCASE.

Defining Variables

Variables are defined as follows −

```
variable_name=variable_value
NAME="Cdac Mumbai"
num= 20
```

Variable Types

In shell scripting there are three main types of variables present. They are –

- Local Variables
- Global Variables or Environment Variables
- Shell Variables or System Variables

Local Variable

A local variable is a special type of variable which has its scope only within a specific function or block of code. Local variables can override the same variable name in the larger scope.

```
name="CDAC"
echo "Name: $name"
```

Global Variables

A global variable is a variable with global scope. It is accessible throughout the program. Global variables are declared outside any block of code or function.

Shell Variables

These are special types of variables. They are created and maintained by Linux Shell itself. These variables are required by the shell to function properly They are defined in Capital letters and to see all of them, we can use the
set / env / printenv command.

Environment Variables

These are variables that are available system-wide or to any child process initiated by the shell. Environment variables are usually defined in shell configuration files like .bashrc or .bash_profile.

```
export PATH="/usr/local/bin:$PATH"
export MY_VAR="Hello"
```

Readonly Variables

These are variables that are defined once and cannot be changed or unset.

```
readonly MY_VAR="10"
MY_VAR="20"  # This will cause an error
unset My_VAR # come out from read only mode
```

What are Shell Variables?

Shell variables are named containers that store data in the shell environment. They allow you to:

- Store values for reuse throughout scripts
- Pass data between commands and functions
- Configure environment settings for programs
- Make scripts dynamic and flexible

Types of Shell Variables

1. Local Variables (Shell-specific)

- Available only in current shell session
- Not inherited by child processes

2. Environment Variables (Global)

- Available to current shell AND child processes
- Exported to subshells and programs

3. Special Variables (Built-in)

- Predefined by the shell
- Store special information (like script arguments)

Creating and Using Variables

1. Basic Variable Assignment

```
# Create variable (NO spaces around =)
name="John Doe"
age=25
city="New York"
# Use variables with $
echo $name
echo "Hello, $name"
echo "Age: $age years"
# Better practice: use braces
echo "Hello, ${name}!"
echo "User: ${name}, Age: ${age}"
```

2. Variable Naming Rules

```
bash
# Valid variable names
username="john"
user_name="john"
userName="john"
USER123="john"
_temp="value"

# Invalid variable names
2name="john"       # Can't start with number
user-name="john"   # No hyphens
user name="john"   # No spaces
```

3. Reading Variable:

```
# Multiple ways to access variables
echo $name         # Simple form
echo ${name}       # Recommended form
echo "$name"       # With quotes (preserves spaces)
echo '$name'       # Single quotes (literal, no expansion)
```

Environment Variables

1. Creating Environment Variables

```
# Local variable (current shell only)
local_var="hello"
# Export to make it available to child processes
export global_var="world"
# Or combine in one line
export PATH_BACKUP="$PATH"
```

2. Common Environment Variables

```
# Display current environment variables
printenv
# or
env
# Important environment variables
echo $HOME        # User home directory
echo $USER        # Current username
echo $PATH        # Command search paths
echo $SHELL       # Current shell
echo $PWD         # Present working directory
echo $HOSTNAME    # Computer name
```

3. Setting Environment Variables

```
# Temporary (current session only)
export EDITOR="nano"
export JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64"
# Permanent (add to ~/.bashrc or ~/.bash_profile)
echo 'export EDITOR="nano"' >> ~/.bashrc
source ~/.bashrc    # Reload configuration
```

Sr.No.	Variable	Description
1	$0	The filename of the current script.
2	$n	These variables correspond to the arguments with which a script was invoked. $1 is the first, $2 is the second, and so on.

3	$#	The number of arguments supplied to a script.
4	$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	$?	The exit status of the last command executed.
7	$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	$!	The process number of the last background command.

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

$* and $@ are special parameters in shell scripting that represent all the arguments passed to a script or a function. However, they behave differently when used within double quotes.

1. $*: $* treats all the arguments as a single string. When referenced within double quotes, all the positional parameters (arguments) are concatenated into a single string, separated by the first character of the IFS (Internal Field Separator), which is usually a space.

- Behavior:
  - Without double quotes: $* expands to a single string containing all the arguments separated by the first character of IFS.
  - With double quotes: "$*" expands to a single string where all arguments are joined together into one string, separated by the first character of IFS.

Example:

```
#!/bin/bash
echo "Using \$*: $*"
echo "Using \"\$*\": \"$*\""
```

If the script is run with two arguments arg1 and arg2, like so: ./script.sh arg1 arg2, the output would be:

```
Using $*: arg1 arg2
Using "$*": arg1 arg2
```

2. $@: $@ treats all the arguments as separate individual strings. When referenced within double quotes, each positional parameter (argument) is treated as a separate quoted string.

- Behavior:
    - Without double quotes: $@ expands to separate strings for each argument, just like when they were passed.
    - With double quotes: "$@" expands to separate quoted strings for each argument, preserving the arguments as distinct entities.

```
#!/bin/bash
echo "Using \$@: $@"
echo "Using \"\$@\": \"$@\""
```

If the script is run with two arguments arg1 and arg2, like so: ./script.sh arg1 arg2, the output would be:

```
Using $@: arg1 arg2
Using "$@": "arg1" "arg2"
```

Key DifferencesSummary

Aspect	$*	$@
Without Double Quotes	Arguments are joined as a single string, separated by the first character of IFS.	Arguments are passed as separate strings.
With Double Quotes	All arguments are joined into a single quoted string.	Each argument is treated as a separate quoted string.

- Use "$*" when you want to treat all arguments as a single string.
- Use "$@" when you want to treat each argument as a separate string, which is usually more common and safer, especially when dealing with arguments that might contain spaces or special characters.

Exit Status

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

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

```
ls /home/vboxuser/root1
echo $?
```

Common Exit Status Codes:

- 0: Success.
- 1: General error (catchall for general errors).
- 2: Misuse of shell builtins (e.g., cd to a non-existent directory).
- 126: Command invoked cannot execute (permission issue).
- 127: Command not found (e.g., mistyping a command).
- 128+n: Fatal error signal n.
- 130: Script terminated by Ctrl+C.
- 255: Exit status out of range (exit statuses should be between 0 and 255).

Program 2:

Adding our name variable here in the file, with a welcome message. Our file now looks like this:

```
#!/bin/bash  name="CDAC"

echo "Hi there $name"
```

Program 3:

```
#!/bin/bash
echo "Hello there" $1  # $1 : first parameter
echo "Hello there" $2  # $2 : second parameter  echo "Hello there" $@  # $@ : all

#!/bin/bash

echo "1. Script name: $0"
echo "2. First argument: $1"
echo "3. Second argument: $2"
echo "4. Total arguments: $#"
echo "5. All args using \$*: $*"
echo "6. All args using \$@: $@"
echo "7. Script PID: $$"
echo "9. PID of background process: $!"
```

Bash User Input

With the previous script, we defined a variable, and we output the value of the variable on the screen with the echo $name.

```
#!/bin/bash

echo "What is your name?"
read name

echo "Hi  there  $name"
echo "Welcome to CDAC!"
```

To reduce the code, we could change the first echo statement with the read -p, the read command used with -p flag will print a message before prompting the user for their input:

```
#!/bin/bash
read -p "What is your name? " name
echo "Hi there $name"
echo "Welcome to CDAC!"
```

Slicing:

Let's review the following example of slicing in a string in Bash:

```
#!/bin/bash

letters=( "A""B""C""D""E" )
echo ${letters[@]}
```

This command will print all the elements of an array.

Output:

```
$ ABCDE
```

Program:

```
#!/bin/bash

letters=( "A""B""C""D""E" )
b=${letters:0:2}
echo "${b}”
```

Operators:
In shell scripting, arithmetic operations can be performed using various methods, such as expr, $((...))

```
#!/bin/bash

# Initialize variables
a=10
b=3

# Addition
sum=$((a + b))
echo "Addition: $a + $b = $sum"

# Subtraction
diff=$((a - b))
echo "Subtraction: $a - $b = $diff"

# Multiplication
product=$((a * b))
echo "Multiplication: $a * $b = $product"

# Division
quotient=$((a / b))
echo "Division: $a / $b = $quotient"

# Modulus
remainder=$((a % b))
echo "Modulus: $a % $b = $remainder"

# Exponentiation
power=$((a ** b))
echo "Exponentiation: $a ^ $b = $power"
```

```
# Increment
((a++))
echo "Increment: a++ = $a"

# Decrement
((b--))
echo "Decrement: b-- = $b”
```

Arithmetic Operators in Bash

Operator	Description	Example
+	Addition	result=$((a + b))
-	Subtraction	result=$((a - b))
*	Multiplication	result=$((a * b))
/	Division	result=$((a / b))
%	Modulus (remainder)	result=$((a % b))
**	Exponentiation	result=$((a ** b))
++	Increment (add 1)	((a++)) or ((++a))
--	Decrement (subtract 1)	((a--)) or ((--a))

Relational operators in shell scripting are used to compare two values or expressions. These operators return a Boolean value (true or false), which can be used in conditional statements like if, while, and until.

Common Relational Operators

Operator	Description	Example
-eq	Equal to	[$a -eq $b]
-ne	Not equal to	[$a -ne $b]
-lt	Less than	[$a -lt $b]
-le	Less than or equal to	[$a -le $b]

-gt	Greater than	[$a -gt $b]
-ge	Greater than or equal to	[$a -ge $b]

```
#!/bin/bash

# Initialize variables
a=10
b=20

# Equal to
if [ $a -eq $b ]; then
   echo "$a is equal to $b"
else
   echo "$a is not equal to $b"
fi

# Not equal to
if [ $a -ne $b ]; then
   echo "$a is not equal to $b"
else
   echo "$a is equal to $b"
fi

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

# Less than or equal to

if [ $a -le $b ]; then
   echo "$a is less than or equal to $b"
else
   echo "$a is greater than $b"
fi

# Greater than
```

```
if [ $a -gt $b ]; then
   echo "$a is greater than $b"
else
   echo "$a is not greater than $b"
fi

# Greater than or equal to
if [ $a -ge $b ]; then
   echo "$a is greater than or equal to $b"
else
   echo "$a is less than $b"
fi
```

Check if a file exists

```
#!/bin/bash
FILE="test.txt"
if [ -f "$FILE" ]; then
   echo "$FILE exists."
else
   echo "$FILE does not exist."
fi
```

Boolean operators are used to perform logical operations, typically in the context of conditional statements like if, while, and until. The common Boolean operators in Bash are &&, ||, and !.

Boolean Operators in Bash

Operator	Description	Example
&&	Logical AND: True if both conditions are true.	[$a -gt 0] && [$b -gt 0]
`		`
!	Logical NOT: Inverts the condition.	[! -f "file.txt"]

AND:

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

```
a=10
b=20

if [ $a -gt 5 ] && [ $b -gt 15 ]; then
   echo "Both conditions are true."
else
   echo "One or both conditions are false."
fi
```

OR:

```
#!/bin/bash

a=10
b=5

if [ $a -gt 15 ] || [ $b -gt 3 ]; then
   echo "At least one condition is true."
else
   echo "Both conditions are false."
fi
```

NOT:

```
#!/bin/bash

file="example.txt"

if [ ! -f "$file" ]; then
   echo "File does not exist."
else
   echo "File exists."
fi
```

String operators are used to manipulate and compare strings. Below is a list of common string operators in Bash, along with examples of how to use them.

