Session 1 and 2:

#Introduction to Operating System

An Operating System (OS) is system software that manages hardware resources and provides services to applications. It acts as an intermediary between users and computer hardware.

Functions of an OS:

-Process Management

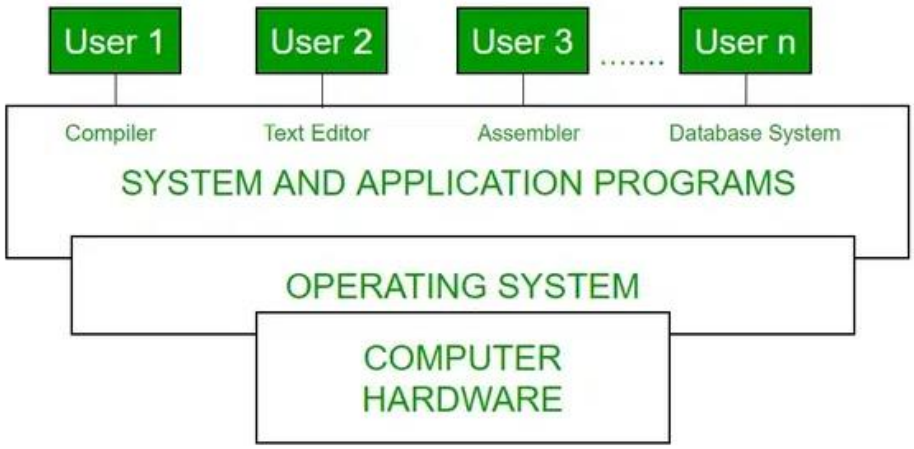
-Memory Management

-File System Management

-Device Management

-Security and Protection

-User Interface (CLI or GUI)



#History of Operating Systems (Brief)

1940s–1950s: No operating systems. Programs ran directly on hardware.

1960s: Batch processing systems (e.g., IBM OS/360).

1970s: Time-sharing and multi-user systems (e.g., UNIX).

1980s: Personal computer OS (MS-DOS, Mac OS).

1990s–2000s: GUI-based OS (Windows 95, Linux).

Present: Mobile and embedded OS (Android, iOS), cloud OS, virtualization, real-time systems.

#System Components / Services

Major Components:

-Kernel: Core of OS, manages system resources.

-Process Manager: Handles process creation, scheduling, and termination.

-Memory Manager: Allocates and deallocates memory to processes.

-File System Manager: Organizes, stores, and retrieves files.

-Device Manager: Manages input/output devices.

-User Interface: CLI or GUI.

System Services Include:

-Program execution

-I/O operations

-File manipulation

-Communication

-Error detection

-Resource allocation

#Interrupts & System Calls

Interrupts:

-Hardware or software signals that temporarily halt the CPU's current activity.

-Types: Hardware interrupts (e.g., keyboard input), Software interrupts (e.g., exceptions).

System Calls:

-Interface for user applications to request services from the OS kernel.

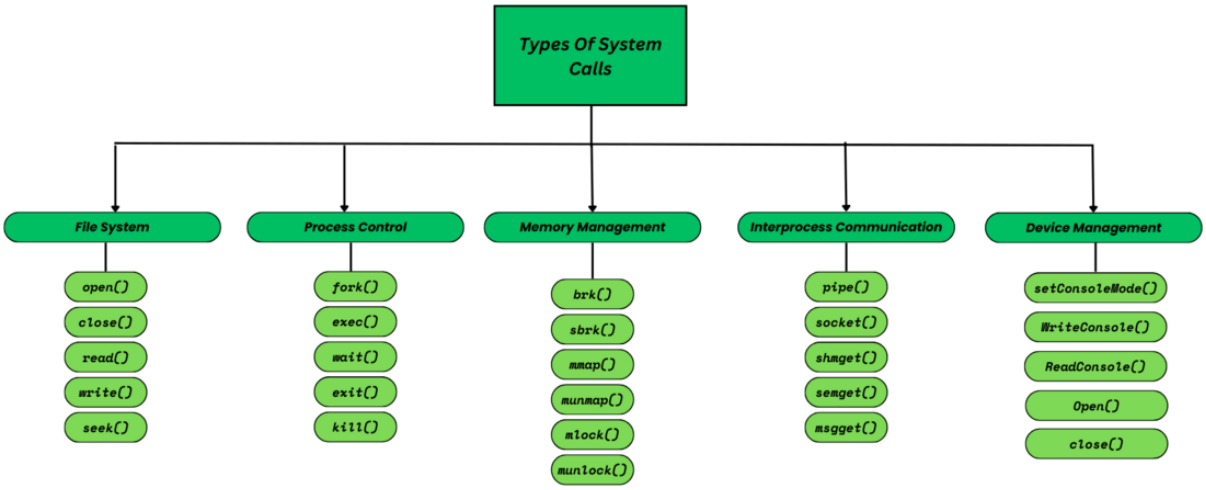
-Categories:

