# OS

**📌 What is an OS?**

An **Operating System** is system software that:

* Acts as an **intermediary** between user and hardware.
* Manages **hardware**, **software**, and **resources** (CPU, memory, I/O).
* Examples: Windows, Linux, Android, iOS.

**🔧 Why OS is hardware dependent?**

* OS interacts directly with the hardware using **drivers**.
* Each OS needs to be designed to handle **specific processor architecture** (e.g., x86, ARM).

**⚙️ Components of an OS**

**1 Kernel:**

* The core of the OS. It's the first part of the OS loaded into memory during boot-up.
* Responsible for the most critical functions: managing processes, memory, and hardware devices.
* Acts as the bridge between hardware and applications.

**Shell/User Interface (UI):**

* This is how users interact with the OS.
* **Graphical User Interface (GUI):** (e.g., Windows Desktop, macOS Finder, Android/iOS home screen) Provides icons, windows, menus, and pointers. Most common for typical users.
* **Command Line Interface (CLI):** (e.g., Linux Bash, Windows Command Prompt) Allows users to type commands to interact with the OS. Often preferred by developers and system administrators for automation and precise control.

1. **File System** – Manages data storage and retrieval. Examples: NTFS (Windows), HFS+/APFS (macOS), ext4 (Linux).
2. **Device Drivers** – Help OS communicate with hardware.
3. **System Utilities** – Basic tools (e.g., file manager).

**🖥️ Basic Computer Organization for OS**

* CPU: Executes instructions.
* RAM: Stores temporary data.
* I/O devices: Keyboard, mouse, etc.
* Bus: Connects all components.

**📱 Types of OS**

| **Type** | **Examples** | **Key Features** |
| --- | --- | --- |
| Mobile OS | Android, iOS | Touch-friendly, energy-efficient |
| Embedded OS | RTOS, VxWorks | Runs on specific hardware, real-time |
| Real-Time OS (RTOS) | QNX, FreeRTOS | Deterministic response, used in robots |
| Desktop OS | Windows, macOS | GUI-heavy, multitasking |
| Server OS | Ubuntu Server, RHEL | Designed for handling multiple users |

**🛠️ Functions of OS**

1. **Process Management**
2. **Memory Management**
3. **File System Management**
4. **Device Management**
5. **Security and Access Control**
6. **User Interface**

**🧍‍♂️ User and Kernel Mode**

* **User Mode**: Restricted access (apps run here).
* **Kernel Mode**: Full access (OS runs here).
* OS switches to kernel mode for critical tasks.