Common String Operators in Bash

Operator	Description	Example
=	Checks if two strings are equal.	["$a" = "$b"]

!=	Checks if two strings are not equal.	["$a" != "$b"]
-z	Checks if the length of a string is zero (empty).	[-z "$a"]
-n	Checks if the length of a string is non-zero (not empty).	[-n "$a"]
>	Checks if a string is greater than another (lexicographically).	["$a" \> "$b"]
<	Checks if a string is less than another (lexicographically).	["$a" \< "$b"]

Examples of String Operators

1. Equality (=)

The = operator is used to check if two strings are equal.

```
#!/bin/bash

a="hello"
b="hello"

if [ "$a" = "$b" ]; then
   echo "Strings are equal."
else
   echo "Strings are not equal."
fi
```

Inequality (!=)

The != operator is used to check if two strings are not equal.

```
#!/bin/bash

a="hello"
b="world"

if [ "$a" != "$b" ]; then
   echo "Strings are not equal."
else
```

```
    echo "Strings are equal."
fi
```

Empty String Check (-z)

The -z operator checks if the length of a string is zero (i.e., the string is empty).

```
#!/bin/bash

a=""

if [ -z "$a" ]; then
    echo "String is empty."
else
    echo "String is not empty."
fi
```

Non-Empty String Check (-n)

The -n operator checks if the length of a string is non-zero (i.e., the string is not empty).

```
#!/bin/bash

a="hello"

if [ -n "$a" ]; then
    echo "String is not empty."
else
    echo "String is empty."
fi
```

File test operators in shell scripting are used to check properties of files and directories, such as whether a file exists, whether it is a regular file or a directory, whether it has specific permissions, and more. These operators are crucial for writing scripts that need to interact with the filesystem.

Common File Test Operators in Bash

Operator	Description	Example
-e	Checks if a file (or directory) exists.	[-e file.txt]

-f	Checks if a file exists and is a regular file (not a directory or device).	[-f file.txt]
-d	Checks if a directory exists.	[-d /path/to/dir]
-r	Checks if a file is readable.	[-r file.txt]
-w	Checks if a file is writable.	[-w file.txt]
-x	Checks if a file is executable.	[-x script.sh]
-s	Checks if a file exists and is not empty (size is greater than zero).	[-s file.txt]
-L	Checks if a file is a symbolic link.	[-L symlink]
-c	Checks if a file is a character special file (device file).	[-c /dev/ttyS0]
-b	Checks if a file is a block special file (device file).	[-b /dev/sda]
-p	Checks if a file is a named pipe (FIFO).	[-p /path/to/fifo]
-S	Checks if a file is a socket.	[-S /path/to/socket]
-u	Checks if a file has the setuid bit set.	[-u file.txt]
-g	Checks if a file has the setgid bit set.	[-g file.txt]
-k	Checks if a file has the sticky bit set.	[-k /path/to/dir]

Check if a File Exists (-e)

The -e operator is used to check if a file or directory exists.

```
#!/bin/bash

if [ -e file.txt ]; then
   echo "file.txt exists."
else
   echo "file.txt does not exist."
fi
```

Check if a Regular File Exists (-f)

```
#!/bin/bash

if [ -f file.txt ]; then
    echo "file.txt is a regular file."
else
    echo "file.txt is not a regular file or does not exist."
fi
```

Check if a Directory Exists (-d)

```
#!/bin/bash

if [ -d /path/to/dir ]; then
    echo "Directory exists."
else
    echo "Directory does not exist."
fi
```

Check if a File is Readable (-r)

```
#!/bin/bash

if [ -r file.txt ]; then
    echo "file.txt is readable."
else
    echo "file.txt is not readable."
fi
```

Check if a File is Not Empty (-s)

```
#!/bin/bash

if [ -s file.txt ]; then
    echo "file.txt is not empty."
else
    echo "file.txt is empty or does not exist."
fi
```

Bash Conditional Expressions

In Bash, conditional expressions are used by the [[compound command and the [built-in commands to test file attributes and perform string and arithmetic comparisons.

To match True condition: [[${string1} == ${string2}]]

True if the strings are not equal.: [[${string1} != ${string2}]]

Arithmetic operators

Returns true if the numbers are equal : [[${arg1} -eq ${arg2}]]

Returns true if the numbers are not equal : [[${arg1} -ne ${arg2}]]

Returns true if arg1 is less than arg2: [[${arg1} -lt ${arg2}]]

Returns true if arg1 is less than or equal arg2: [[${arg1} -le ${arg2}]]

Returns true if arg1 is greater than arg2: [[${arg1} -gt ${arg2}]]

Returns true if arg1 is greater than or equal arg2: [[${arg1} -ge ${arg2}]]

AND: [[test_case_1]] && [[test_case_2]]
OR: [[test_case_1]] || [[test_case_2]]

```
#!/bin/bash
p_pass="CDAC"

read -p "Enter your username? " password
# Check if the username provided is the admin
if [[ "${p_pass}" == "${password}" ]] ; then
echo "You have entered the correct password!"
else
echo "You have entered the wrong password!"
fi
```

Program:

```
#!/bin/bash
read -p "Enter a number: " num  if [[ $num -gt 0]] ; then
echo "The number is positive"  elif [[ $num -lt 0]] ; then
echo "The number is negative"
else
echo "The number is 0"
fi
```

Example:

```
#!/bin/bash

# Prompt the user to enter a number
echo "Enter a number between 1 and 3:"
read number

# Check the value of the number using if, elif, else
if [ "$number" -eq 1 ]; then
   echo "You entered one."
elif [ "$number" -eq 2 ]; then
   echo "You entered two."
elif [ "$number" -eq 3 ]; then
   echo "You entered three."
else
   echo "You entered a number outside the range of 1 to 3."
fi
```

Checking if a Number is Positive, Negative, or Zero

```
#!/bin/bash

# Prompt the user to enter a number
echo "Enter a number:"
read number

# Check if the number is positive, negative, or zero
if [ "$number" -gt 0 ]; then
   echo "The number is positive."
elif [ "$number" -lt 0 ]; then
   echo "The number is negative."
```

```
else
   echo "The number is zero."
fi
```

Checking if a Character is a Vowel or a Consonant

```
#!/bin/bash
# Prompt the user to enter a single character

echo "Enter a single letter:"
read char
# Check if the character is a vowel or consonant
if [ "$char" == "a" ] || [ "$char" == "e" ] || [ "$char" == "i" ] || [ "$char" == "o" ] || [ "$char" ==
"u" ]; then
   echo "$char is a vowel."
elif [[ "$char" =~ [a-zA-Z] ]]; then
   echo "$char is a consonant."
else
   echo "$char is not a valid letter."
fi
```

Switch case statements

As in other programming languages, you can use a case statement to simplify complex conditionals when there are multiple different choices. So rather than using a few if, and if-else statements, you could use a single case statement.

The Bash case statement syntax looks like this:

```
case $some_variable in  pattern_1)
commands
;;

pattern_2| pattern_3)  commands
;;

*)
default commands
;;
esac
```

A quick rundown of the structure:

- All case statements start with the case keyword.
- On the same line as the case keyword, you need to specify a variable or an expression followed by the in keyword.
- After that, you have your case patterns, where you need to use) to identify the end of the pattern.
- You can specify multiple patterns divided by a pipe: |.
- After the pattern, you specify the commands that you would like to be executed in case that the pattern matches the variable or the expression that you've specified.
- All clauses have to be terminated by adding ;; at the end.   You can have a default statement by adding a * as the pattern.
- To close the case statement, use the esac (case typed backwards) keyword.

Here is an example of a Bash case statement:

```
#!/bin/bash
read -p "Enter the name of your car brand: " car  case $car in

Tesla)
echo -n "${car}'s car factory is in the USA."
;;

BMW | Mercedes | Audi | Porsche)
echo -n "${car}'s car factory is in Germany."
;;

Toyota | Mazda | Mitsubishi | Subaru)
echo -n "${car}'s car factory is in Japan."
;;

*)
echo -n "${car} is an unknown car brand"
;;

esac
```

Days of the Week

```
#!/bin/bash

# Prompt the user to enter a day of the week
echo "Enter a day of the week (e.g., Monday, Tuesday, etc.):"
read day

# Use a case statement to match the input to a specific day
case $day in
    monday)
        echo "Today is Monday. Start of the workweek!"
        ;;
    tuesday)
        echo "Today is Tuesday. Second day of the workweek!"
        ;;
    wednesday)
        echo "Today is Wednesday. Midweek already!"
        ;;
    thursday)
        echo "Today is Thursday. Almost the weekend!"
        ;;
    friday)
        echo "Today is Friday. The weekend is near!"
        ;;
    saturday)
        echo "Today is Saturday. Enjoy your weekend!"
        ;;
    sunday)
        echo "Today is Sunday. Rest well for the week ahead!"
        ;;
    *)
        echo "That's not a valid day of the week."
        ;;
esac
```

Basic Calculator

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

```
# Prompt the user to enter two numbers
echo "Enter the first number:"
read num1
echo "Enter the second number:"
read num2

# Prompt the user to choose an operation
echo "Choose an operation (+, -, *, /):"
read operation

# Use a case statement to perform the chosen operation
case $operation in
  +)
    result=$((num1 + num2))
    echo "The result of $num1 + $num2 is: $result"
    ;;
  -)
    result=$((num1 - num2))
    echo "The result of $num1 - $num2 is: $result"
    ;;
  \*)
    result=$((num1 * num2))
    echo "The result of $num1 * $num2 is: $result"
    ;;
  /)
    if [ $num2 -ne 0 ]; then
      result=$((num1 / num2))
      echo "The result of $num1 / $num2 is: $result"
    else
      echo "Error: Division by zero is not allowed."
    fi
    ;;
  *)
    echo "Invalid operation. Please choose +, -, *, or /."
    ;;
esac
```

Menu-Driven Program

```
#!/bin/bash

while true; do
  # Display the menu
  echo "Please choose an option:"
  echo "1. Display the current date and time"
  echo "2. List files in the current directory"
  echo "3. Show the current user"
  echo "4. Exit"

  # Read the user's choice
  read -p "Enter your choice [1-4]: " choice

  # Use a case statement to perform the selected action
  case $choice in
    1)
      echo "Current date and time: $(date)"
      ;;
    2)
      echo "Files in the current directory:"
      ls
      ;;
    3)
      echo "Current user: $USER"
      ;;
    4)
      echo "Exiting the program. Goodbye!"
      break
      ;;
    *)
      echo "Invalid option. Please choose a number between 1 and 4."
      ;;
  esac

  echo "__________________________"
done
```

For loops

Here is the structure of a for loop:

```
for var in ${list}  do
your_commands
done
```

Program:

```
#!/bin/bash  users="CDAC Kharghar "
 for user in ${users}
do
echo "${user}"
done
```

```
#!/bin/bash

for num in {1..10}
do
echo ${num}
done
```

Iterating Over a List of Items

```
#!/bin/bash

# Define an array of team members
list_team_member=("Anand" "Nandni" "Chandni" "Nidhi")

# Print total number of team members
echo "Total team members: ${#list_team_member[@]}"

# Print all names
echo "Team members are:"
for team_member in "${list_team_member[@]}"
do
  echo "- $team_member "
```

```
done
```

Iterating Over Files in a Directory

```
#!/bin/bash

# Use a for loop to list all files in the current directory
for file in *; do
  echo "Found file: $file"
done
```

Calculating the Sum of a List of Numbers

```
#!/bin/bash

numbers=(10 20 30 40 50)

sum=0
for number in "${numbers[@]}"; do
  sum=$((sum + number))
done

echo "The total sum is: $sum"
```

While loops

The structure of a while loop is quite similar to the for loop:

```
while [ your_condition ]
do
your_commands
done
```

Here is an example of a while loop:

```
#!/bin/bash
counter=1
```

```
while [[ $counter -le 10 ]]
do
echo $counter  ((counter++))
done
```

shell Input/Output Redirections

Input and output redirection are fundamental concepts in shell scripting that allow you to control the flow of data to and from commands. Here’s an overview of how input/output redirections work in the shell, along with examples.

