1. Introduction to UNIX Operating System and Basic UNIX Commands

Introduction to UNIX Operating System

UNIX is a powerful, multiuser, multitasking operating system originally developed in 1969 at Bell Labs. It provides a stable, multiuser environment that manages hardware and software resources efficiently. It is widely used on servers, workstations, and mainframes.

Key Features:

- Portable (can run on different hardware)
- Multiuser and multitasking
- Hierarchical file system
- Extensible through shell programming and utilities

Operating System Architecture & Components

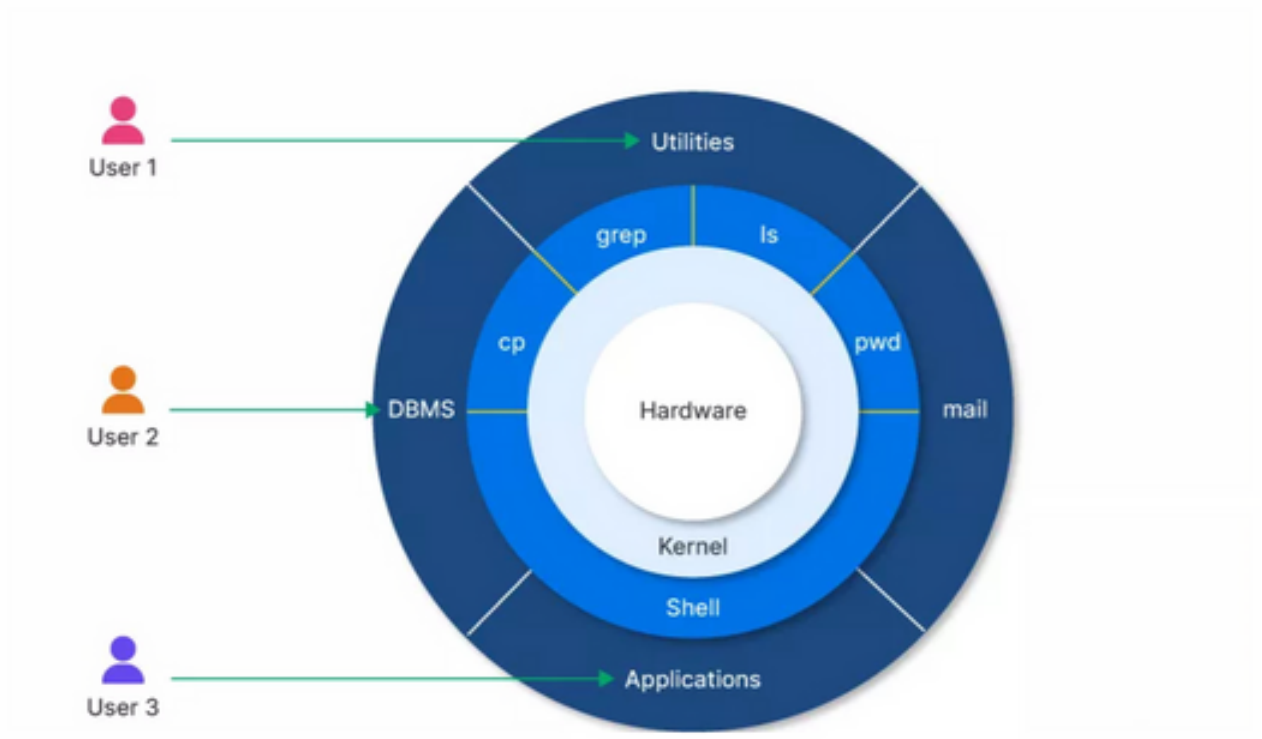
UNIX architecture is layered and divided into four main components:

1. Hardware: Physical devices like CPU, memory, and storage devices.
2. Kernel: Core of the OS, directly interfaces with **hardware** and **manages resources** such as **memory, processes, devices, file systems,** and **security**.
3. Shell (Command Interpreter): **Interface** between the **user** and **kernel**. It interprets commands and executes programs. Common shells include bash, C shell, and Korn shell.
4. Application Programs: Software utilities and user programs running in user space.

Kernel Functions:

- Process **scheduling** and **management**
- **Memory** management and **virtual memory**
- Device management
- File system handling
- Inter-process communication
- System calls handling

Diagram:



What is Kernel?