means

|  |  |
| --- | --- |
| /bin | Essential binaries |

|  |  |
| --- | --- |
| /etc | Configuration files |

ls # List files

cd # Change directory

pwd # Show current directory

mkdir # Create directory

touch # Create file

rm # Remove file/directory

cp # Copy files

mv # Move/rename

cat # Display file contents

echo $PS1 # Show primary prompt symbol

🐚 **Shell Programming Intro**

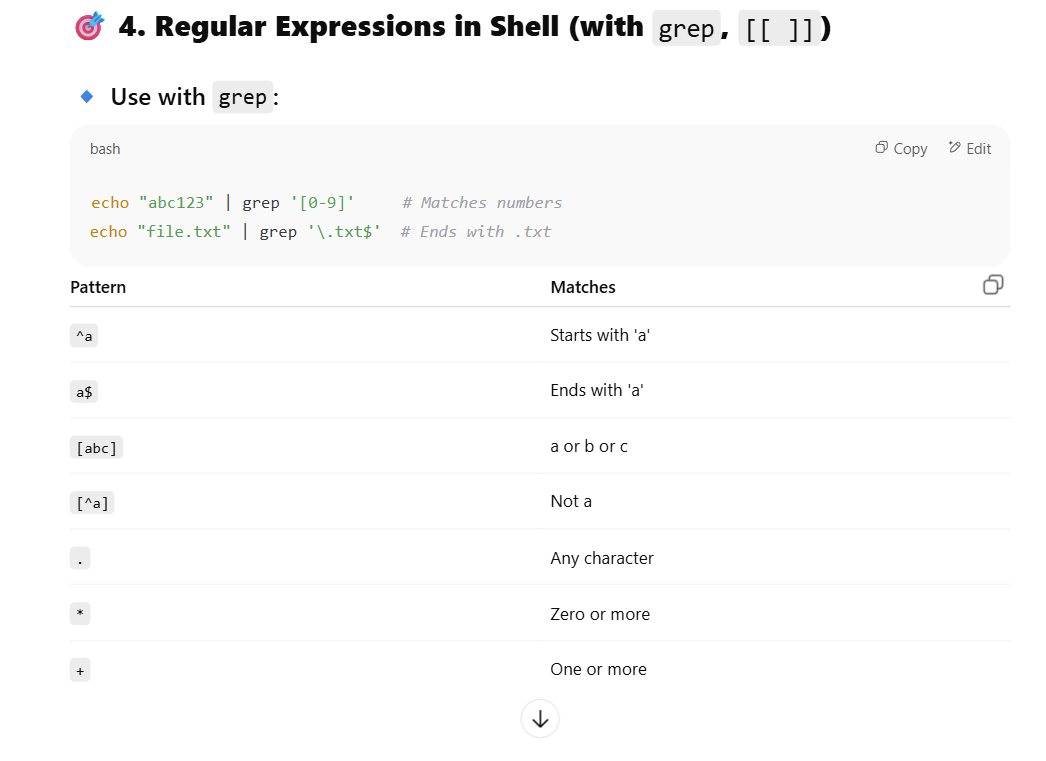
**What is Shell?**

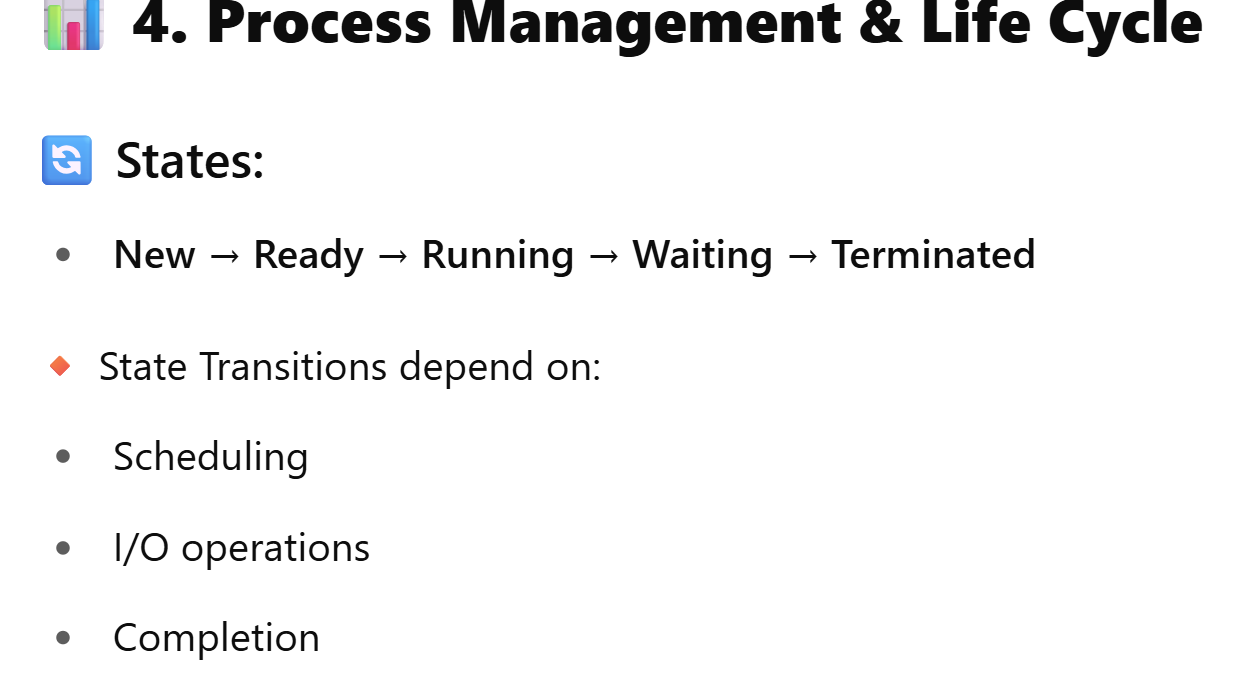
* A **command-line interpreter** that lets you interact with the OS.
* Types of shells:
  + sh – Bourne shell
  + bash – Bourne Again Shell (most used)
  + zsh, ksh, csh – Other shells