1. Output Redirection

Output redirection allows you to direct the output of a command to a file or another command instead of the standard output (which is usually the terminal).

Redirecting Standard Output (> and >>)

>: Redirects the output to a file. If the file exists, it will be overwritten. If the file does not exist, it will be created.
Example:

```
echo "Hello, World!" > output.txt
```

This command writes "Hello, World!" to output.txt. If output.txt exists, it will be overwritten.
>>: Appends the output to a file. If the file does not exist, it will be created.
Example:

```
echo "Hello again!" >> output.txt
```

This command appends "Hello again!" to output.txt without overwriting the existing content.

2. Input Redirection

Input redirection allows you to take input for a command from a file instead of the standard input (which is usually the keyboard).

Redirecting Standard Input (<)

<: Redirects the input from a file.
Example:

```
cat < input.txt
```

This command reads the contents of input.txt and displays them using cat.

3. Redirecting Standard Error

Standard error redirection is used to direct error messages to a file or another location.

Redirecting Standard Error (2> and 2>>)

2>: Redirects standard error to a file. If the file exists, it will be overwritten.
Example:

```
ls /nonexistent 2> error.log
```

This command tries to list a non-existent directory, and the error message is redirected to error.log.
2>>: Appends standard error to a file. If the file does not exist, it will be created.
Example:

```
ls /anothernonexistent 2>> error.log
```

This command appends the error message to error.log without overwriting the existing content.

4. Redirecting Both Standard Output and Error

Sometimes you may want to redirect both standard output and error to the same file.

Redirecting Both Standard Output and Error (&> and 2>&1)

&>: Redirects both standard output and standard error to a file.
Example:

```
ls /nonexistent &> output_and_error.log
```

This command redirects both the standard output and error to output_and_error.log.
2>&1: Redirects standard error to the same place as standard output.
Example:

```
ls /nonexistent > output_and_error.log 2>&1
```

This command first redirects standard output to output_and_error.log, then redirects standard error to the same file.

5. Here Document (<<)

A here document allows you to redirect multiple lines of input to a command.
Example:

```
cat << EOF
This is line 1.
This is line 2.
```

```
EOF
```

This command sends the lines between << EOF and EOF as input to the cat command, which then outputs them.

6. Here Strings (<<<)

A here string redirects a single string as input to a command.
Example:

```
grep "hello" <<< "hello world"
```

This command searches for the string "hello" in the provided string "hello world".

Function:
A function is a block of code that you can define once and call multiple times throughout your script. Functions help you to organize and reuse code, making your scripts more modular and easier to maintain.

```
#!/bin/bash

# Define a function named greet
greet() {
   echo "Hello, $1!"
}

# Call the function greet with an argument
greet "CDAC"
greet "MUMBAI"
```

```
#!/bin/bash

# Define a function that adds two numbers
add_numbers() {
   result=$(( $1 + $2 ))
   echo "The sum of $1 and $2 is: $result"
}

# Call the function with two arguments
```

```
add_numbers 5 10



#!/bin/bash

# Define a function that returns the square of a number
square() {
   echo $(( $1 * $1 ))
}

# Capture the return value in a variable
result=$(square 4)

# Print the result
echo "The square of 4 is: $result”



# Define a function that checks if a number is positive, negative, or zero
#!/bin/bash
check_number() {
   if [ $1 -gt 0 ]; then
      echo "$1 is positive."
   elif [ $1 -lt 0 ]; then
      echo "$1 is negative."
   else
      echo "$1 is zero."
   fi
}

# Call the function with different arguments
check_number 10
check_number -5
check_number 0
```

Basic Hello World

```
#!/bin/bash
# hello.sh - My first shell script
echo "Hello, World!"
echo "Welcome to Shell Scripting!"
```

Interactive Hello

```
#!/bin/bash
# interactive_hello.sh
echo "What's your name?"
read name
echo "Hello, $name! Welcome to Shell Scripting!"
```

Hello with Date

```
#!/bin/bash
# hello_date.sh
echo "========================================="
echo "Hello World Script"
echo "Current Date: $(date)"
echo "Script run by: $(whoami)"
echo "========================================="
```

How to run:

```
chmod +x hello.sh
./hello.sh
```

Well-Documented Script

```
#!/bin/bash
#
# Main script starts here
echo "System Information Report"
echo "========================="

# Show hostname
echo "Hostname: $(hostname)"

# Show current user
echo "Current User: $(whoami)"

# Show current directory
echo "Current Directory: $(pwd)"
```

Script with Functions and Comments

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

```
# Function to display usage
show_usage() {
    echo "Usage: $0 filename"
    echo "Checks if the specified file exists"
}

# Main logic
if [ $# -eq 0 ]; then
    show_usage
    exit 1
fi

# Check if file exists
if [ -f "$1" ]; then
    echo "File '$1' exists and is a regular file"
else
    echo "File '$1' does not exist or is not a regular file"
fi
```

Basic Variables

```
#!/bin/bash
# variables_basic.sh

# Creating variables
name="John Doe"
age=25
city="New York"

# Using variables
echo "Name: $name"
echo "Age: $age"
echo "City: $city"

# Variable operations
full_info="$name is $age years old and lives in $city"
echo "$full_info"

# Variable modification
age=$((age + 1))
echo "Next year, $name will be $age years old"
```

Local vs Global Variables

```
#!/bin/bash
# local_global.sh

# Global variable
global_var="I am global"

function demo_local() {
   local local_var="I am local"
   local global_var="I am local override"

   echo "Inside function:"
   echo "  Local var: $local_var"
   echo "  Global var: $global_var"
}

function demo_global() {
   echo "In another function:"
   echo "  Global var: $global_var"
   # echo "  Local var: $local_var"  # This would fail
}

echo "Before function call: $global_var"
demo_local
demo_global
echo "After function call: $global_var"
```

Variable Assignment Methods

```
#!/bin/bash
# variable_methods.sh

# Direct assignment
direct_var="Direct Value"

# Command substitution
current_date=$(date)
user_count=$(who | wc -l)

# Arithmetic assignment
```

```
number=5
result=$((number * 2 + 3))

# Read from user
echo "Enter your favorite color:"
read favorite_color

# Default value assignment
default_name=${1:-"Anonymous"}

# Display all variables
echo "Direct: $direct_var"
echo "Date: $current_date"
echo "Users logged in: $user_count"
echo "Calculation result: $result"
echo "Favorite color: $favorite_color"
echo "Name: $default_name"
```

Command Line Arguments

```
#!/bin/bash
# arguments.sh

echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Third argument: $3"
echo "Number of arguments: $#"
echo "All arguments (\$*): $*"
echo "All arguments (\$@): $@"
echo "Process ID: $$"
echo "Last command exit status: $?"

# Test with different numbers of arguments
if [ $# -eq 0 ]; then
   echo "No arguments provided"
elif [ $# -eq 1 ]; then
   echo "One argument provided: $1"
else
   echo "Multiple arguments provided"
fi
```

Basic Environment Variables

```
#!/bin/bash
# environment.sh

echo "=== System Environment Variables ==="
echo "Home Directory: $HOME"
echo "Current User: $USER"
echo "Shell: $SHELL"
echo "Path: $PATH"
echo "Working Directory: $PWD"
echo "Language: $LANG"

echo -e "\n=== Custom Environment Variables ==="

# Create and export custom variable
export MY_APP_VERSION="1.0.0"
export MY_APP_NAME="Shell Tutorial"

echo "App Name: $MY_APP_NAME"
echo "App Version: $MY_APP_VERSION"

# Show all environment variables
echo -e "\n=== All Environment Variables ==="
env | head -10
```

Reading User Input

```
#!/bin/bash

# Simple read
echo "What's your name?"
read name
echo "Hello, $name!"

# Read with prompt
read -p "Enter your age: " age

# Read password (hidden input)
read -s -p "Enter password: " password
echo  # New line after hidden input
```

```
# Read with timeout
echo "Quick! Enter something within 5 seconds:"
if read -t 5 response; then
   echo "You entered: $response"
else
   echo "Timeout! No input received."
fi

# Read multiple values
echo "Enter first name and last name:"
read first_name last_name
echo "Full name: $first_name $last_name"

# Validate input
while true; do
   read -p "Enter a number (1-10): " number
   if [[ "$number" =~ ^[1-9]|10$ ]]; then
      echo "Valid number: $number"
      break
   else
      echo "Invalid! Please enter a number between 1 and 10."
   fi
done
```

Command Substitution

```
#!/bin/bash
echo "=== Command Substitution Examples ==="

# Modern syntax $()
current_date=$(date)
file_count=$(ls | wc -l)
disk_usage=$(df -h / | tail -1 | awk '{print $5}')

echo "Current date: $current_date"
echo "Files in current directory: $file_count"
echo "Root disk usage: $disk_usage"

# Nested command substitution
echo "Files modified today: $(find . -type f -mtime -1 | wc -l)"
```

```
# Command substitution in variables
log_file="app_$(date +%Y%m%d).log"
echo "Today's log file would be: $log_file"

# Arithmetic with command substitution
total_size=$(du -sb . | cut -f1)
size_mb=$((total_size / 1024 / 1024))
echo "Current directory size: ${size_mb}MB"

# Multiple commands
system_info="Host: $(hostname), User: $(whoami), Shell: $SHELL"
echo "System Info: $system_info"
```

 Reading from Files

```
#!/bin/bash
# file_input.sh

# Create sample file
cat > sample.txt << EOF
Line 1: Apple
Line 2: Banana
Line 3: Cherry
Line 4: Date
Line 5: Elderberry
EOF

echo "=== Reading entire file ==="
cat sample.txt

echo -e "\n=== Reading file line by line ==="
while read -r line; do
   echo "Processing: $line"
done < sample.txt

echo -e "\n=== Reading specific lines ==="
# Read first 3 lines
head -n 3 sample.txt

echo -e "\n=== Reading with line numbers ==="
```

```
cat -n sample.txt

# Reading into array
echo -e "\n=== Reading into array ==="
readarray -t lines < sample.txt
for i in "${!lines[@]}"; do
   echo "Array[$i]: ${lines[$i]}"
done

# Cleanup
rm -f sample.txt
```

Basic If-Then-Else

```
#!/bin/bash
echo "Enter a number:"
read number

if [ "$number" -gt 10 ]; then
   echo "$number is greater than 10"
elif [ "$number" -eq 10 ]; then
   echo "$number is exactly 10"
else
   echo "$number is less than 10"
fi

# File existence check
filename="test.txt"
if [ -f "$filename" ]; then
   echo "File $filename exists"
else
   echo "File $filename does not exist"
   echo "Creating file..."
   touch "$filename"
fi
```