Process control (e.g., fork(), exec())

File operations (e.g., open(), read())

Device management

Communication (e.g., pipe(), socket())



#Introduction to Process Management

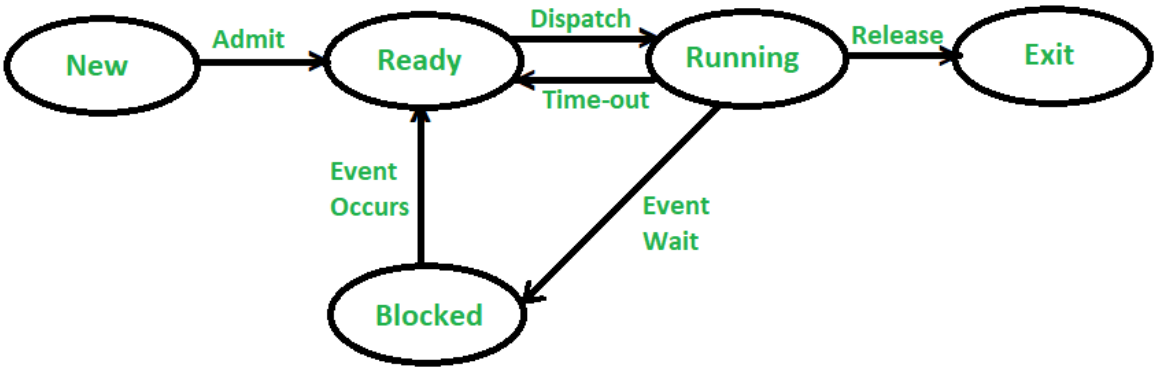
A process is a program in execution, including its current activity, variables, and program counter.

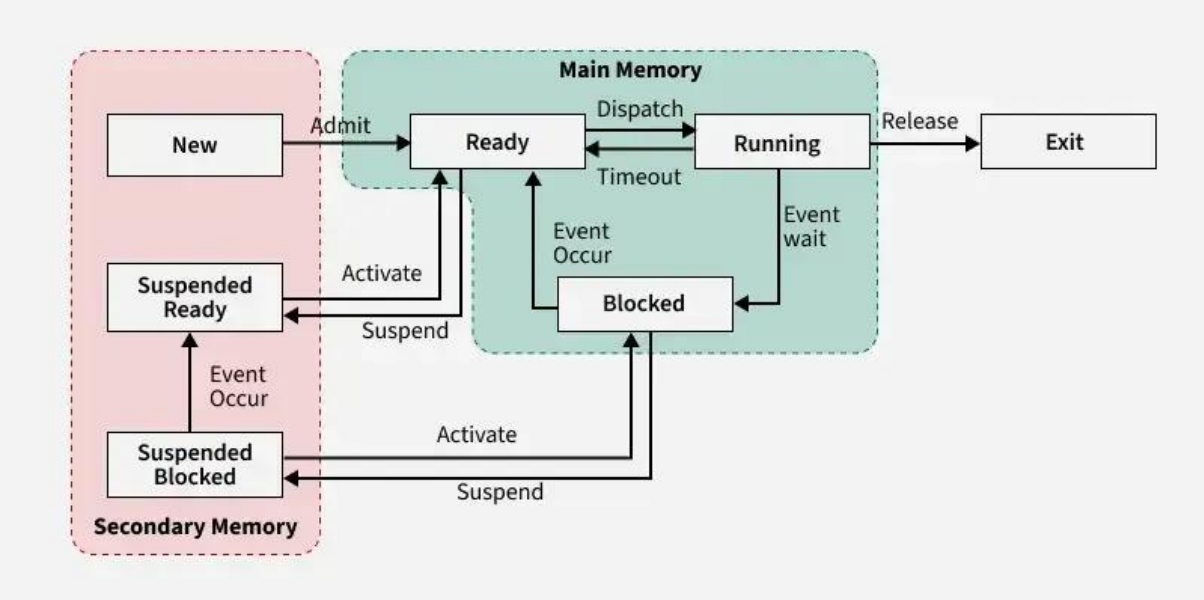
Key concepts:

-OS schedules processes to ensure efficient CPU usage.

-Supports multitasking and process isolation.

-Involves creating, scheduling, suspending, and terminating processes.





#Process States and Life Cycle

Common Process States:

New – Process is being created.

Ready – Waiting to be assigned to a processor.

Running – Instructions are being executed.

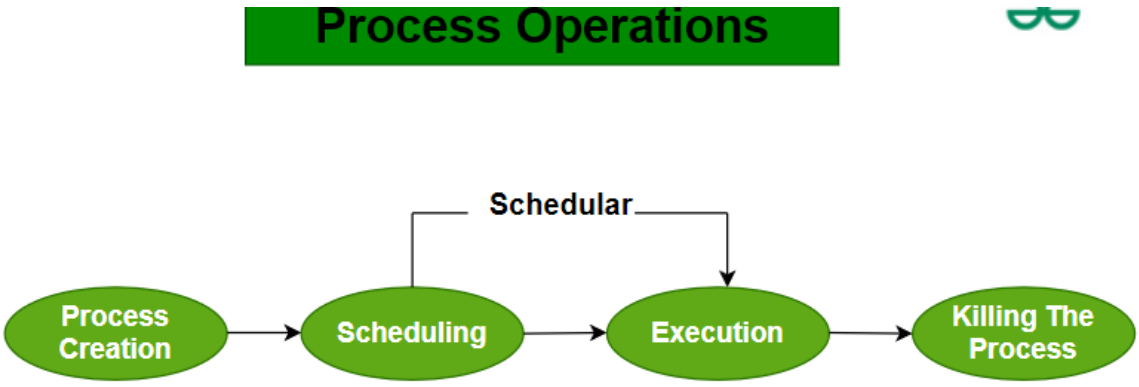
Waiting/Blocked – Waiting for I/O or event.

Terminated – Process has finished execution.

Lifecycle:

New → Ready → Running → (Waiting/Ready) → Terminated

State transitions are managed by the scheduler and dispatcher.



#Multithreading

Multithreading is the ability of a CPU or a single core to manage multiple threads of execution within a single process.

Benefits:

-Better resource utilization

-Increased performance

-Improved responsiveness

Threads share:

-Code

-Data

-Open files

Each thread has:

-Its own stack

-Its own program counter

-Its own registers

Types:

User-level threads

Kernel-level threads

#Basic Concepts

In OS, process scheduling and synchronization are critical for managing CPU usage and maintaining consistency when multiple processes or threads access shared resources.

#Scheduling Criteria

Scheduling criteria define how the OS evaluates scheduling algorithms:

-CPU Utilization: Keep CPU as busy as possible.

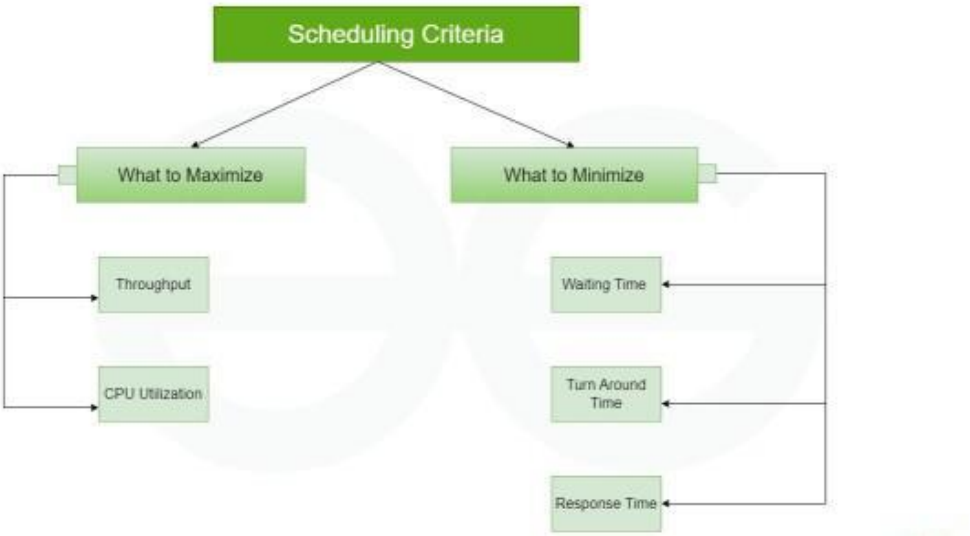
-Throughput: Number of processes completed per time unit.