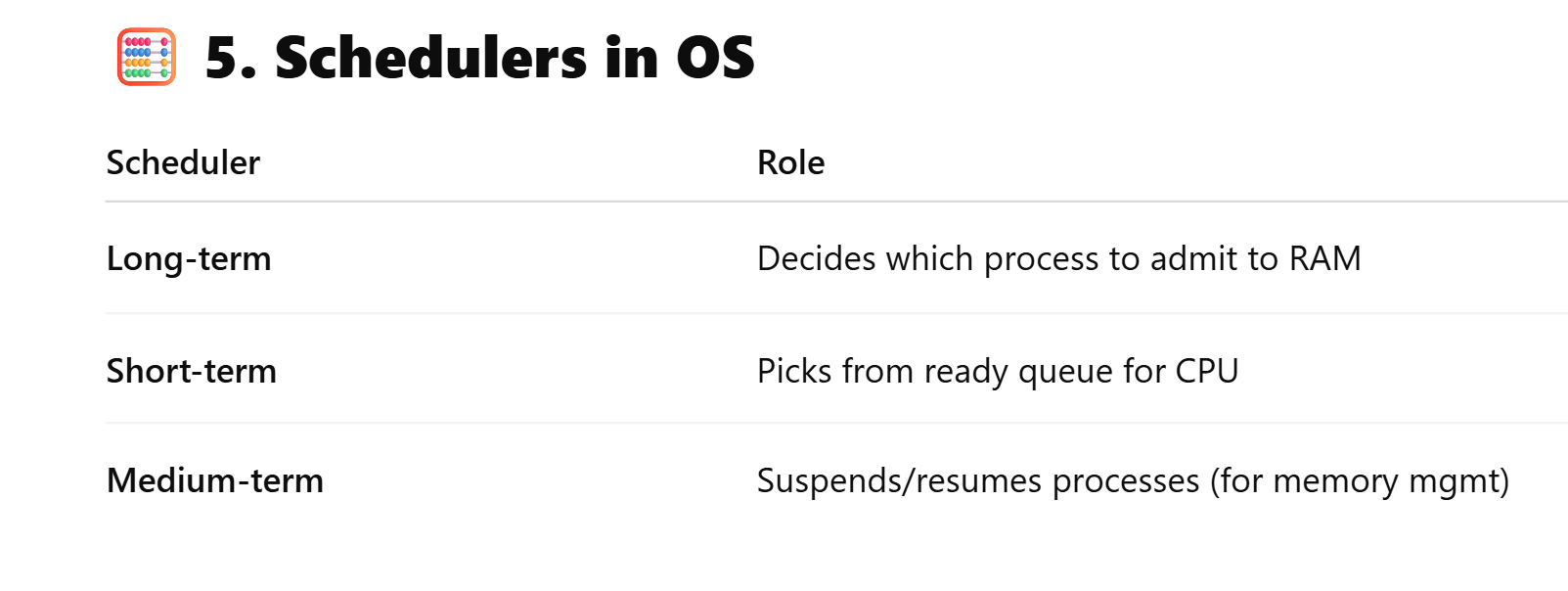
**💬 Command Line Arguments**

* $0 = script name
* $1, $2 = 1st, 2nd arguments
* $# = Number of arguments
* $@ = All arguments