Multiple Conditions

```
#!/bin/bash
# multiple_conditions.sh

read -p "Enter your age: " age
read -p "Do you have a license? (y/n): " license

# Multiple conditions with AND
if [ "$age" -ge 18 ] && [ "$license" = "y" ]; then
   echo "You can drive!"
elif [ "$age" -ge 18 ] && [ "$license" = "n" ]; then
   echo "You're old enough but need a license"
elif [ "$age" -lt 18 ] && [ "$license" = "y" ]; then
   echo "You have a license but are too young"
else
   echo "You cannot drive (too young and no license)"
fi

# OR condition example
read -p "Enter day of week: " day
if [ "$day" = "Saturday" ] || [ "$day" = "Sunday" ]; then
   echo "It's weekend!"
else
   echo "It's a weekday"
fi
```

Nested Conditions

```
#!/bin/bash
# nested_conditions.sh

read -p "Enter username: " username
read -s -p "Enter password: " password
echo

# Check user authentication
if [ -n "$username" ]; then
   if [ "$username" = "admin" ]; then
      if [ "$password" = "secret123" ]; then
         echo "Welcome, Administrator!"

         # Admin menu
         echo "What would you like to do?"
         echo "1. View logs"
         echo "2. Manage users"
         echo "3. System status"
```

```
        read -p "Choose option (1-3): " choice

        if [ "$choice" = "1" ]; then
            echo "Displaying logs..."
        elif [ "$choice" = "2" ]; then
            echo "Managing users..."
        elif [ "$choice" = "3" ]; then
            echo "System status: OK"
        else
            echo "Invalid option"
        fi
    else
        echo "Invalid password for admin"
    fi
  else
    echo "Regular user: $username"
    if [ "$password" = "user123" ]; then
        echo "Welcome, $username!"
    else
        echo "Invalid password"
    fi
  fi
else
  echo "Username cannot be empty"
fi
```

Basic Case Statement

```
#!/bin/bash
# case_basic.sh

echo "=== Simple Menu System ==="
echo "1. View files"
echo "2. View processes"
echo "3. View disk usage"
echo "4. Exit"

read -p "Choose option (1-4): " choice

case $choice in
  1)
```

```
        echo "Files in current directory:"
        ls -la
        ;;
    2)
        echo "Current processes:"
        ps aux | head -10
        ;;
    3)
        echo "Disk usage:"
        df -h
        ;;
    4)
        echo "Goodbye!"
        exit 0
        ;;
    *)
        echo "Invalid option: $choice"
        echo "Please choose 1, 2, 3, or 4"
        ;;
esac
```

Example 2: Pattern Matching

```
#!/bin/bash
# pattern_matching.sh

read -p "Enter a filename: " filename

echo "File analysis for: $filename"

case "$filename" in
    *.txt)
        echo "Text file detected"
        [ -f "$filename" ] && wc -l "$filename"
        ;;
    *.sh)
        echo "Shell script detected"
        [ -f "$filename" ] && echo "Lines of code: $(grep -v '^#\|^$' "$filename" | wc -l)"
        ;;
    *.jpg|*.jpeg|*.png|*.gif)
        echo "Image file detected"
        [ -f "$filename" ] && file "$filename"
        ;;
    *.log)
        echo "Log file detected"
        [ -f "$filename" ] && echo "Last 5 lines:" && tail -5 "$filename"
        ;;
    [0-9]*)
        echo "Filename starts with a number"
        ;;
    [A-Z]*)
        echo "Filename starts with uppercase letter"
        ;;
    [a-z]*)
        echo "Filename starts with lowercase letter"
        ;;
    *)
        echo "Unknown file type"
        ;;
esac

#!/bin/bash
# advanced_case.sh

# System service manager
read -p "Enter service command: " command

case "$command" in
    start|START|Start)
        echo "Starting service..."
        echo "Service started successfully"
        ;;
    stop|STOP|Stop)
        echo "Stopping service..."
        echo "Service stopped"
        ;;
    restart|RESTART|Restart)
        echo "Restarting service..."
        echo "Service restarted"
        ;;
    status|STATUS|Status)
```

```
        echo "Service status: Running"
        echo "PID: $$"
        echo "Uptime: $(uptime)"
        ;;
    reload|RELOAD)
        echo "Reloading configuration..."
        ;;
    enable)
        echo "Service enabled for auto-start"
        ;;
    disable)
        echo "Service disabled"
        ;;
    help|--help|-h)
        echo "Available commands:"
        echo "  start, stop, restart, status, reload, enable, disable"
        ;;
    quit|exit|q)
        echo "Exiting service manager"
        exit 0
        ;;
    "")
        echo "No command entered"
        ;;
    *)
        echo "Unknown command: $command"
        echo "Type 'help' for available commands"
        ;;
esac
```

Basic For Loops

```
#!/bin/bash
# for_basic.sh

echo "=== Basic For Loop ==="
for i in 1 2 3 4 5; do
    echo "Number: $i"
done

echo -e "\n=== For Loop with Range ==="
```

```
for i in {1..5}; do
   echo "Count: $i"
done

echo -e "\n=== For Loop with Files ==="
for file in *.txt; do
   if [ -f "$file" ]; then
      echo "Processing file: $file"
      wc -l "$file"
   fi
done

echo -e "\n=== For Loop with Command Output ==="
for user in $(cut -d: -f1 /etc/passwd | head -5); do
   echo "User: $user"
done

echo -e "\n=== For Loop with Array ==="
fruits=("apple" "banana" "cherry" "date")
for fruit in "${fruits[@]}"; do
   echo "Fruit: $fruit"
done

#!/bin/bash
# for_c_style.sh

echo "=== C-Style For Loop ==="
for ((i=1; i<=10; i++)); do
   echo "Iteration $i"
done

echo -e "\n=== Countdown Timer ==="
for ((i=10; i>=1; i--)); do
   echo "Countdown: $i"
   sleep 1
done
echo "Blast off!"

echo -e "\n=== Multiplication Table ==="
read -p "Enter number for multiplication table: " number
```

```
for ((i=1; i<=12; i++)); do
   result=$((number * i))
   printf "%d x %d = %d\n" $number $i $result
done

echo -e "\n=== Processing Array with Index ==="
colors=("red" "green" "blue" "yellow" "purple")
for ((i=0; i<${#colors[@]}; i++)); do
   printf "Color[%d] = %s\n" $i "${colors[$i]}"
done

#!/bin/bash
# for_advanced.sh

echo "=== Processing Multiple Directories ==="
for dir in /etc /var/log /home; do
   if [ -d "$dir" ]; then
      echo "Directory: $dir"
      echo "  Files: $(find "$dir" -maxdepth 1 -type f 2>/dev/null | wc -l)"
      echo "  Subdirs: $(find "$dir" -maxdepth 1 -type d 2>/dev/null | wc -l)"
      echo "  Size: $(du -sh "$dir" 2>/dev/null | cut -f1)"
      echo "---"
   fi
done

echo -e "\n=== File Processing with Patterns ==="
# Create some test files
touch file1.txt file2.log file3.conf test.backup

for file in file*.{txt,log,conf}; do
   [ -f "$file" ] || continue  # Skip if file doesn't exist

   echo "Processing: $file"
   case "$file" in
      *.txt)  echo "  Type: Text file" ;;
      *.log)  echo "  Type: Log file" ;;
      *.conf) echo "  Type: Config file" ;;
   esac

   echo "  Size: $(stat -f%z "$file" 2>/dev/null || stat -c%s "$file" 2>/dev/null) bytes"
```

```
done

# Cleanup
rm -f file1.txt file2.log file3.conf test.backup

echo -e "\n=== Nested For Loops ==="
echo "Multiplication table (1-5):"
for ((i=1; i<=5; i++)); do
    for ((j=1; j<=5; j++)); do
        printf "%3d" $((i * j))
    done
    echo
done

#!/bin/bash
# while_basic.sh

echo "=== Basic While Loop ==="
counter=1
while [ $counter -le 5 ]; do
    echo "Counter: $counter"
    counter=$((counter + 1))
done

echo -e "\n=== User Input Loop ==="
while true; do
    read -p "Enter 'quit' to exit: " input

    if [ "$input" = "quit" ]; then
        echo "Exiting..."
        break
    else
        echo "You entered: $input"
    fi
done

echo -e "\n=== Number Guessing Game ==="
secret=7
guess=0
```

```
while [ $guess -ne $secret ]; do
   read -p "Guess the number (1-10): " guess

   if [ $guess -lt $secret ]; then
      echo "Too low!"
   elif [ $guess -gt $secret ]; then
      echo "Too high!"
   else
      echo "Correct! The number was $secret"
   fi
done

#!/bin/bash
# arrays_basic.sh

echo "=== Basic Array Operations ==="

# Create arrays
fruits=("apple" "banana" "cherry")
numbers=(1 2 3 4 5)

# Another way to create arrays
colors[0]="red"
colors[1]="green"
colors[2]="blue"

# Display array elements
echo "Fruits array:"
for i in "${!fruits[@]}"; do
   echo "  fruits[$i] = ${fruits[$i]}"
done

echo -e "\nAll fruits: ${fruits[*]}"
echo "Number of fruits: ${#fruits[@]}"

# Adding elements
fruits+=("date" "elderberry")
echo "After adding: ${fruits[*]}"

# Accessing specific elements
```

```
echo "First fruit: ${fruits[0]}"
echo "Last fruit: ${fruits[-1]}"  # Bash 4.3+
echo "Second to last: ${fruits[-2]}"

# Slicing arrays
echo "First 3 fruits: ${fruits[*]:0:3}"
echo "From index 2: ${fruits[*]:2}"
```

```
#!/bin/bash

advanced_case_examples() {
    echo "=== Advanced Case Statement Examples ==="

    # Multi-level menu system
    main_menu() {
        while true; do
            echo -e "\n=== Main Menu ==="
            echo "1. User Management"
            echo "2. System Tools"
            echo "3. File Operations"
            echo "4. Network Tools"
            echo "5. Exit"

            read -p "Choose option (1-5): " choice

            case "$choice" in
                1) user_management_menu ;;
                2) system_tools_menu ;;
                3) file_operations_menu ;;
                4) network_tools_menu ;;
                5) echo "Goodbye!"; break ;;
                *) echo " Invalid option. Please choose 1-5." ;;
            esac
        done
    }

    # Submenu functions
    user_management_menu() {
```

```
    echo -e "\n--- User Management ---"
    echo "a. List users"
    echo "b. Add user"
    echo "c. Delete user"
    echo "d. Back to main menu"

    read -p "Choose option (a-d): " sub_choice

    case "$sub_choice" in
        a|A) echo "Listing users..."; cut -d: -f1 /etc/passwd | tail -5 ;;
        b|B) echo "Add user functionality..." ;;
        c|C) echo "Delete user functionality..." ;;
        d|D) return ;;
        *) echo "Invalid option" ;;
    esac
}

system_tools_menu() {
    echo -e "\n--- System Tools ---"
    read -p "Choose: (s)ystem info, (p)rocesses, (d)isk usage, (b)ack: " sub_choice

    case "$sub_choice" in
        s|S|system) uname -a ;;
        p|P|processes) ps aux | head -10 ;;
        d|D|disk) df -h ;;
        b|B|back) return ;;
        *) echo "Unknown option: $sub_choice" ;;
    esac
}

file_operations_menu() {
    echo -e "\n--- File Operations ---"
    echo "Enter file operation:"
    read -p "(l)ist, (c)reate, (d)elete, (b)ack: " sub_choice

    case "$sub_choice" in
        l|list) ls -la ;;
        c|create)
            read -p "Enter filename: " filename
            touch "$filename" && echo "Created: $filename"
            ;;
        d|delete)
            read -p "Enter filename to delete: " filename
            [[ -f "$filename" ]] && rm "$filename" && echo "Deleted: $filename"
            ;;
        b|back) return ;;
        *) echo "Invalid operation" ;;
    esac
  }

  network_tools_menu() {
    echo -e "\n--- Network Tools ---"
    read -p "Enter tool: (p)ing, (n)etstat, (i)fconfig, (b)ack: " sub_choice

    case "$sub_choice" in
        p|ping)
            read -p "Enter host to ping: " host
            ping -c 3 "$host" 2>/dev/null || echo "Ping failed"
            ;;
        n|netstat)
            command -v netstat >/dev/null && netstat -tulnp | head -10 || echo "netstat not
available"
            ;;
        i|ifconfig)
            ip addr show 2>/dev/null || ifconfig 2>/dev/null || echo "Network tools not available"
            ;;
        b|back) return ;;
        *) echo "Invalid tool" ;;
    esac
  }

  # Start the menu system
  main_menu
}

# Pattern matching with case
pattern_matching_case() {
  echo -e "\n=== Pattern Matching with Case ==="

  file_type_analyzer() {
    local filename="$1"

    case "$filename" in
        # Multiple patterns
        *.jpg|*.jpeg|*.png|*.gif|*.bmp)
            echo "Image file: $filename"
            ;;
        # Character classes
        *.[Tt][Xx][Tt])
            echo " Text file: $filename"
            ;;
        # Wildcards
        [Rr][Ee][Aa][Dd][Mm][Ee]*)
            echo "Documentation: $filename"
            ;;
        # Number patterns
        *[0-9][0-9][0-9][0-9].log)
            echo "Yearly log file: $filename"
            ;;
        # Complex patterns
        backup_[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]_*.tar.gz)
            echo "Dated backup file: $filename"
            ;;
        # Default
        *)
            echo " Unknown file type: $filename"
            ;;
    esac
}
```