-Turnaround Time: Time taken from submission to completion.

-Waiting Time: Total time a process waits in the ready queue.

-Response Time: Time from request submission to first response.

-Fairness: Equal CPU time allocation to all processes.



#3. Scheduling Algorithms

Scheduling algorithms decide the order in which processes run on the CPU:

-FCFS (First Come First Serve): Non-preemptive, simple but can cause long waiting time.

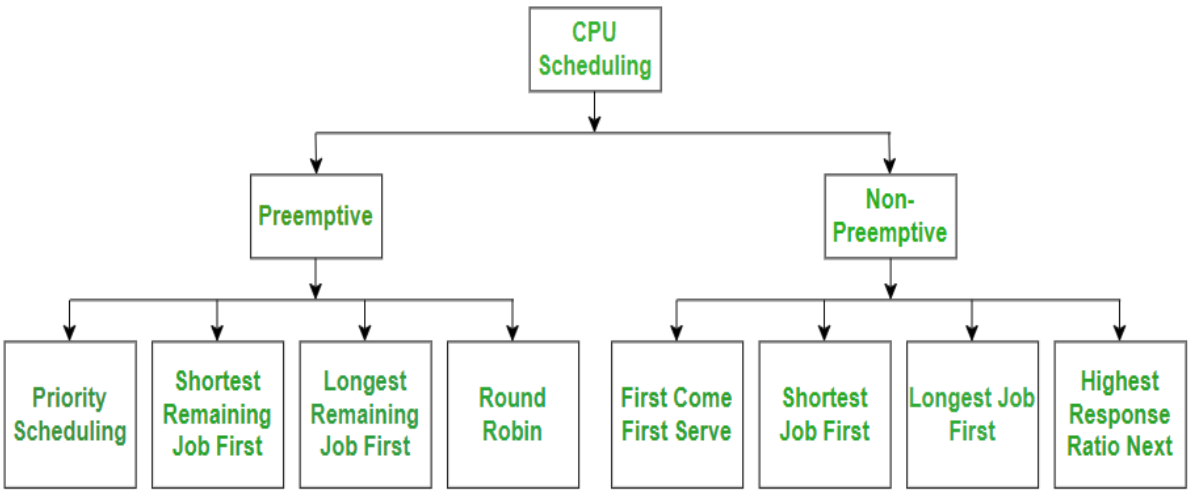
-SJF (Shortest Job First): Optimal for turnaround time; may cause starvation.

-Round Robin (RR): Preemptive, uses time quantum for fair CPU distribution.

-Priority Scheduling: Processes scheduled based on priority; lower-priority processes may starve.

-Multilevel Queue: Separate queues for different types of processes.

-Multilevel Feedback Queue: Allows processes to move between queues for better flexibility.



#Linux Scheduling Policies

Linux uses a completely fair scheduler (CFS) for normal tasks and real-time scheduling for time-sensitive tasks.

Normal vs Real-Time Scheduling

a)Normal Scheduling (CFS):

-Uses virtual runtime.

-Suitable for general-purpose user applications.

-Scheduling class: SCHED_NORMAL

b)Real-Time Scheduling:

-Uses fixed priorities.

-Minimal latency, guaranteed CPU access.

-Scheduling classes:

SCHED_FIFO (First-in-first-out)

SCHED_RR (Round Robin with priority)

Priorities in Linux

Nice value (-20 to 19): For normal scheduling; lower value = higher priority.