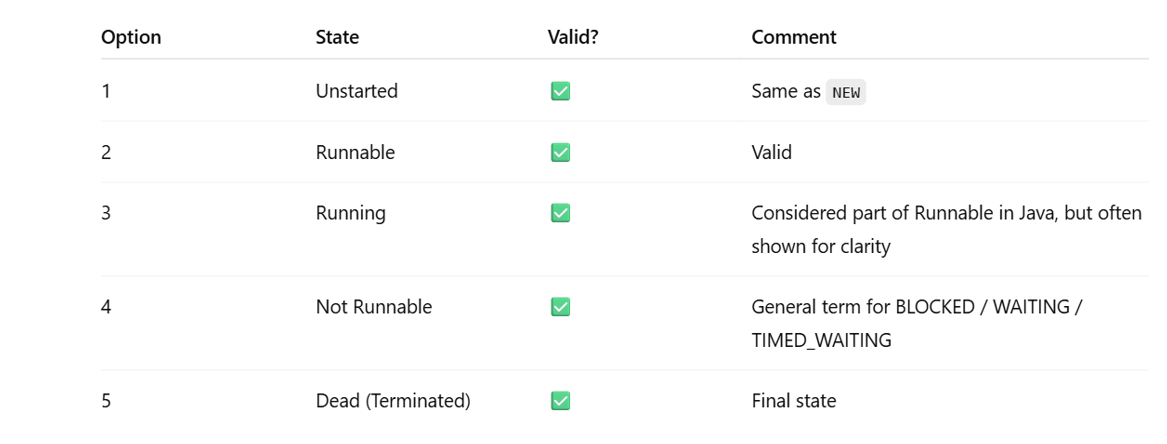




**Transitions:**

* New -> Ready: OS admits the process to the ready queue.
* Ready -> Running: Scheduler dispatches the process to the CPU.
* Running -> Ready: Time slice expires (preemption), higher priority process arrives.
* Running -> Waiting: Process requests an I/O operation or waits for an event.
* Waiting -> Ready: The event the process was waiting for occurs.
* Running -> Terminated: Process finishes execution or is explicitly killed.

| **Algorithm** | **Uses Preemption?** | **Common Transitions Used** |
| --- | --- | --- |
| FCFS | ❌ No | Ready→Running, Running→Waiting, Running→Terminated |
| SJF (non-preemptive) | ❌ No | Ready→Running, Running→Waiting |
| SJF (preemptive) | ✅ Yes | Running→Ready (shorter job arrives) |
| **Round Robin** | ✅ Yes | **Running→Ready (time quantum ends)** |
| Priority (preemptive) | ✅ Yes | Running→Ready (higher-priority arrives) |
| Multilevel Feedback | ✅ Yes | Running→Ready (based on queue feedback logic) |