RIPGREP (rg) - Modern Alternative to grep

RipGrep is a line-oriented search tool that recursively searches directories for a regex pattern. It's designed to be faster than traditional grep and includes many modern features like automatic gitignore support, Unicode handling, and smart case sensitivity.
Key Features:

- Fast parallel searching across multiple files
- Automatic file type detection and filtering
- Respects .gitignore and other ignore files

- Smart case sensitivity (case-insensitive if all lowercase)
- Better default behavior than traditional grep

Installation

```
# Ubuntu/Debian
sudo apt install ripgrep

# Alternative names
rg --version
```

Basic Syntax

```
rg [OPTIONS] PATTERN [PATH...]
rg "search_term"                # Search in current directory
rg "pattern" /path/to/search        # Search in specific path
```

Essential Options

Option	Description	Example
-i	Case-insensitive search	rg -i "error"
-w	Match whole words only	rg -w "test"
-v	Invert match (exclude)	rg -v "debug"
-n	Show line numbers	rg -n "pattern"
-A NUM	Show NUM lines after	rg -A 3 "error"
-B NUM	Show NUM lines before	rg -B 2 "error"
-C NUM	Show NUM lines context	rg -C 2 "error"

Advanced Options

Option	Description	Example
--type TYPE	Search only specific file types	rg --type py "function"
--type-not TYPE	Exclude file types	rg --type-not log "error"

-g GLOB	Include files matching glob	rg -g "*.conf" "setting"
--hidden	Search hidden files	rg --hidden "pattern"
--no-ignore	Don't respect ignore files	rg --no-ignore "pattern"
-r REPLACEMENT	Replace matches	rg "old" -r "new"

```
# Basic searching
rg "function"                    # Find all occurrences of "function"
rg -i "error"                    # Case-insensitive search for "error"
rg -w "test"                     # Match whole word "test" only

# File type filtering
rg --type py "import"            # Search only Python files
rg --type-not log "exception"    # Exclude log files
rg -g "*.conf" "port"            # Search only .conf files

# Context and formatting
rg -n "TODO"                     # Show line numbers
rg -A 3 -B 2 "error"             # 3 lines after, 2 before
rg --color=always "pattern" | less -R # Preserve colors in pager

# Advanced patterns
rg "\b\d{3}-\d{2}-\d{4}\b"          # Social Security Number pattern
rg "function\s+\w+\(" --type py     # Function definitions in Python
rg "https?://[^\s]+" --type html    # URLs in HTML files

# Replacement (preview only)
rg "old_function" -r "new_function" --dry-run
# Find configuration issues
rg -i "error|warn|fail" /etc/ 2>/dev/null

# Search for hardcoded passwords
rg -i "password\s*=\s*['\"][^'\"]{3,}" --type py

# Find TODO items across project
rg -n "TODO|FIXME|HACK" --type-not log

# Search for specific API usage
```

```
rg "requests\.(get|post|put|delete)" --type py

# Find large functions (lines starting with 'def' followed by many lines)
rg -A 50 "^def " --type py | rg "^def " -c
```

CUT - Extract Columns

CUT is a command for extracting sections from each line of files. It can extract:

- Specific characters by position
- Fields separated by delimiters
- Byte ranges from binary data

CUT is particularly useful for processing structured data like CSV files, log files, and system outputs.

Basic Syntax

```
cut [OPTIONS] [FILE...]
cut -d'DELIMITER' -f'FIELDS' file      # Extract fields
cut -c'CHARACTERS' file                # Extract characters
cut -b'BYTES' file                     # Extract bytes
```

Field Options (-f)

Option	Description	Example
-f1	First field	cut -d',' -f1 data.csv
-f1,3	Fields 1 and 3	cut -d',' -f1,3 data.csv
-f2-5	Fields 2 through 5	cut -d',' -f2-5 data.csv
-f3-	From field 3 to end	cut -d',' -f3- data.csv
-f-2	From start to field 2	cut -d',' -f-2 data.csv

Character Options (-c)

Option	Description	Example
-c1-5	Characters 1-5	cut -c1-5 file.txt
-c10-	From character 10 to end	cut -c10- file.txt
-c-20	First 20 characters	cut -c-20 file.txt

-c1,5,10	Characters 1, 5, and 10	cut -c1,5,10 file.txt

```
# Sample data
echo "name,age,city,salary" > employees.csv
echo "john,25,nyc,50000" >> employees.csv
echo "jane,30,la,75000" >> employees.csv
echo "bob,22,chicago,45000" >> employees.csv

# Extract specific fields
cut -d',' -f1 employees.csv           # Names only
cut -d',' -f1,4 employees.csv         # Names and salaries
cut -d',' -f2-4 employees.csv         # Age, city, salary

# Character extraction
cut -c1-4 employees.csv               # First 4 characters of each line
cut -c6- employees.csv                # From 6th character to end

# System administration examples
cut -d':' -f1 /etc/passwd             # All usernames
cut -d':' -f1,3 /etc/passwd           # Usernames and UIDs
cut -d':' -f6 /etc/passwd             # Home directories

# Process information
ps aux | cut -c1-20                   # First 20 characters
ps aux | tr -s ' ' | cut -d' ' -f1,3,4 # User, CPU%, MEM%

# Log processing
cut -d' ' -f1 /var/log/apache2/access.log | sort | uniq -c # IP frequency
cut -d'"' -f2 /var/log/apache2/access.log | cut -d' ' -f1  # HTTP methods

# CSV with quotes handling (requires preprocessing)
sed 's/,/ /g' complex.csv | cut -d' ' -f2

# Extract email domains
cut -d'@' -f2 email_list.txt | sort | uniq -c

# Network interface information
ip addr show | grep "inet " | cut -d' ' -f6 | cut -d'/' -f1

# Extract time from timestamp
```

```
echo "2024-01-15 14:30:25" | cut -d' ' -f2 | cut -d':' -f1,2

# Process /proc files
cut -d' ' -f1 /proc/loadavg          # 1-minute load average
cut -f1 /proc/uptime                 # System uptime in seconds
```

AWK - Pattern Scanning & Processing

AWK is a powerful programming language designed for processing and analyzing structured text data. It works by:

1. Reading input line by line
2. Splitting each line into fields (by whitespace or custom delimiter)
3. Executing pattern-action rules on each line
4. Providing built-in variables and functions

AWK is particularly strong at:

- Field-based processing (columns in data)
- Mathematical calculations
- Pattern matching and conditional logic
- Report generation

Basic Syntax

```
awk 'pattern { action }' file
awk -F'delimiter' 'program' file     # Custom field separator
awk -f script.awk file               # Run AWK script from file
```

Built-in Variables

Variable	Description	Example
$0	Entire line	awk '{print $0}' file
$1, $2, $3...	Fields 1, 2, 3...	awk '{print $1, $3}' file
NF	Number of fields	awk '{print NF}' file
NR	Line number	awk '{print NR, $0}' file
FS	Field separator	awk 'BEGIN{FS=","} {print $1}' file
RS	Record separator	awk 'BEGIN{RS=";"} {print}' file

Pattern Types

```
# No pattern (all lines)
awk '{print $1}' file
```

```
# Regular expression pattern
awk '/pattern/ {print}' file
awk '/^[Ee]rror/ {print}' file

# Comparison patterns
awk '$2 > 50 {print}' file
awk 'length($0) > 80 {print}' file

# Range patterns
awk '/start/,/end/ {print}' file

# BEGIN and END patterns
awk 'BEGIN {print "Starting"} {print} END {print "Done"}' file

# Basic field processing
awk '{print $1}' /etc/passwd | head -5          # First field (usernames)
awk -F: '{print $1, $3}' /etc/passwd            # Username and UID
awk '{print $1, $3, $4}' file.txt               # Multiple fields

# Mathematical operations
awk '{sum += $2} END {print "Total:", sum}' numbers.txt
awk '{print $1, $2 * 1.1}' prices.txt           # Increase by 10%
awk '{avg = ($2 + $3 + $4) / 3; print $1, avg}' scores.txt

# Conditional processing
awk '$2 > 100 {print $1, "HIGH"}' data.txt          # Conditional output
awk 'length($0) > 80 {print NR, $0}' file.txt       # Long lines
awk 'NF > 5 {print "Line", NR, "has", NF, "fields"}' file.txt

# Pattern matching
awk '/error|ERROR/ {print NR, $0}' logfile.log      # Error lines with numbers
awk '/^#/ {next} {print}' config.conf               # Skip comments
awk '$1 ~ /^[0-9]+$/ {print}' mixed_data.txt        # Lines starting with numbers
```

Advanced AWK Features

```
# String functions
awk '{print toupper($1)}' file.txt              # Convert to uppercase
awk '{print substr($1, 1, 3)}' file.txt         # First 3 characters
awk '{gsub(/old/, "new"); print}' file.txt      # Global substitution
```

```
# Arrays (associative)
awk '{count[$1]++} END {for (i in count) print i, count[i]}' file.txt

# Custom field separators
awk -F',' '{print $1, $3}' data.csv                # CSV processing
awk -F'[,:]' '{print $2}' complex_data.txt         # Multiple separators

# Multiple patterns and actions
awk '/error/ {errors++} /warning/ {warnings++} END {print "Errors:", errors, "Warnings:",
warnings}' log.txt

# Functions and complex logic
awk '
function celsius(f) { return (f-32)*5/9 }
{print $1, celsius($2)}
' temperature_data.txt
```

TR - Translate/Delete Characters

TR (translate) is a command for translating or deleting characters from standard input. It operates on a character-by-character basis and is useful for:

- Character set conversion (uppercase/lowercase)
- Character replacement (spaces to underscores)
- Character deletion (removing unwanted characters)
- Character squeezing (reducing repeated characters)

TR reads from stdin and writes to stdout, making it perfect for pipelines.