Real-time priorities (1 to 99): Higher number = higher priority.

Real-time tasks override normal tasks.

#Critical-Section Problem

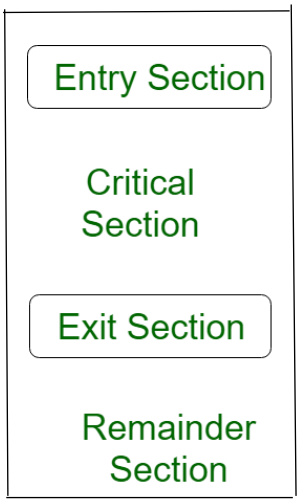
When multiple processes access shared data, the critical section is the portion of code that must not be executed by more than one process at a time.

Requirements for solution:

-Mutual Exclusion: Only one process in the critical section.

-Progress: No indefinite postponement.

-Bounded Waiting: Each process gets a chance eventually.



#Critical Region

A Critical Region is a code block where shared resources are accessed. It should be safely locked and unlocked to avoid race conditions.

OS provides primitives to protect critical regions: semaphores, mutexes, monitors.

#Semaphores & Mutex

a)Semaphore:

-An integer variable used to control access to shared resources.

-Two types:

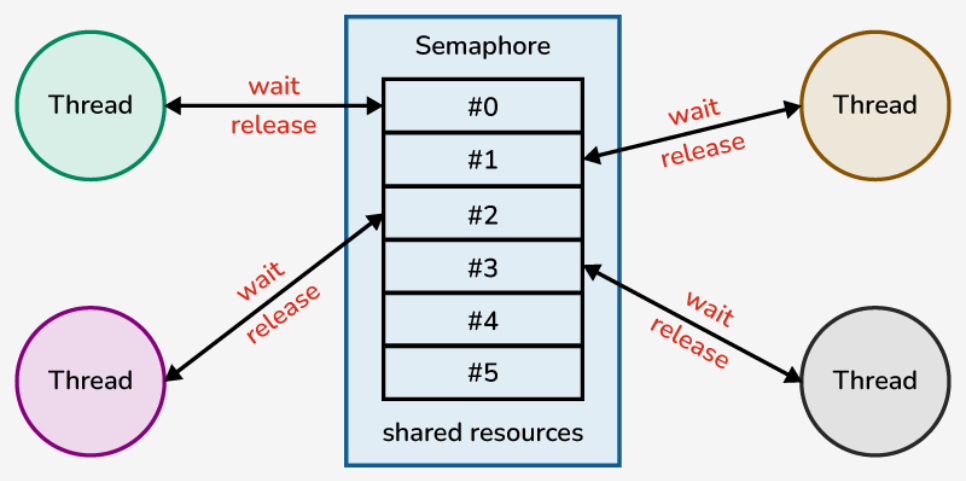
Counting Semaphore: Allows multiple access (e.g., 5 printers).

Binary Semaphore: Acts like a lock (0 or 1).

-Operations:

wait(S) / P(S) → decrements and may block

signal(S) / V(S) → increments and may unblock



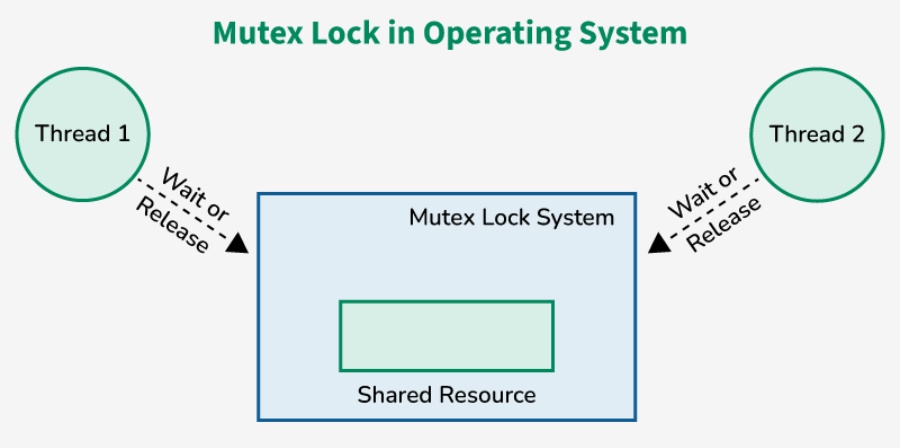
b)Mutex:

-Short for “mutual exclusion”.