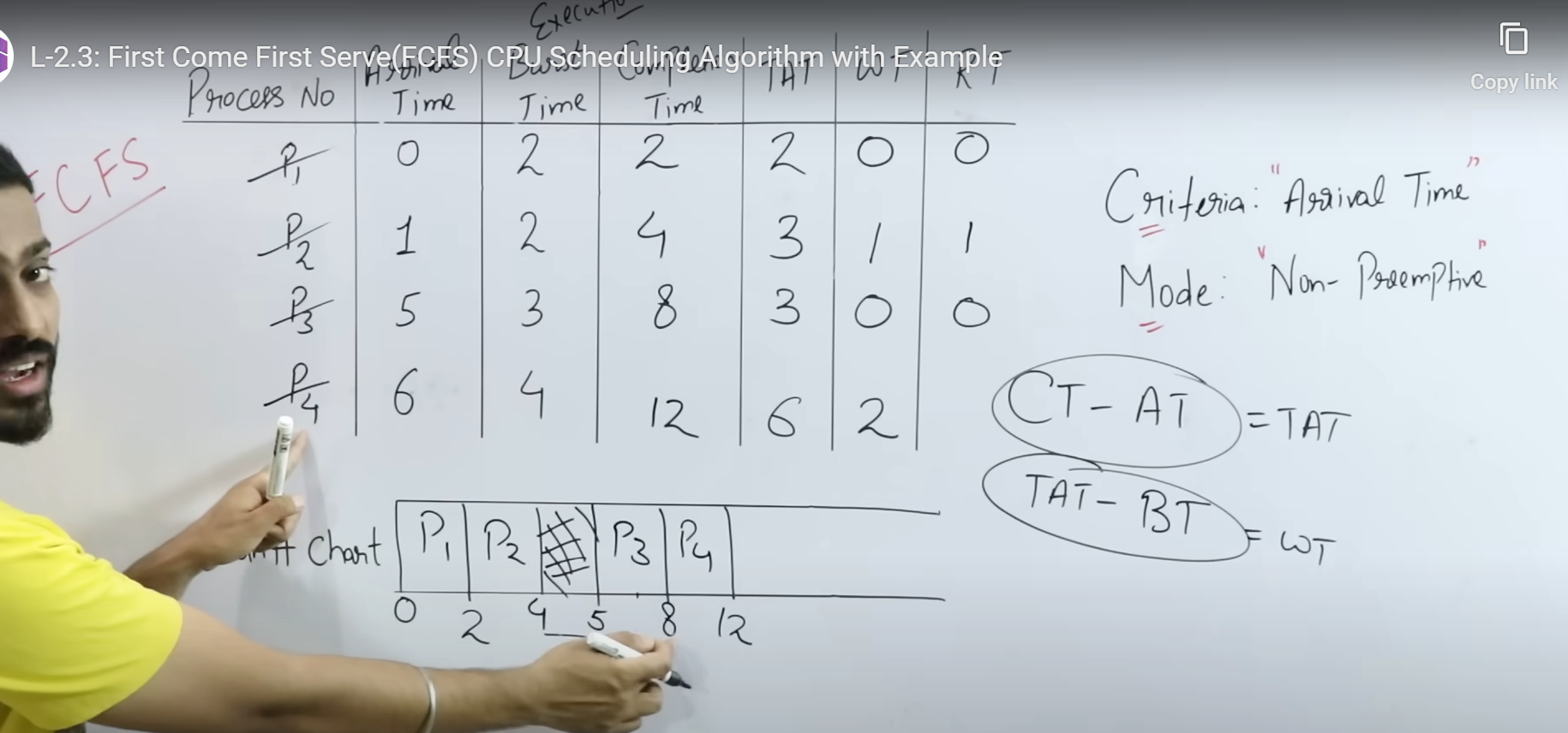
🧾 **6. Process Scheduling Algorithms**

🔹 FCFS (First Come First Serve)

🡺Non-preemptive🡺**he waiting time of the first process is:**🡺 0

🡺 Simple but can lead to **high average waiting time**.

 **Long processes can block short ones**, causing what's known as the **convoy effect**.



**🔹 SJF (Shortest Job First)**

* Executes shortest process first 🡺 We check the burst time jiska kam hoga usko ple run kre ge
* Low average wait time🡺

 Picks the **process with the smallest burst time first**.

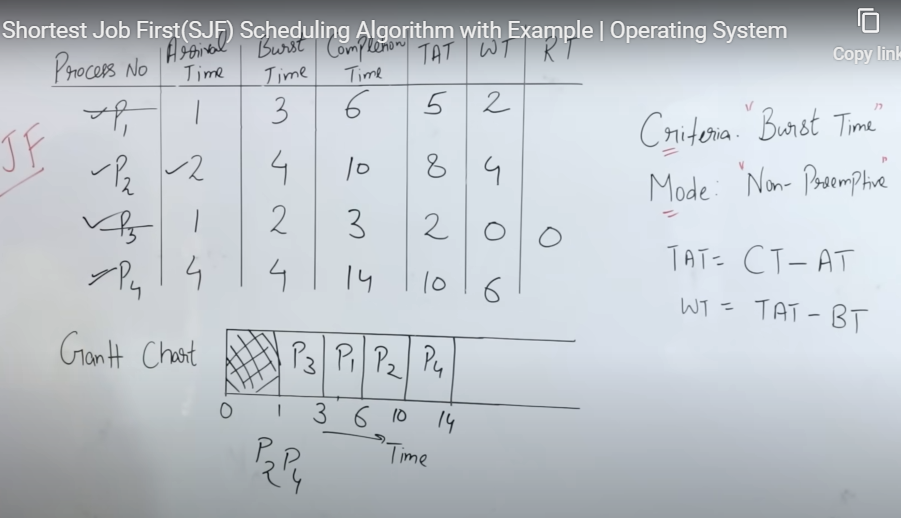
 This reduces the waiting time for short processes, hence **minimizing the average**.

 It's **provably optimal** in terms of average waiting time.