Basic Syntax

```
tr [OPTION] SET1 [SET2]
tr 'source_chars' 'target_chars'       # Translate characters
tr -d 'chars_to_delete'                # Delete characters
tr -s 'chars_to_squeeze'               # Squeeze repeated characters
```

Character Sets

Set	Description	Example
'a-z'	Lowercase letters	tr 'a-z' 'A-Z'

'A-Z'	Uppercase letters	tr 'A-Z' 'a-z'
'0-9'	Digits	tr -d '0-9'
'[:lower:]'	POSIX lowercase	tr '[:lower:]' '[:upper:]'
'[:upper:]'	POSIX uppercase	tr '[:upper:]' '[:lower:]'
'[:digit:]'	POSIX digits	tr -d '[:digit:]'
'[:punct:]'	Punctuation	tr -d '[:punct:]'
'[:space:]'	Whitespace	tr -s '[:space:]'

Options

Option	Description	Example
-d	Delete characters	tr -d 'aeiou'
-s	Squeeze repeated characters	tr -s ' '
-c	Complement (invert set)	tr -cd '[:alnum:]'
-t	Truncate SET1 to SET2 length	tr -t 'abc' 'xy'

Basic Examples

```
# Case conversion
echo "Hello World" | tr 'a-z' 'A-Z'    # HELLO WORLD
echo "HELLO WORLD" | tr 'A-Z' 'a-z'    # hello world
echo "Hello World" | tr '[:lower:]' '[:upper:]' # HELLO WORLD

# Character replacement
echo "hello world" | tr ' ' '_'         # hello_world
echo "hello-world" | tr '-' '_'         # hello_world
echo "file.txt" | tr '.' '_'            # file_txt

# Delete characters
echo "Hello123World456" | tr -d '0-9'   # HelloWorld
echo "Hello, World!" | tr -d '[:punct:]' # Hello World
echo "  spaced  out  " | tr -d ' '      # spacedout
# Squeeze repeated characters
echo "hello    world" | tr -s ' '       # hello world (single spaces)
echo "aaabbbccc" | tr -s 'abc'          # abc
echo -e "line1\n\n\nline2" | tr -s '\n' # Remove blank lines

# ROT13 encoding
echo "Hello World" | tr 'A-Za-z' 'N-ZA-Mn-za-m' # Uryyb Jbeyq

# Clean up text files
tr -d '\r' < dos_file.txt > unix_file.txt # Remove Windows carriage returns
tr -s '\n' < file.txt                    # Remove blank lines
tr -cd '[:print:]\n' < file.txt          # Keep only printable characters

# Data cleaning
echo "Phone: (555) 123-4567" | tr -cd '[:digit:]' # 5551234567
echo "email@domain.com" | tr '[:upper:]' '[:lower:]' # normalize email
```

AWK is a powerful programming language designed for processing and analyzing structured text data. It works by:

1. Reading input line by line
2. Splitting each line into fields (by whitespace or custom delimiter)
3. Executing pattern-action rules on each line
4. Providing built-in variables and functions

AWK is particularly strong at:

- Field-based processing (columns in data)
- Mathematical calculations
- Pattern matching and conditional logic
- Report generation

Basic Syntax

```
awk 'pattern { action }' file
awk -F'delimiter' 'program' file      # Custom field separator
awk -f script.awk file                # Run AWK script from file
```

Built-in Variables

Variable	Description	Example

$0	Entire line	awk '{print $0}' file
$1, $2, $3...	Fields 1, 2, 3...	awk '{print $1, $3}' file
NF	Number of fields	awk '{print NF}' file
NR	Line number	awk '{print NR, $0}' file
FS	Field separator	awk 'BEGIN{FS=","} {print $1}' file
RS	Record separator	awk 'BEGIN{RS=";"} {print}' file

Pattern Types

```
# No pattern (all lines)
awk '{print $1}' file

# Regular expression pattern
awk '/pattern/ {print}' file
awk '/^[Ee]rror/ {print}' file

# Comparison patterns
awk '$2 > 50 {print}' file
awk 'length($0) > 80 {print}' file

# Range patterns
awk '/start/,/end/ {print}' file

# BEGIN and END patterns
awk 'BEGIN {print "Starting"} {print} END {print "Done"}' file
```

Practical Examples

```
# Basic field processing
awk '{print $1}' /etc/passwd | head -5          # First field (usernames)
awk -F: '{print $1, $3}' /etc/passwd            # Username and UID
awk '{print $1, $3, $4}' file.txt               # Multiple fields

# Mathematical operations
awk '{sum += $2} END {print "Total:", sum}' numbers.txt
awk '{print $1, $2 * 1.1}' prices.txt           # Increase by 10%
awk '{avg = ($2 + $3 + $4) / 3; print $1, avg}' scores.txt
```

```
# Conditional processing
awk '$2 > 100 {print $1, "HIGH"}' data.txt          # Conditional output
awk 'length($0) > 80 {print NR, $0}' file.txt       # Long lines
awk 'NF > 5 {print "Line", NR, "has", NF, "fields"}' file.txt

# Pattern matching
awk '/error|ERROR/ {print NR, $0}' logfile.log      # Error lines with numbers
awk '/^#/ {next} {print}' config.conf               # Skip comments
awk '$1 ~ /^[0-9]+$/ {print}' mixed_data.txt        # Lines starting with numbers
```

Advanced AWK Features

```
# String functions
awk '{print toupper($1)}' file.txt                  # Convert to uppercase
awk '{print substr($1, 1, 3)}' file.txt             # First 3 characters
awk '{gsub(/old/, "new"); print}' file.txt          # Global substitution

# Arrays (associative)
awk '{count[$1]++} END {for (i in count) print i, count[i]}' file.txt

# Custom field separators
awk -F',' '{print $1, $3}' data.csv                 # CSV processing
awk -F'[,:]' '{print $2}' complex_data.txt          # Multiple separators

# Multiple patterns and actions
awk '/error/ {errors++} /warning/ {warnings++} END {print "Errors:", errors, "Warnings:",
warnings}' log.txt

# Functions and complex logic
awk '
function celsius(f) { return (f-32)*5/9 }
{print $1, celsius($2)}
' temperature_data.txt
```

TR - Translate/Delete Characters

Theory

TR (translate) is a command for translating or deleting characters from standard input. It operates on a character-by-character basis and is useful for:

- Character set conversion (uppercase/lowercase)
- Character replacement (spaces to underscores)

- Character deletion (removing unwanted characters)
- Character squeezing (reducing repeated characters)

TR reads from stdin and writes to stdout, making it perfect for pipelines.

Basic Syntax

```
tr [OPTION] SET1 [SET2]
tr 'source_chars' 'target_chars'      # Translate characters
tr -d 'chars_to_delete'               # Delete characters
tr -s 'chars_to_squeeze'              # Squeeze repeated characters
```

Character Sets

Set	Description	Example
'a-z'	Lowercase letters	tr 'a-z' 'A-Z'
'A-Z'	Uppercase letters	tr 'A-Z' 'a-z'
'0-9'	Digits	tr -d '0-9'
'[:lower:]'	POSIX lowercase	tr '[:lower:]' '[:upper:]'
'[:upper:]'	POSIX uppercase	tr '[:upper:]' '[:lower:]'
'[:digit:]'	POSIX digits	tr -d '[:digit:]'
'[:punct:]'	Punctuation	tr -d '[:punct:]'
'[:space:]'	Whitespace	tr -s '[:space:]'

Options

Option	Description	Example
-d	Delete characters	tr -d 'aeiou'
-s	Squeeze repeated characters	tr -s ' '
-c	Complement (invert set)	tr -cd '[:alnum:]'
-t	Truncate SET1 to SET2 length	tr -t 'abc' 'xy'

Basic Examples

```
# Case conversion
```

```
echo "Hello World" | tr 'a-z' 'A-Z'    # HELLO WORLD
echo "HELLO WORLD" | tr 'A-Z' 'a-z'    # hello world
echo "Hello World" | tr '[:lower:]' '[:upper:]' # HELLO WORLD

# Character replacement
echo "hello world" | tr ' ' '_'         # hello_world
echo "hello-world" | tr '-' '_'         # hello_world
echo "file.txt" | tr '.' '_'            # file_txt

# Delete characters
echo "Hello123World456" | tr -d '0-9'   # HelloWorld
echo "Hello, World!" | tr -d '[:punct:]' # Hello World
echo "  spaced  out  " | tr -d ' '      # spacedout
```

Advanced Examples

```
# Squeeze repeated characters
echo "hello    world" | tr -s ' '       # hello world (single spaces)
echo "aaabbbccc" | tr -s 'abc'          # abc
echo -e "line1\n\n\nline2" | tr -s '\n' # Remove blank lines

# ROT13 encoding
echo "Hello World" | tr 'A-Za-z' 'N-ZA-Mn-za-m' # Uryyb Jbeyq

# Clean up text files
tr -d '\r' < dos_file.txt > unix_file.txt # Remove Windows carriage returns
tr -s '\n' < file.txt                   # Remove blank lines
tr -cd '[:print:]\n' < file.txt         # Keep only printable characters

# Data cleaning
echo "Phone: (555) 123-4567" | tr -cd '[:digit:]' # 5551234567
echo "email@domain.com" | tr '[:upper:]' '[:lower:]' # normalize email
```

HEAD & TAIL - Display File Portions

HEAD and TAIL are utilities for displaying the beginning and ending portions of files respectively. They are essential for:

- Previewing file contents without opening large files
- Monitoring log files in real-time (tail -f)
- Extracting specific portions of data
- Sampling large datasets

Both commands work with lines by default but can also work with bytes or characters.