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

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

Complete Linux system = Kernel + GNU_system utilities and libraries + other management scripts + installation scripts.

What is Shell?

A shell is a special user program that provides an **interface** for the **user** to use **operating system** services. Shell accepts **human-readable commands** from users and converts them into something which the **kernel can understand**. It is a **command** language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.

Basic UNIX Commands

Here are some fundamental UNIX commands to get started:

```
ls          # List files and directories
pwd         # Print working directory
cd /path    # Change directory
mkdir dir   # Create new directory
rmdir dir   # Remove empty directory
cp file1 file2  # Copy file
mv file1 file2  # Move or rename file
rm file     # Remove file
cat file    # Display file contents
touch file  # Create empty file or update timestamp
chmod 755 file # Change file permissions
ps          # List running processes
```

How to create a Shell script with simple hello world Program – Save & Run

Create a file `hello.sh` with the following content:

```
#!/bin/bash
# This is a simple shell script
echo "Hello, World!"
```

Run steps:

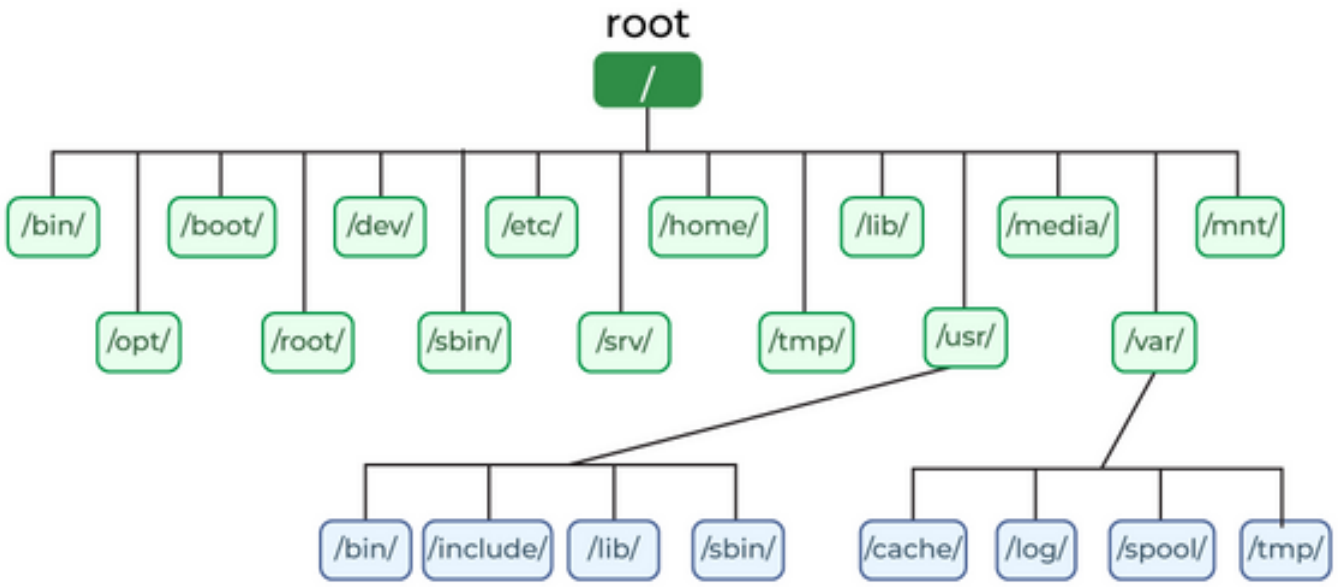
```
chmod +x hello.sh    # Make script executable
./hello.sh           # Run the script
```

2. UNIX File System

UNIX File System

The UNIX file system is hierarchical, starting from the **root directory** `/`. It supports files and directories and manages file storage efficiently with **metadata** stored in inodes.

Diagram:



File Types

UNIX supports several file types:

- **Regular files**: Data files containing user information or program data.
- **Directory files**: Nodes containing other files or directories.
- **Special files:** Device files (block or character devices).
- **Symbolic links**: Pointers to other files.
- **Pipes and sockets**: For inter-process communication.