 **Limitation**: It needs **exact knowledge of burst times**, which is not always possible

Can cause **starvation(Shortest Job First can cause starvation if short processes keep arriving, preventing long ones from executing.)**

**Non Primitive == Ek bar cpu mil gya to process pura chele ga**



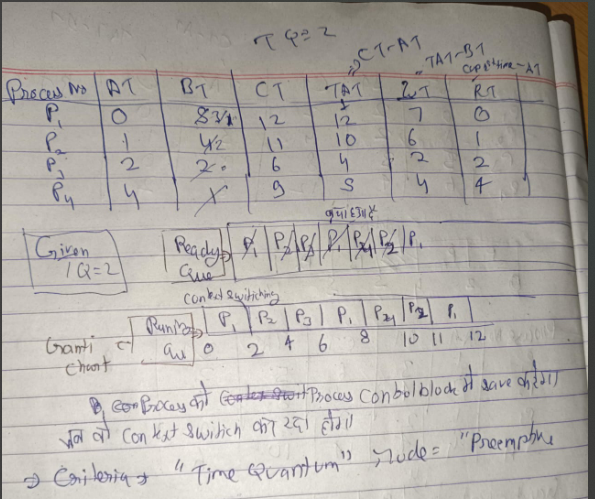
**Primitive 🡺 certain time (1 time ke ) ke baad check kre ga ki koe chota process ya less brust time hai kya hai to usko run kre ga 🡺ess case me Response time nikalte hai**

**🔹 Priority Scheduling**

* Higher priority first
* May be **preemptive** or not
*  Depends on **assigned priorities**.
*  Can result in **starvation** (low-priority processes waiting indefinitely) unless aging is used.

**🔹 RR (Round Robin) always preemptive**

* Fixed **time quantum🡺*In RR scheduling, each process is given a fixed amount of CPU time known as the time quantum.==>prevent from starvation***
* Fair in all scenarios but context-switching overhead🡺The number of context switches in RR depends on🡺Time quantum🡺If time quantum is too small in RR=More context switching overhead
* Round Robin is ideal for time-sharing as it shares CPU fairly.
* Fair, but has **higher average waiting time**, especially with many processes or small quantum.



**🔹 Multilevel Queue**

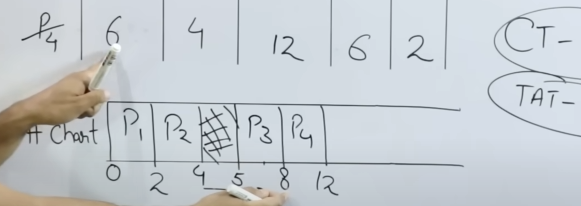
* Multiple queues for different types (foreground, background)
* Each queue has different policy

**Imp 🔁 7. Turnaround Time & Waiting Time**

**📐 Formulas:**

* **Turnaround Time** = Completion - Arrival
* **Waiting Time** = Turnaround – Burst
* **Completion Time = gaint chart me right hand vala kb process complete ho gya**

**Note 🡺Premitive ke case me Resposine time same hoga Wt ke Non prementive me different ho ga**

* **Response Time = Arrival time – Pahli bar cpu kb Allocate hu i.e**
* **Digram me P4 aaya 6 pr tha aur usko cpu 8 pr mila tha to RT🡺2 hua**
* 
* Left hand side vala dekhe ge vo bataye ga cpu kb mila

**👨‍👩‍👧‍👦 8. Process Creation**

**🔧 Important System Calls (in Linux):**

fork() – Creates a **new child process**🡺fork() is used in Unix/Linux systems to create a new child process by duplicating the current process. 🡺fork() is called once total processes are created is One parent and one child = 2 processes. 🡺fork() returns 0 in the child, and child PID in the parent.

* exec() – Replaces process memory with new program i,e Loads new program.( replaces the current process memory with a new program)
* waitpid() – Parent waits for child

## 🧟 **9. Orphan & Zombie Processes**

| **Type** | **Meaning** |
| --- | --- |
| **Orphan** | Parent terminated, child still running🡺A parent dies before child |
| **Zombie** | Child terminated, parent didn’t read status🡺*ps aux shows all processes, including zombies marked as <defunct>.*  Zombie have no active execution and hold no system resources except the PID. |

🡺**A terminated process still in process table**  
➡️ *Explanation: A zombie process has completed execution but still has an entry in the process table, as the parent hasn't read its status.*

**The parent of all orphan processes in Linux is:**  
A) systemd ✅

**Orphan Process:**

* **Definition:** A child process whose parent process has terminated *before* the child process.
* **What happens:** When the parent dies, the orphaned child process is immediately **adopted by the init process** (PID 1 on Linux/Unix systems). The init process then becomes its new parent and will handle its wait() call when the child eventually terminates.
* **Impact:** Generally harmless, as init is designed to handle this. The child continues to run normally.