HEAD - Show Beginning

Basic Syntax

```
head [OPTIONS] [FILE...]
head -n NUMBER file                 # Show NUMBER lines
head -c NUMBER file                 # Show NUMBER characters
```

Options

Option	Description	Example
-n NUM	Show NUM lines	head -n 20 file.txt
-c NUM	Show NUM bytes/characters	head -c 100 file.txt
-q	Quiet (suppress headers)	head -q file1 file2
-v	Verbose (show headers)	head -v file.txt

TAIL - Show End

Basic Syntax

```
tail [OPTIONS] [FILE...]
tail -n NUMBER file                 # Show last NUMBER lines
tail -f file                  # Follow file (real-time)
```

Options

Option	Description	Example
-n NUM	Show last NUM lines	tail -n 20 file.txt
-n +NUM	Show from line NUM to end	tail -n +10 file.txt
-c NUM	Show last NUM bytes	tail -c 100 file.txt
-f	Follow file (monitor)	tail -f /var/log/syslog

-F	Follow file (retry if deleted)	tail -F /var/log/app.log
--pid=PID	Stop following when PID dies	tail -f --pid=1234 log.txt

Practical Examples

```
# Basic usage
head file.txt                    # First 10 lines (default)
head -n 5 file.txt               # First 5 lines
head -c 50 file.txt              # First 50 characters

tail file.txt                    # Last 10 lines (default)
tail -n 15 file.txt              # Last 15 lines
tail -n +20 file.txt             # From line 20 to end

# Multiple files
head -n 3 *.txt                  # First 3 lines of each .txt file
tail -n 2 file1.txt file2.txt    # Last 2 lines of each file

# Monitoring logs
tail -f /var/log/syslog              # Follow system log
tail -f /var/log/apache2/error.log   # Follow Apache errors
tail -F /var/log/app.log             # Follow, retry if file rotated

# Extract middle sections
head -n 100 file.txt | tail -n 10    # Lines 91-100
tail -n +50 file.txt | head -n 20    # Lines 50-69
```

Advanced Examples

```
# Real-time monitoring with filtering
tail -f /var/log/syslog | grep ERROR
tail -f /var/log/apache2/access.log | awk '{print $1}' | sort | uniq -c

# File analysis
head -1 data.csv                 # Show CSV headers
tail -n +2 data.csv              # Skip CSV headers

# Performance monitoring
```

```
tail -f /var/log/apache2/access.log | awk '{print strftime("%H:%M:%S"), $0}'

# Multiple log monitoring
tail -f /var/log/{syslog,auth.log,kern.log}

# Conditional monitoring
tail -f app.log --pid=$(pgrep app_process)
```

CAT & TAC - Display Files

CAT (concatenate) is one of the most basic and versatile commands for displaying file contents. TAC is its reverse counterpart that displays files in reverse line order.

CAT Features:

- Display file contents
- Concatenate multiple files
- Create files from input
- Show special characters

TAC Features:

- Display lines in reverse order
- Useful for log analysis (newest entries first)

CAT - Display Forward

Basic Syntax

```
cat [OPTIONS] [FILE...]
cat file1 file2 > combined_file        # Concatenate files
cat > new_file                         # Create file from input
```

Options

Option	Description	Example
-n	Number all lines	cat -n file.txt
-b	Number non-blank lines	cat -b file.txt
-s	Suppress multiple blank lines	cat -s file.txt
-A	Show all characters	cat -A file.txt
-E	Show line endings ($)	cat -E file.txt

-T	Show tabs as ^I	cat -T file.txt
-v	Show non-printing characters	cat -v file.txt

TAC - Display Reverse

Basic Syntax

```
tac [OPTIONS] [FILE...]
tac file.txt                    # Display lines in reverse order
```

Practical Examples

CAT Examples

```
# Basic display
cat file.txt                    # Display file contents
cat file1.txt file2.txt             # Display multiple files

# File creation
cat > new_file.txt << EOF
Line 1
Line 2
Line 3
EOF

# File concatenation
cat *.txt > all_files.txt             # Combine all .txt files
cat header.txt data.txt footer.txt > complete.txt

# Numbering lines
cat -n file.txt                  # Number all lines
cat -b file.txt                  # Number only non-blank lines

# Show special characters
cat -A file.txt                   # Show all special characters
cat -E file.txt                  # Show line endings
cat -T file.txt                  # Show tabs

# Clean up files
```

```
cat -s messy_file.txt > clean_file.txt # Remove multiple blank lines
```

TAC Examples

```
# Reverse display
tac file.txt                    # Show lines in reverse order
tac /var/log/syslog | head -10      # Show last 10 log entries first

# Log analysis
tac access.log | grep "ERROR" | head -5 # Latest 5 errors
tac build.log | sed '/SUCCESS/q'      # Show from end until first SUCCESS
```

Advanced Examples

```
# File processing pipelines
cat /etc/passwd | cut -d: -f1 | sort   # Extract and sort usernames
cat *.log | grep ERROR | sort | uniq -c # Count errors across log files

# Document processing
cat document.txt | tr '[:upper:]' '[:lower:]' | wc -w # Count words (lowercase)

# Configuration file merging
cat /etc/default/* > combined_defaults.txt

# Binary file inspection
cat -A binary_file | head -20        # Show binary file with special chars

# Creating scripts
cat > backup_script.sh << 'EOF'
#!/bin/bash
tar -czf backup_$(date +%Y%m%d).tar.gz /home/user/important/
EOF
chmod +x backup_script.sh
```

NL - Number Lines

NL (Number Lines) is a command for adding line numbers to text files. Unlike cat -n, NL provides more sophisticated numbering options including:

- Conditional numbering (only non-empty lines, matching patterns)
- Custom number formats (width, alignment)

- Custom separators between numbers and text
- Different numbering styles

Basic Syntax

```
nl [OPTIONS] [FILE...]
nl file.txt                    # Number non-empty lines
```

Numbering Options

Option	Description	Example
-ba	Number all lines	nl -ba file.txt
-bt	Number non-empty lines (default)	nl -bt file.txt
-bn	No numbering	nl -bn file.txt
-bp PATTERN	Number lines matching pattern	nl -bp '^[A-Z]' file.txt

Format Options

Option	Description	Example
-nln	Left-aligned numbers	nl -nln file.txt
-nrn	Right-aligned numbers (default)	nl -nrn file.txt
-nrz	Right-aligned with leading zeros	nl -nrz file.txt
-w NUM	Width of number field	nl -w5 file.txt
-s STRING	Separator between number and line	nl -s': ' file.txt

Practical Examples

```
# Basic numbering
nl file.txt                    # Number non-empty lines
nl -ba file.txt                  # Number all lines
nl -bn file.txt                  # Display without numbers

# Custom formatting
nl -nln file.txt                 # Left-aligned numbers
nl -nrz -w4 file.txt               # Right-aligned, zero-padded, width 4
nl -s': ' file.txt                    # Custom separator

# Pattern-based numbering
nl -bp '^#' config.conf               # Number only comment lines
nl -bp 'ERROR' logfile.log            # Number only error lines

# Advanced formatting
nl -w3 -s' | ' -nln file.txt          # 3-digit width, custom separator, left-aligned
```

Real-world Examples

```
# Code review
nl -ba script.py                      # Number all lines for review
nl -bp '^def\|^class' script.py       # Number only function/class definitions

# Log analysis
nl -ba /var/log/syslog | tail -20     # Recent log entries with line numbers
nl -bp 'ERROR' application.log        # Number only error lines

# Documentation
nl -s'. ' -w2 steps.txt               # Format as numbered steps
nl -ba README.md | grep -n "TODO"     # Find TODO items with line numbers

# Configuration files
nl -bt /etc/nginx/nginx.conf          # Number non-empty config lines
```

FOLD - Wrap Lines

FOLD is a command for wrapping long lines of text to fit within specified column widths. It's useful for:

- Formatting text for display or printing
- Preparing text for terminals with limited width
- Email formatting (traditional 72-character limit)
- Breaking long lines without breaking words (optionally)

Basic Syntax

```
fold [OPTIONS] [FILE...]
fold -w WIDTH file                    # Wrap at WIDTH characters
fold -s file                          # Break at spaces only
```

Options

Option	Description	Example
-w WIDTH	Set line width	fold -w 60 file.txt
-s	Break at spaces only	fold -s file.txt
-b	Count bytes instead of columns	fold -b file.txt

Practical Examples

```
# Basic wrapping
fold file.txt                    # Wrap at 80 characters (default)
fold -w 50 file.txt              # Wrap at 50 characters
fold -w 72 email.txt             # Traditional email width

# Word-boundary wrapping
fold -s -w 60 document.txt       # Wrap at 60 chars, break at spaces
fold -s -w 40 long_paragraph.txt # Narrow columns, preserve words

# Processing long lines
echo "This is a very long line that should be wrapped at word boundaries" | fold -s -w 20

# Configuration and formatting
fold -w 80 README.md > README_formatted.md
cat long_config.conf | fold -s -w 60 | nl -ba
```

JOIN - Join Based on Common Field

JOIN is a command for joining lines of two files based on a common field, similar to SQL JOIN operations. It's useful for:

- Merging related data from separate files
- Database-like operations on text files
- Combining datasets with common keys
- Creating reports from multiple data sources

JOIN requires that both input files be sorted by the join field.

Basic Syntax

```
join [OPTIONS] FILE1 FILE2
join -t DELIMITER file1 file2       # Specify field delimiter
join -1 FIELD -2 FIELD file1 file2  # Join on specific fields
```

Options

Option	Description	Example
-t CHAR	Field delimiter	join -t',' file1.csv file2.csv
-1 FIELD	Join field for file1	join -1 2 -2 1 file1 file2
-2 FIELD	Join field for file2	join -1 1 -2 3 file1 file2
-a FILENUM	Include unpairable lines	join -a 1 file1 file2
-v FILENUM	Only unpairable lines	join -v 1 file1 file2
-e STRING	Replace missing fields	join -e 'NULL' -a 1 file1 file2
-o LIST	Output format	join -o 1.1,2.2 file1 file2

Join Types

- Inner Join (default): Only matching records
- Left Outer Join (-a 1): All from file1, matching from file2
- Right Outer Join (-a 2): All from file2, matching from file1
- Full Outer Join (-a 1 -a 2): All records from both files

Practical Examples

```
# Sample data preparation
echo -e "1,Alice,Engineering\n2,Bob,Sales\n3,Carol,Marketing" | sort > employees.txt
echo -e "1,50000\n2,45000\n3,55000" | sort > salaries.txt

# Basic join (inner join)
join -t',' employees.txt salaries.txt
# Output: 1,Alice,Engineering,50000

# Different join types
join -t',' -a 1 employees.txt salaries.txt    # Left outer join
join -t',' -a 2 employees.txt salaries.txt    # Right outer join
join -t',' -a 1 -a 2 employees.txt salaries.txt # Full outer join

# Join on different fields
echo -e "Alice,1\nBob,2\nCarol,3" | sort > names_ids.txt
join -t',' -1 2 -2 1 names_ids.txt salaries.txt # Join names_ids.field2 with salaries.field1
```

```
# Custom output format
join -t',' -o 1.2,2.2 employees.txt salaries.txt # Only name and salary

# Handle missing data
join -t',' -e 'N/A' -a 1 -a 2 employees.txt salaries.txt
```

Advanced Examples

```
# Multiple field joins (simulate with sort)
sort -t',' -k1,2 file1.csv > sorted1.csv
sort -t',' -k1,2 file2.csv > sorted2.csv
join -t',' -j1 1 -j2 1 sorted1.csv sorted2.csv

# Unmatched records analysis
join -t',' -v 1 employees.txt salaries.txt    # Employees without salaries
join -t',' -v 2 employees.txt salaries.txt    # Salaries without employees

# Complex data processing
cut -d: -f1,3 /etc/passwd | sort > users_uids.txt
cut -d: -f1,4 /etc/passwd | sort > users_gids.txt
join users_uids.txt users_gids.txt            # User info join
```

Real-world Use Cases

```
# Sales data analysis
sort -t',' -k1 customers.csv > customers_sorted.csv
sort -t',' -k1 orders.csv > orders_sorted.csv
join -t',' customers_sorted.csv orders_sorted.csv > customer_orders.csv

# Log correlation
awk '{print $1, $0}' access.log | sort > access_by_ip.log
awk '{print $1, $0}' error.log | sort > error_by_ip.log
join access_by_ip.log error_by_ip.log         # Correlate access and error logs

# System administration
getent passwd | cut -d: -f1,3 | sort > user_uids.txt
ps -eo user,pid,comm | tail -n +2 | sort > processes.txt
join user_uids.txt processes.txt              # Match users with processes
```

```
# Inventory management
sort -t',' -k1 inventory.csv > inventory_sorted.csv
sort -t',' -k1 orders.csv > orders_sorted.csv
join -t',' -a 1 -e '0' inventory_sorted.csv orders_sorted.csv # Stock vs orders
```