Diagram

Symbol	File Types
-	Regular File
d	Directory

```
|      l       | Link File                |
|      c       | Character Device File    |
|      s       | Local Socket File        |
|      p       | Named Pipe File          |
|      b       | Block Device File        |
+--------------+--------------------------+
```

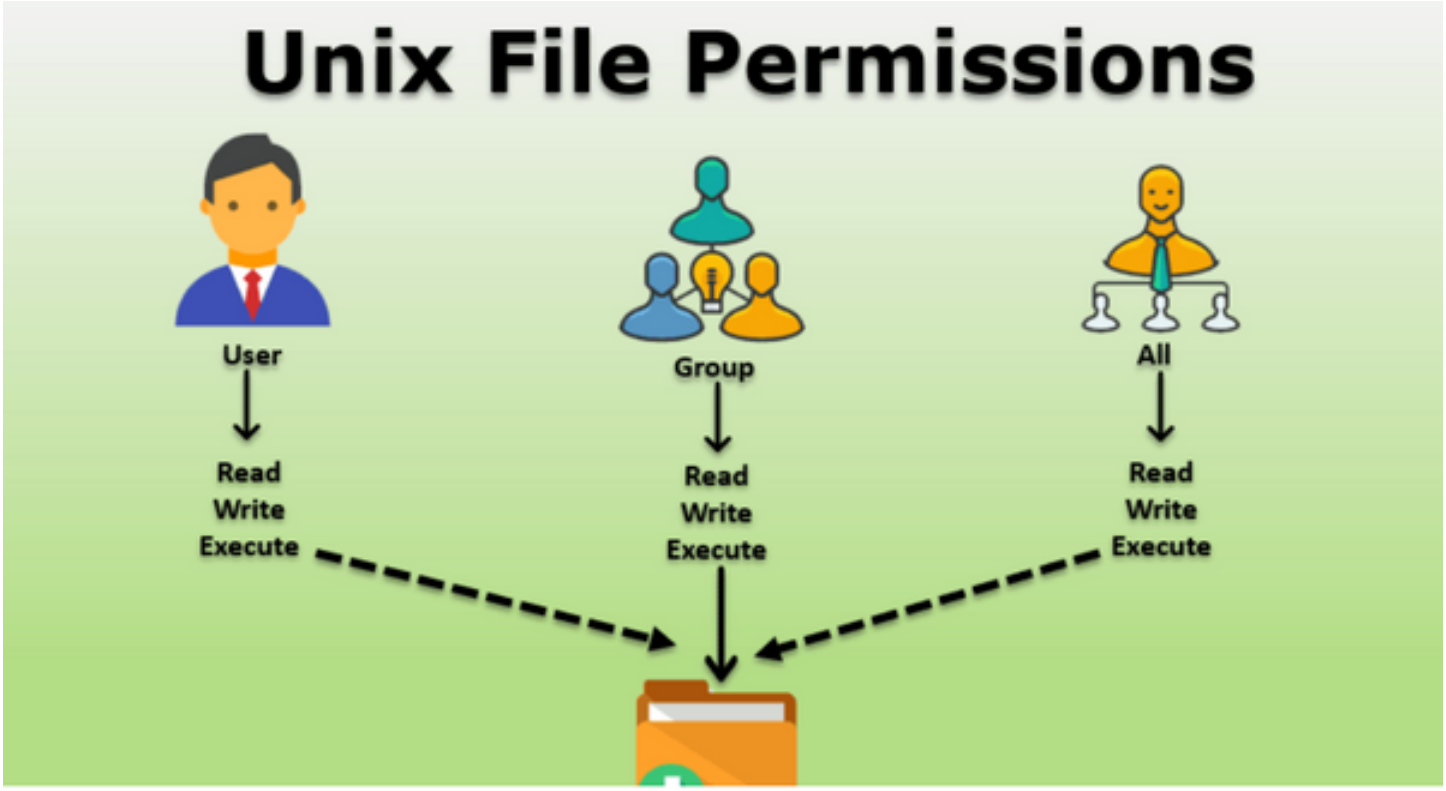
File Permissions

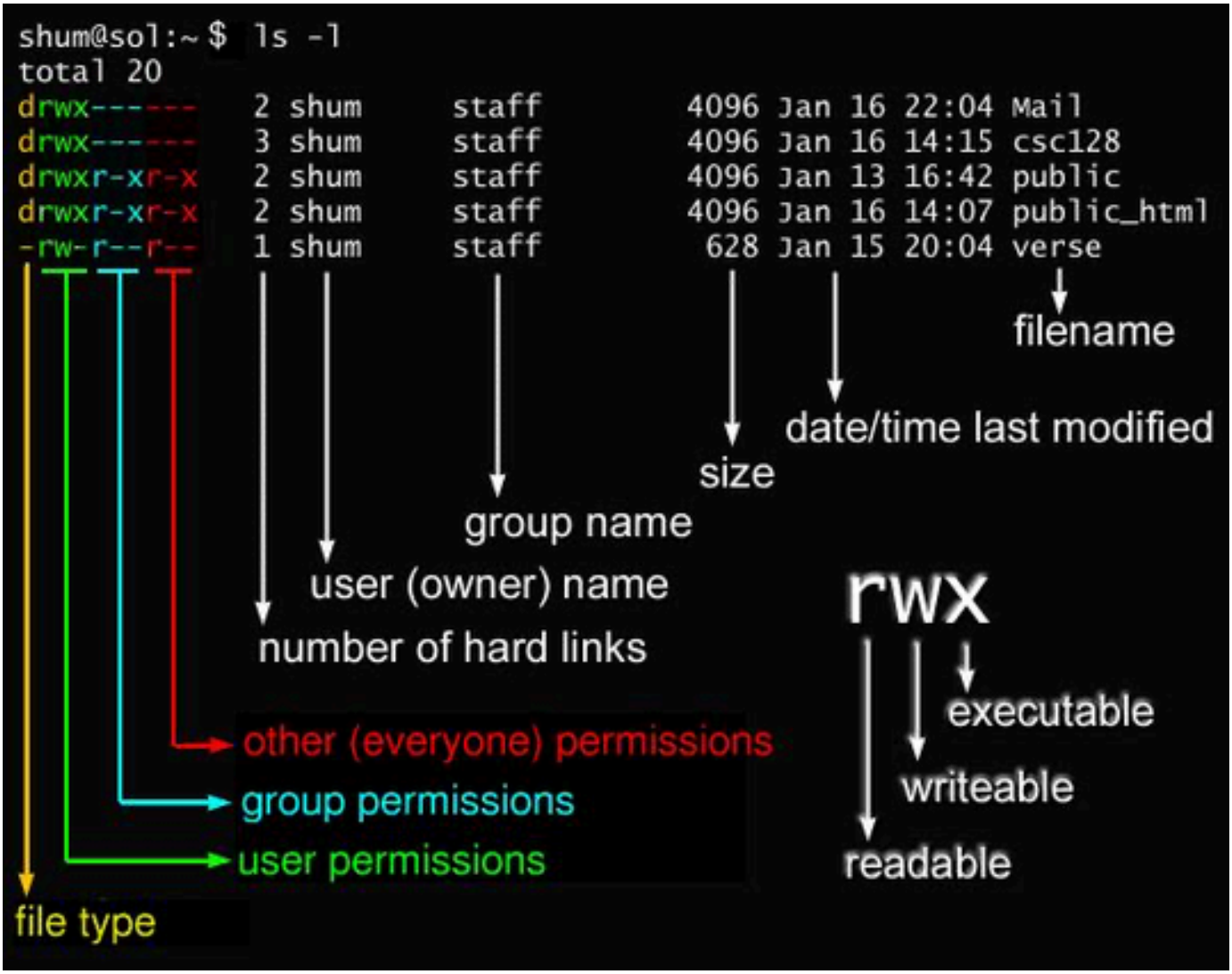
Permissions define who can read (r), write (w), or execute (x) a file. These are split into:

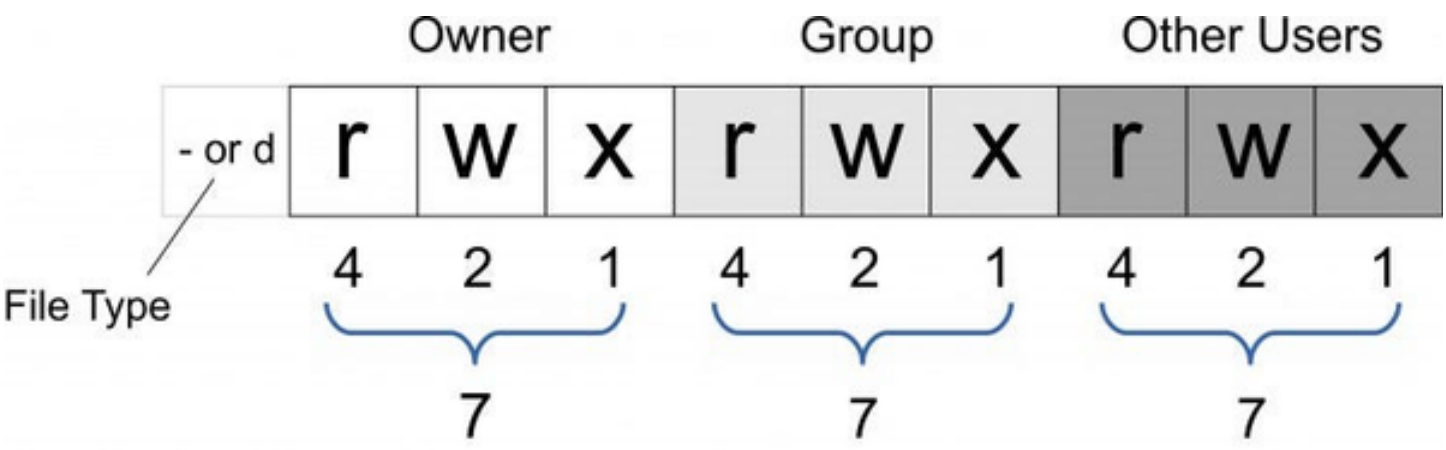
- Owner permissions
- Group permissions
- Others permissions

Example permission string: `-rwxr-xr-`

Diagram




shum@sol:~ $ ls -l
total 20
drwx------ 2 shum staff 4096 Jan 16 22:04 Mail
drwx------ 3 shum staff 4096 Jan 16 14:15 csc128
drwxr-xr-x 2 shum staff 4096 Jan 13 16:42 public
drwxr-xr-x 2 shum staff 4096 Jan 16 14:07 public_html
-rw-r--r-- 1 shum staff 628 Jan 15 20:04 verse
filename
date/time last modified
size
group name
user (owner) name
number of hard links
other (everyone) permissions
group permissions
user permissions
file type
rwx
executable
writeable
readable


Owner
Group
Other Users
- or d
r w x r w x r w x
4 2 1 4 2 1 4 2 1
7 7 7
File Type

I-node entries

An **i-node** holds **metadata** about a file such as its **size, owner, permissions, timestamps,** and **pointers** to data blocks. Each file or directory has one inode.