-Binary locking mechanism.

-Only one thread can lock a mutex at a time.

-Used for thread-level synchronization.



#Producer-Consumer Problem

A classic synchronization problem:

-Producer adds items to a buffer.

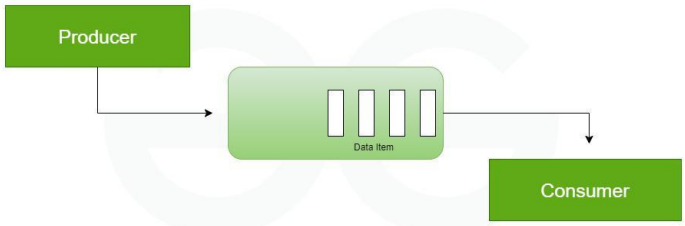
-Consumer removes items from the buffer.

Challenges:

-Prevent race conditions.

-Avoid buffer overflow and underflow.

Solution: Use semaphores or mutexes with condition variables to synchronize producer and consumer access.



#Monitors

A monitor is a high-level synchronization construct that:

-Encapsulates shared data, procedures, and synchronization.

-Allows only one process in the monitor at a time.

-Uses condition variables with wait() and signal() operations.

Monitors simplify complex thread synchronization problems like bounded-buffer and readers-writers.

#Deadlocks

Definition:

A deadlock is a situation in an operating system where a set of processes are blocked because each process is holding a resource and waiting for another resource held by another process in the same set. None of the processes can proceed, causing a standstill.

Necessary Conditions for Deadlock (Coffman Conditions):

1)Mutual Exclusion:

At least one resource must be held in a non-sharable mode (only one process can use it at a time).

2)Hold and Wait:

A process holding at least one resource is waiting to acquire additional resources held by other processes.

3)No Preemption:

Resources cannot be forcibly taken from a process; they must be released voluntarily.

4)Circular Wait:

Deadlock Handling Strategies

1)Deadlock Prevention:

Prevent one of the Coffman conditions to ensure deadlock never occurs (e.g., disallow circular wait).

2)Deadlock Avoidance:

The system dynamically checks resource allocation to avoid unsafe states (e.g., Banker's Algorithm).

3)Deadlock Detection and Recovery:

Allow deadlocks to occur, detect them using algorithms, and recover by terminating or rolling back processes.

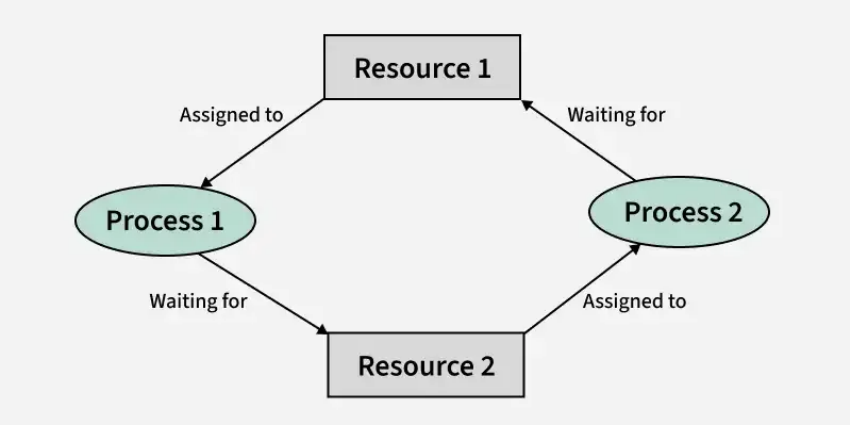
4)Ignore Deadlocks:

Some systems (like Linux) do not actively prevent or detect deadlocks; they rely on programmer care.

Deadlock in Linux

-Linux generally uses deadlock avoidance techniques internally and offers synchronization primitives (mutexes, semaphores) that programmers must use carefully.

-Kernel handles some deadlocks, but programmer attention is critical to avoid user-space deadlocks.