**Zombie Process (Defunct Process):**

* **Definition:** A child process that has terminated its execution, but its entry (Process Control Block - PCB) still remains in the process table because its parent process has not yet called wait() or waitpid() to retrieve its exit status.
* **Characteristics:** It consumes no CPU time or memory (except for its PCB entry). It's essentially a "dead" process waiting to be "reaped."
* **Problem:** If many zombie processes accumulate without being reaped, they can fill up the process table, preventing new processes from being created. This is a resource leak.
* **Prevention:** Always use wait() or waitpid() in the parent process to collect the child's exit status.
* **Solution:** The only way to get rid of a zombie process is to kill its parent, which will then cause the init process to adopt the zombie and immediately wait() on it, cleaning it up.

MCQ Points

🡺*The short-term scheduler (CPU scheduler) selects processes from the ready queue and allocates CPU time.*

🡺 **Long-term Scheduler**:  
Decides **which jobs are admitted** into the system for processing. Controls the **degree of multiprogramming** (how many processes are in memory).==> *I/O is handled by the OS, not the scheduler.*

🡺 **Medium-term Scheduler**:  
Handles **suspending and resuming processes**. Used in swapping systems to remove processes from memory to reduce load.

🡺**Bootloader**:  
Not a scheduler. It loads the **operating system into memory** when the system starts.

🡺After I/O, the process goes to the ready queue, waiting to be scheduled for CPU.

🡺Context switch saves the state of the old process and loads the state of the new one.

🡺Ready queue holds all processes that are ready and waiting for CPU.🡺

🡺*Multilevel queues divide processes based on priority/class (foreground, background, etc).*

🡺In suspended state, a process is swapped out of memory temporarily.

🡺**29. Which system call allows a parent to wait for child to finish?**  
A) waitpid() ✅  
B) exec()  
C) fork()  
D) pause()

➡️ *Explanation: waitpid() and wait() are used to wait for child processes.*

🡺context switching time considered as🡺Overhead

🡺**The state of a process performing I/O is:**  
A) Ready  
B) Running  
C) Waiting ✅  
D) Terminated

🡺**Which of the following is not part of process control block (PCB)?**  
A) Program counter  
B) Process ID  
C) Page table  
D) GUI components ✅

# Unit 5: Memory Management

**🔍 1. Why Memory Management?**

* The **main memory (RAM)** is limited.
* The OS needs to **efficiently allocate and deallocate memory** to multiple running processes.
* Memory management ensures:
  + **Process isolation** (security)
  + **Efficient usage** of RAM
  + **Process execution** without failure

**🧠 OS Responsibilities:**

* Keep track of **which parts of memory are in use**
* **Allocate memory** to processes
* **Deallocate memory** when not needed
* Prevent one process from **accessing another’s memory**

**🧱 2. Types of Memory**

| **Memory Type** | **Description** |
| --- | --- |
| **RAM (Main Memory)** | Temporary memory used by active programs |
| **Cache** | High-speed memory close to CPU and stores frequently used data. |
| **Registers** | Smallest, fastest memory inside CPU |
| **Secondary Storage** | Permanent (SSD/HDD) |
| **Virtual Memory** | Uses disk space to extend RAM |

**🧩 3. Memory Allocation Techniques**

**📍 A. Contiguous Allocation**

* Each process is allocated **a single block of memory** in one piece.
* Very simple, but:
  + Can cause **fragmentation**
  + Not flexible

🔷 Imagine memory like a hotel floor. If each guest (process) needs continuous rooms, it may be hard to find such space even if total space is available.

**📍 B. Dynamic Partitioning (Variable Sized)**

* Memory is allocated dynamically based on process size.
* Can still cause **external fragmentation** (gaps between allocated memory).

**⚙️ 4. Allocation Strategies**

**➤ First Fit**

* Allocate the **first hole** large enough.
* **Fast**, but may leave many small unusable gaps.
* **In dynamic partitioning, which strategy picks the first suitable block for allocation?**  
   First Fit

**➤ Best Fit**

* Allocate the **smallest hole** that fits.
* **Minimizes waste** but slow (needs full scan).