Diagram

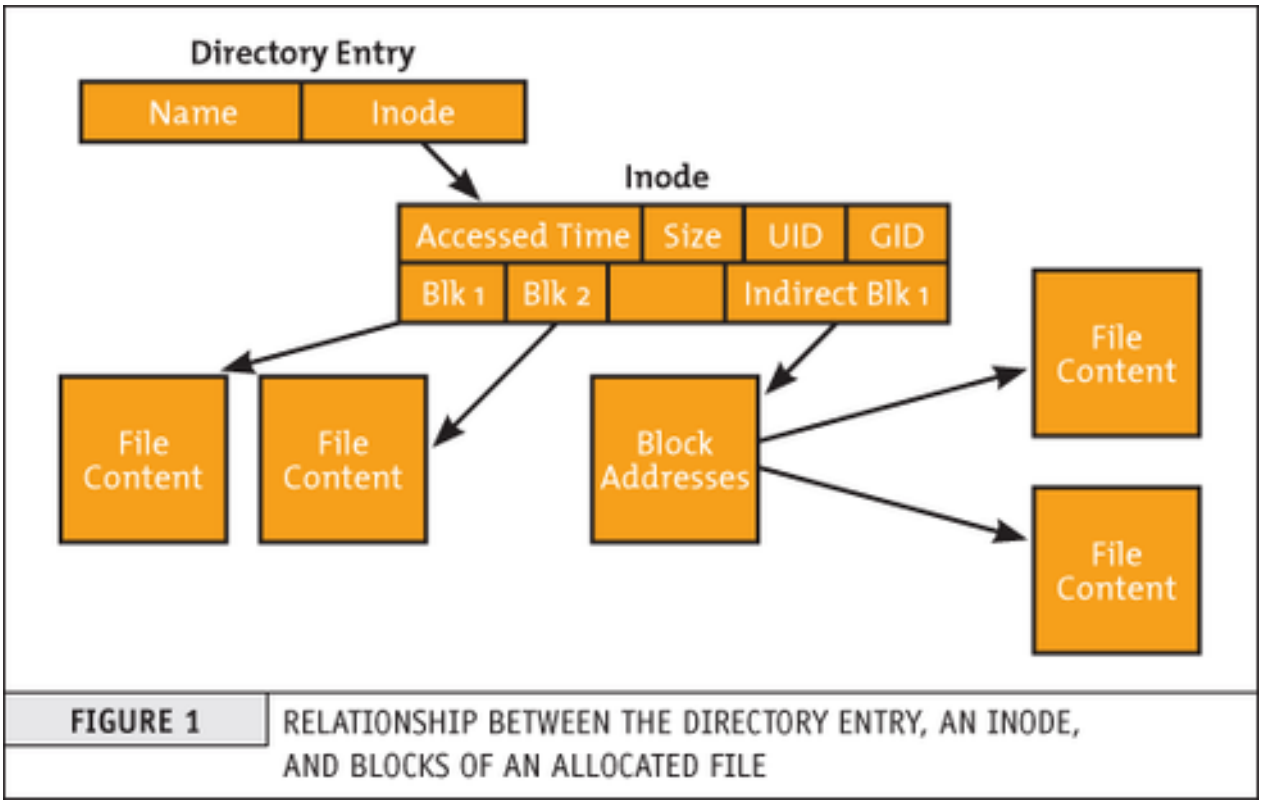


FIGURE 1 RELATIONSHIP BETWEEN THE DIRECTORY ENTRY, AN INODE, AND BLOCKS OF AN ALLOCATED FILE

File Related Commands

```
ls -l              # List files with permissions and details
stat filename      # Show detailed inode info of a file
chmod 755 filename # Change permissions
chown user file    # Change owner
find /path -name "*.txt"   # Find files matching pattern
```

```
file filename      # Show file type
```

Pipe Concept – Std input, output and Error

A pipe is a mechanism for **interprocess communication (IPC)** that allows the output of one process to be used as the input of another process. It provides a **unidirectional** flow of data, meaning data flows from one end (the "write" end) to the other (the "read" end).

- **Connecting Processes:**Pipes serve as a conduit to link the standard output (stdout) of one command or program to the standard input (stdin) of another. This creates a "pipeline" of commands, where each command in the sequence processes the output of the preceding one.
- **Unidirectional Flow:**Data travels in one direction only. A process writes data into the pipe, and another process reads data from it.
- **In-Memory Buffer:**Pipes are typically implemented as an in-memory buffer managed by the operating system kernel. This means data doesn't need to be written to and read from disk, which significantly improves efficiency for chaining commands.
- Standard Input (stdin): Default input to a command, usually keyboard.
- Standard Output (stdout): Default output, usually terminal screen.
- Standard Error (stderr): Output for error messages.

You can use | (pipe) to send stdout of one command as stdin to another, e.g.,

```
ls -l | grep "txt"   # Lists only files with 'txt' in details
```

Redirect outputs:

```
command > out.txt    # Redirect stdout to file
command 2> error.txt # Redirect stderr to file
command &> all.txt   # Redirect both stdout and stderr
```

3. Unix Environment

Shells (bash, Korn, C shell)

Bash (Bourne-Again SHell), sh (Bourne Shell), and Korn Shell (ksh) are all **command-line interpreters**, or shells, used in Unix-like operating systems.They provide an interface for users to interact with the operating system and execute commands or scripts.

- Bash (Bourne Again Shell): Most common shell, compatible with sh, with advanced scripting features.
- Korn Shell (ksh): Compatible with sh, includes features for scripting and interactive use.
- C Shell (csh, tcsh): Syntax similar to C programming, used less commonly today.

sh (Bourne Shell):

- This was the original and standard Unix shell, developed by Stephen Bourne at Bell Labs.
- It is known for its simplicity and portability, forming the basis for many shell scripting conventions.
- Many scripts are written to be compatible with sh to ensure broad compatibility across different Unix systems.

ksh (Korn Shell):

- Developed by David Korn, ksh aimed to combine the best features of sh and the C Shell (csh), offering more advanced scripting capabilities and interactive features.
- It introduced features like command-line editing, job control, and associative arrays, making it more powerful for both scripting and interactive use than sh.
- ksh is often favored in enterprise Unix environments like AIX, Solaris, and HP-UX.