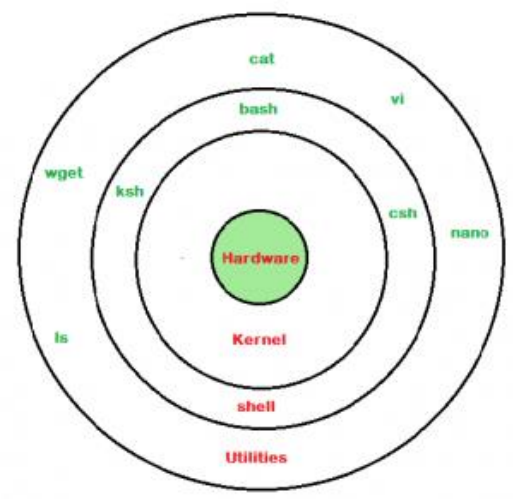
Section 7 and 8:

#Introduction to Shell

A shell is a command-line interface (CLI) that interacts with the operating system.

It interprets user commands and runs system programs.

Acts as both a command interpreter and a scripting environment.



#Types of Linux Shells

sh: Bourne shell (original Unix shell)

bash: Bourne Again Shell (improved version of sh)

csh: C Shell (C-like syntax)

ksh: Korn Shell (combines sh and csh features)

zsh: Extended Bourne shell with advanced features

#BASH (Bourne Again Shell)

Most commonly used Linux shell.

Supports scripting, command history, job control, and command-line editing.

Filename completion, arithmetic, and shell functions are supported.

#Shell Variables

a)Environment variables: System-wide variables (e.g., PATH, HOME)

b)User-defined variables: Created by users for scripts or sessions

Example: name="Alice"

#Shell Files

.bashrc: Runs for each interactive non-login shell.

.bash_profile: Runs once at login (for login shells).

.profile: Legacy login shell config.

.bash_logout: Executes commands at logout.

#Positional Parameters

Used to access command-line arguments in a script:

$0 → Script name

$1, $2, ... → Arguments

$# → Total number of arguments

$@, $* → All arguments

#Wildcards (* and ?)

*: Matches zero or more characters (e.g., *.txt)

?: Matches exactly one character (e.g., file?.txt)

#Command-Line Arguments

Passed when executing a script:

./script.sh arg1 arg2

Inside script:

echo "First argument is $1"

#Arithmetic in Shell Scripts

Use expr, $(( )), or let for calculations:

sum=$((4 + 5))

 # read and echo Commands

read takes input from user:

read name

echo displays output:

echo "Hello, $name"

 # tput Command

Used for terminal control like setting text colors or cursor positions:

tput bold

tput setaf 2 # green text

#Conditional Statements

a)if-then-fi

```
if [ $a -gt 5 ]; then
 echo "Greater"
fi
```

b)if-then-else-fi

```
if [ $a -gt 5 ]; then
 echo "Greater"
else
 echo "Smaller"
fi
```

c)test command: Checks conditions

File: -f file.txt, -d dir/

String: -z str, str1 = str2

d)Nested if-else: if inside another if

e)case (like switch-case)

```
case $1 in
 1) echo "One" ;;
 2) echo "Two" ;;
esac
```

 # Loop Structures

a)while loop

```
while [ $a -lt 10 ]; do
  echo $a
  a=$((a+1))
done
```

b)until loop: opposite of while

```
until [ $a -ge 10 ]; do
  echo $a
  a=$((a+1))
done
```

c)for loop

```
for i in 1 2 3; do
  echo $i
done
```

d)break/continue

```
if [ $i -eq 3 ]; then break; fi
```

#Shell Metacharacters

Special characters interpreted by the shell:

*, ?, >, <, |, &, ;, (), {}

#Command-Line Expansion

Tilde (~): Home directory

Variable Expansion: $USER

Command substitution: `date` or $(date)

#Directory Stack Manipulation

Commands:

a)pushd dir/: Save and go to directory

b)popd: Return to previous directory

c)dirs: Show directory stack

#Job Control, History, and Processes

&: Run in background

jobs: List background jobs

fg / bg: Bring job to foreground/background

ps, top: Show running processes

history: Show command history

kill PID: Terminate process

#Built-ins and Functions

a)Built-ins: Shell-internal commands (e.g., cd, echo, read)

b)Functions:

```
greet() {
 echo "Hello $1"
}
greet "World"
```