DIFF - Compare Files

DIFF is a command for comparing files and directories line by line. It shows:

- Lines that differ between files
- Added or deleted lines
- Context around changes
- Directory structure differences

DIFF is essential for:

- Version control (basis for git diff)
- Configuration management
- Code review
- System auditing

Basic Syntax

```
diff [OPTIONS] FILE1 FILE2
diff [OPTIONS] DIR1 DIR2          # Compare directories
```

Output Formats

Option	Format	Description
(default)	Normal	Traditional diff format
-c	Context	Show context around changes
-u	Unified	Modern unified format (used by git)
-y	Side-by-side	Two-column comparison
--brief	Brief	Only report if files differ

Common Options

Option	Description	Example

-i	Ignore case	diff -i file1.txt file2.txt
-w	Ignore whitespace changes	diff -w file1.txt file2.txt
-b	Ignore space changes	diff -b file1.txt file2.txt
-B	Ignore blank lines	diff -B file1.txt file2.txt
-r	Recursive (directories)	diff -r dir1/ dir2/
-q	Brief output	diff -q file1.txt file2.txt

Practical Examples

```
# Sample files
echo -e "line1\nline2\nline3" > file1.txt
echo -e "line1\nline2 modified\nline3\nline4" > file2.txt

# Basic comparison
diff file1.txt file2.txt              # Normal format
diff -u file1.txt file2.txt           # Unified format (recommended)
diff -c file1.txt file2.txt           # Context format

# Side-by-side comparison
diff -y file1.txt file2.txt           # Two-column view
diff -y -W 120 file1.txt file2.txt    # Custom width

# Ignore various differences
diff -i file1.txt file2.txt           # Ignore case
diff -w file1.txt file2.txt           # Ignore whitespace
diff -b file1.txt file2.txt           # Ignore space changes

# Directory comparison
diff -r dir1/ dir2/                   # Recursive directory diff
diff -r --brief dir1/ dir2/           # Brief directory comparison
```

Advanced Examples

```
# Configuration file comparison
diff -u /etc/nginx/nginx.conf /etc/nginx/nginx.conf.backup
diff -w config.conf config.conf.new    # Ignore whitespace changes

# Exclude files from directory diff
diff -r --exclude="*.log" --exclude="tmp/" dir1/ dir2/
diff -r --exclude-from=exclude_list.txt dir1/ dir2/

# Generate patches
diff -u original.c modified.c > changes.patch
patch original.c < changes.patch       # Apply the patch

# Binary file comparison
diff --brief binary1 binary2           # Just check if different
cmp binary1 binary2                    # Alternative for binary files
```

Real-world Use Cases

```
# System administration
diff /etc/passwd /etc/passwd.backup    # Check user account changes
diff -r /etc/ /etc.backup/             # Full system configuration diff

# Development
diff -u old_version.py new_version.py > code_changes.patch
diff -r --exclude="*.pyc" src/ backup_src/ # Compare source directories

# Log analysis
diff yesterday.log today.log | grep "^>" # New log entries
diff -u access.log.1 access.log        # Compare log files

# Backup verification
diff -r /home/user/ /backup/user/      # Verify backup completeness
diff -q /important/file /backup/important/file # Quick check

# Documentation
diff -u README.md README.md.new        # Review documentation changes
```

Understanding DIFF Output

```
# Normal format symbols:
# a = add
# c = change
# d = delete
# < = from first file
# > = from second file

# Unified format symbols:
# - = removed from first file
# + = added to second file
# @@ = line number context
```

COMM - Compare Sorted Files

COMM (common) compares two sorted files line by line and produces three-column output:

1. Column 1: Lines unique to file1
2. Column 2: Lines unique to file2
3. Column 3: Lines common to both files

COMM is particularly useful for:

- Set operations (union, intersection, difference)
- Comparing sorted lists
- Finding commonalities between datasets
- Analyzing differences in structured data

Important: Both files must be sorted for COMM to work correctly.

Basic Syntax

```
comm [OPTIONS] FILE1 FILE2
sort file1 | comm - sorted_file2       # Use with pipes
```

Options

Option	Description	Example
-1	Suppress column 1 (unique to file1)	comm -1 file1 file2
-2	Suppress column 2 (unique to file2)	comm -2 file1 file2
-3	Suppress column 3 (common lines)	comm -3 file1 file2
-12	Show only common lines	comm -12 file1 file2
-23	Show only lines unique to file1	comm -23 file1 file2
-13	Show only lines unique to file2	comm -13 file1 file2

Set Operations

Operation	Command	Description
Intersection	comm -12 file1 file2	Lines in both files
Union	comm file1 file2 | tr -d '\t'	All unique lines
Difference A-B	comm -23 file1 file2	In file1 but not file2

Difference B-A	comm -13 file1 file2	In file2 but not file1
Symmetric Diff	comm -3 file1 file2	In either but not both

Practical Examples

```
# Sample sorted files
echo -e "apple\nbanana\ncherry\ndate" > fruits1.txt
echo -e "banana\ncherry\nelderberry\nfig" > fruits2.txt

# Basic comparison (three columns)
comm fruits1.txt fruits2.txt
#    apple
#            banana
#            cherry
#        elderberry
#    date
#        fig

# Set operations
comm -12 fruits1.txt fruits2.txt      # Common items: banana, cherry
comm -23 fruits1.txt fruits2.txt      # Only in fruits1: apple, date
comm -13 fruits1.txt fruits2.txt      # Only in fruits2: elderberry, fig
comm -3 fruits1.txt fruits2.txt       # Symmetric difference: apple, date, elderberry, fig
```

Advanced Examples

```
# Working with unsorted files
sort file1.txt > temp1
sort file2.txt > temp2
comm temp1 temp2
rm temp1 temp2

# Pipeline usage
sort users_today.txt | comm -23 - <(sort users_yesterday.txt) # New users
sort current_processes.txt | comm -12 - <(sort baseline_processes.txt) # Common processes

# Multiple file comparison
sort file1.txt > sorted1
sort file2.txt > sorted2
sort file3.txt > sorted3
comm -12 sorted1 sorted2 | comm -12 - sorted3  # Common to all three

# Case-insensitive comparison
sort -f file1.txt | comm -12 - <(sort -f file2.txt)
```

SPLIT - Split Files

SPLIT is a command for dividing large files into smaller pieces. It's useful for:

- Managing large files that are difficult to handle
- Preparing files for transfer over limited connections
- Parallel processing of large datasets
- Backup strategies (splitting archives)
- Email attachments (size limitations)

SPLIT can divide files by:

- Line count (-l)
- Byte size (-b)
- Line boundaries at byte limits (-C)

Basic Syntax

```
split [OPTIONS] [INPUT [PREFIX]]
split -l LINES file prefix           # Split by line count
split -b SIZE file prefix            # Split by byte size
split -C SIZE file prefix            # Split by size at line boundaries
```

Size Options

Suffix	**Unit**	**Example**
b	**Bytes**	**split -b 1024b file**
k	**Kilobytes (1024 bytes)**	**split -b 10k file**
m	**Megabytes**	**split -b 5m file**
g	**Gigabytes**	**split -b 1g file**

Common Options

Option	**Description**	**Example**
-l NUM	**Split by line count**	**split -l 1000 file.txt**
-b SIZE	**Split by byte size**	**split -b 1m file.txt**

-C SIZE	**Split by size, break at lines**	**split -C 1m file.txt**
-d	**Use numeric suffixes**	**split -d -l 100 file.txt part_**
-a LENGTH	**Suffix length**	**split -a 3 -l 100 file.txt**
--verbose	**Show progress**	**split --verbose -l 1000 file.txt**

Practical Examples

```
# Basic splitting
split -l 1000 large_file.txt chunk_    # 1000 lines per chunk
split -b 1m video.mp4 video_part_     # 1MB chunks
split -C 500k textfile.txt text_       # 500KB chunks, break at lines

# Custom naming
split -d -l 500 data.txt part_         # part_00, part_01, etc.
split -a 3 -l 100 file.txt section_    # section_aaa, section_aab, etc.

# Size-based splitting
split -b 10m backup.tar.gz backup_     # 10MB chunks for email
split -b 650m movie.avi cd_            # CD-sized chunks (650MB)

# Line-boundary splitting (for text files)
split -C 1m logfile.log log_section_   # 1MB chunks, don't break lines
```

Rejoining Split Files

```
# Rejoin files in order
cat chunk_* > original_file.txt

# For binary files
cat video_part_* > reconstructed_video.mp4

# Verify integrity
md5sum original_file.txt
md5sum reconstructed_file.txt

# Rejoin numbered splits
cat part_{00..99} > complete_file.txt  # If you know the range
```

STRINGS - Extract Printable Strings

STRINGS extracts and displays printable character sequences from binary files. It's essential for:

- Analyzing binary executables for embedded text
- Forensic analysis of files
- Reverse engineering and malware analysis
- Finding embedded configuration in binaries
- Debugging binary file corruption

STRINGS finds sequences of printable characters that are at least a certain length (default 4 characters) and displays them one per line.

Basic Syntax

```
strings [OPTIONS] [FILE...]
strings binary_file                    # Extract printable strings
strings -n LENGTH file                 # Minimum string length
```

Options

Option	Description	Example
-n NUM	Minimum string length	strings -n 8 binary_file
-a	Scan entire file	strings -a executable
-t FORMAT	Show offset of string	strings -t x binary_file
-e ENCODING	Character encoding	strings -e l binary_file
-f	Show filename before strings	strings -f *.bin
-o	Show octal offset	strings -o binary_file

Encoding Options (-e)

Encoding	Description	Example
s	Single 7-bit (default)	strings -e s file
S	Single 8-bit	strings -e S file
b	16-bit bigendian	strings -e b file

l	16-bit littleendian	strings -e l file
B	32-bit bigendian	strings -e B file
L	32-bit littleendian	strings -e L file

Practical Examples

```
# Basic string extraction
strings /bin/ls                    # Strings in ls command
strings -n 10 /usr/bin/gcc         # Strings at least 10 chars long
strings -a program.exe             # Scan entire file

# With offset information
strings -t x /bin/cat              # Hexadecimal offsets
strings -t d binary_file           # Decimal offsets
strings -o program.bin             # Octal offsets

# Multiple files
strings -f /usr/bin/*              # Show filename with each string
strings *.so | grep -i error       # Find error strings in libraries
```

Advanced Examples

```
# Forensic analysis
strings -n 8 suspicious_file | grep -E "(password|key|secret)"
strings -a disk_image.dd | grep -E "email|@" | head -20

# Configuration discovery
strings config.bin | grep -E "^[A-Z_]+=" # Environment-style configs
strings firmware.bin | grep -E "^\/" | head -10 # File paths

# Error message analysis
strings program | grep -i "error\|fail\|exception" | sort | uniq

# URL and domain extraction
strings binary_file | grep -E "https?://" | head -10
strings malware.bin | grep -E "\.(com|net|org)" | sort | uniq

# Version information
strings program.exe | grep -E "version|v[0-9]" -i
strings library.so | grep -E "copyright|license" -i
```