Bash (Bourne-Again SHell):

- Developed by Brian Fox for the GNU Project, Bash is a free and open-source shell that is widely used on Linux and other Unix-like systems.
- It was designed to be compatible with the Bourne Shell (sh) while incorporating features from both ksh and csh, such as command-line editing, history, and programmable completion.
- Bash has become the default shell for most Linux distributions and macOS due to its rich feature set, active community support, and open-source nature.

System Variables

System variables in Unix-like operating systems, often referred to as environment variables, are dynamic named values that can affect the way running processes behave. They are part of the shell environment and can be accessed by programs and scripts.

Here are some common system variables and their uses:

- **PATH:** This variable specifies the directories the shell searches for executable commands. When you type a command, the shell looks in each directory listed in PATH until it finds the command.
- **HOME**: This variable stores the path to the current user's home directory.
- **PWD**: This variable indicates the present working directory, which is the directory you are currently in.
- **HOSTNAME**: This variable holds the name of the machine or host.

- **UID**: This variable stores the User ID of the current user. It is often used in scripts to check if a user has root privileges.
- **SHELL**: This variable specifies the default shell for the current user.
- **USER**: This variable stores the username of the current user.
- **LANG**: This variable defines the default locale and character encoding settings for the system.
- **TERM**: This variable specifies the type of terminal emulation being used.
- **PS1**: This variable defines the primary prompt string displayed in the terminal.

Environment variables like

- `PATH`,
- `HOME`,
- `USER`

define your session environment.

Example:

```
echo $PATH

export MYVAR="Hello"
```

Set options

Are command is a built-in shell command used to **control the behavior** of the shell and manage shell options and positional parameters.

Commonly Used Options:

- -e (errexit): Exits immediately if a command exits with a non-zero status (indicating an error).
- -u (nounset): Treats unset variables as an error, causing the shell to exit.
- -x (xtrace): Prints commands and their arguments as they are executed, useful for debugging.
- -f (noglob): Disables filename expansion (globbing).
- -v (verbose): Prints shell input lines as they are read.
- -C (noclobber): Prevents redirection from overwriting existing files.

You can set shell options using:

```
set -o option    # enable option

set +o option    # disable option
```

Or shorter flags in bash:

```
set -e   # Exit on error
```

```
set -x  # Debug/trace commands
```

The Process

A process is an executing instance of a program. Use `ps`, `top` to view processes.

It is an instance of a running program.It represents an active execution of a set of instructions and associated data.Every time a program is launched, a command is executed, or an application is started, a new process is initiated.

- **Process ID (PID):** A unique numerical identifier assigned to each active process.
- **Parent Process ID (PPID):** The PID of the process that created the current process.
- **Process States:** Processes transition through various states during their lifecycle, such as:
  - **Running/Runnable:** The process is currently executing on the CPU or is ready to be executed.
  - **Sleeping:** The process is temporarily inactive, waiting for an event (e.g., I/O completion, a signal).
  - **Stopped:** The process has been paused, often by a user or another process.
  - **Zombie:** The process has finished execution but its entry in the process table remains until its parent process collects its exit status.
- **Process Creation:** New processes are typically created using the fork() system call, which creates a child process as a copy of the parent. The exec() system call can then be used to load a new program into the child process's address space.
- **Process Management Commands:** Several commands are available for managing processes, including:
  - ps: Displays information about currently running processes.
  - top: Provides a dynamic, real-time view of processes and system resource usage.
  - kill: Sends a signal to terminate or control a process.
  - renice: Changes the scheduling priority of a process.
- **Daemons:** Background processes that run independently of a controlling terminal and typically perform system-level tasks.

Running commands in background

Add `&` after command:

```
sleep 100 &
```

View background jobs with `jobs`, bring back with `fg`.

Cron scheduler

Cron is used to schedule repetitive tasks (cron jobs).Cron is a time-based job scheduler in Unix-like operating systems, including Linux. It allows users to automate the execution of tasks (known as "cron jobs") at specified intervals, such as minutes, hours, days, months, or weekdays.

https://crontab.guru/

Edit cron jobs with:

crontab -e

Example cron entry (run script every day at 6 AM):

```
text
```

0 6 * * * /path/to/script.sh

4. Filters

A filter is a program or command that primarily receives its input from standard input (stdin), performs some operation or transformation on that data, and then writes its output to standard output (stdout). This design principle allows for modularity and the creation of pipelines, where the output of one filter can be seamlessly directed as the input to another.

Simple Filters (grep*Global Regular Expression Print*, head, tail & tr)

- `grep "pattern" file`: Search lines matching pattern
- `head -n 10 file`: First 10 lines
- `tail -n 10 file`: Last 10 lines
- `tr 'a-z' 'A-Z'`: Translate lowercase to uppercase characters

Common Filter Commands:

- `grep:` Filters lines that match a specified pattern.
- `sort`: Sorts lines of text alphabetically or numerically.
- `uniq`: Filters out duplicate adjacent lines.
- `cat`: Concatenates files and prints to stdout; can also be used to simply display file content or act as a basic filter for stdin.
- `sed`: A stream editor used for powerful text transformations and substitutions.
- `awk`: A programming language designed for text processing and pattern matching.
- `cut`: Extracts specific columns or fields from lines of text.
- `tr`: Translates or deletes characters.
- `wc`: Counts lines, words, and characters.
- `more / less:` Paginates output, allowing users to view large files or command output one screen at a time.

Advanced Filters (sort & find)

- `sort file`: Sort file lines
- `find /path -name "pattern"`: Find files

Tools & Regular Expressions (sed & awk)

- `sed 's/old/new/g' file`: Stream editor for text substitution
- `awk '{print $1, $3}' file`: Pattern scanning and processing language

Search – File & Nested search

```
grep -r "pattern" /path   # Recursively search in files

find /path -type f -exec grep -H "pattern" {} \;  # Search with find + grep
```

5. Basic OS commands to check Processes, memory, CPU & Thread, Storage/space

Process Related Commands

```
ps aux          # List all processes
top             # Dynamic real-time process viewer
htop            # Enhanced process viewer (needs install)
kill PID        # Kill process by PID
```

Memory and CPU

```
free -m         # Show memory usage in MB
vmstat 1        # Reports system stats every 1 second
iostat          # CPU and I/O statistics
```

Thread

```
ps -eLf         # List processes with threads
top -H          # Show threads in top
```

Storage/Space

```
df -h           # Disk free space in human readable format
du -sh directory  # Size of directory
lsblk           # List block devices
```

Vi Editor

Vi Editor

Vi is a powerful text editor present in all UNIX/Linux systems.

Input Mode Commands

- Press `i` to enter insert mode
- Press `ESC` to exit insert mode

Vi Editor – Save & Quit

- `:w` - save file
- `:q` - quit editor
- `:wq` or `ZZ` - save and quit
- `:q!` - quit without saving

Vim Cursor Movement Commands

Command	Action
h	Move **left** one character
l	Move **right** one character
j	Move **down** one line
k	Move **up** one line
0	Move to the **beginning of the current line**
^	Move to the **first non-blank character** of the line
$	Move to the **end of the current line**
gg	Move to the **top of the file**
G	Move to the **end of the file**
nG	Move to **line number n** (e.g., 25G goes to line 25)
w	Move **forward by one word**
b	Move **backward by one word**
e	Move to the **end of the current/next word**
H	Move to the **top line** of the screen
M	Move to the **middle line** of the screen

L	Move to the **last line** of the screen
Ctrl + d	Move **down half a screen**
Ctrl + u	Move **up half a screen**
Ctrl + f	Move **forward one full screen**
Ctrl + b	Move **backward one full screen**

Using Ctrl (Control) Key:

Shortcut	Action
Ctrl + A	Move cursor to the beginning of the line
Ctrl + E	Move cursor to the end of the line
Ctrl + B	Move backward one character
Ctrl + F	Move forward one character
Ctrl + U	Cut (delete) everything before the cursor
Ctrl + K	Cut everything after the cursor
Ctrl + W	Delete the word before the cursor
Ctrl + Y	Paste the last cut text (yank)
Ctrl + L	Clear the terminal screen (same as clear)