**➤ Worst Fit**

* Allocate the **largest hole**.
* Can leave **large** remaining gaps (inefficient).

**🚫 5. Fragmentation**

**➤ Internal Fragmentation**

* Memory **allocated > requested**.
* Waste **inside** allocated block.

Example: Requested 29KB, allocated 32KB → 3KB wasted inside block.

**➤ External Fragmentation**

* Enough total memory is free, but not **contiguous**.
* Can’t fit large processes despite having space.

*Smaller pages reduce internal waste but increase page table size.*

**🔧 Compaction**

* OS tries to move processes together to **merge free memory blocks**.
* Time-consuming(CPU time) but reduces external fragmentation.
* What does memory compaction do? 🡺 Rearranges memory to reduce fragmentation

**🧠 6. Segmentation**

* Memory is divided into **logical segments**: code, stack, data, heap, etc.
* Each segment has:
  + **Base**: Starting physical address
  + **Limit**: Length of segment

**🧾 Segmentation Table:**

| **Segment** | **Base** | **Limit** |
| --- | --- | --- |
| Code | 1000 | 500 |
| Stack | 2000 | 300 |
| Data | 3000 | 400 |

**➤ Logical Address: (Segment number, offset) 🡺** **is converted to physical address using: Base + Offset**

Example: (1, 100) = Base of segment 1 + 100 = 2000 + 100 = 2100

🧠 **Advantage**: Allows logical separation and protection. 🡺 Provides process isolation  
🚫 **Disadvantage**: Still prone to fragmentation.

**📦 7. Paging**

***Paging allows processes to occupy non-contiguous memory blocks, reducing external fragmentation.***

**What is paging?**

* **Paging is a memory management scheme that divides logical memory into fixed-size pages and physical memory into fixed-size frames.**
* **A page is loaded into an available frame for execution.**
* 📘 *Logical address in paging = page number + offset within that page.*
* Physical Address = (Frame Number \* Frame Size) + Offset d

**It eliminates external Fragmentation but not internal**

**🔹 What is internal fragmentation?**

* **Internal fragmentation occurs when allocated memory may have unused space within it.**
* **In paging, if a process's last page doesn’t perfectly fill the frame, the leftover space in that frame is wasted.**

**📌 Example:**

* **Suppose page size = frame size = 4 KB.**
* **If a process needs 10 KB, it will require 3 pages:**
  + **1st page: 4 KB**
  + **2nd page: 4 KB**
  + **3rd page: only 2 KB used, 2 KB is wasted**
* **That 2 KB of unused space is internal fragmentation.**
* **Page table base register (PTBR)🡺** **holds the starting address of the page table for the current process.**
* *page table maps logical page numbers to physical frame numbers.i,e 🡺* *Page to frame mapping*

Disadvantage 🡺 *Fixed-size pages can waste space inside frames, causing internal fragmentation.*

**🚀 8. TLB (Translation Lookaside Buffer)**

* A **small cache** that stores recent page–frame mappings.
* Helps speed up memory access by avoiding full page table lookup.
* access time in a memory system with TLB🡺 TLB lookup time + memory access time TLB stores🡺 *It stores recent virtual-to-physical address translations🡺* High TLB hit ratio leads to🡺 reduces access time, improving speed.()

**🧽 9. Dirty Bit**

* Each page has a **dirty bit** that tells if the page was **modified**.
* If a dirty page is swapped out to disk, it must be **written back**.
* If clean, it can just be discarded.
* Dirty pages (modified) must be saved before removal.
* Dirty bit set values A) on, off  
  **Explanation:** “On” means modified, needs to be saved before replacement.
*  **How it works:**
* Initially set to 0 when a page is loaded from disk.
* Set to 1 by the hardware whenever any byte within that page is written to.
*  **Importance (especially for Page Replacement):**
* When the OS needs to select a page to evict from main memory (during a page fault and no free frame is available), it checks the dirty bit.
* If the dirty bit is 0 (clean page), the page has not been modified, and the copy on disk is still valid. The page can simply be overwritten in memory without writing it back to disk.
* If the dirty bit is 1 (dirty page), the page has been modified. The modified content *must* be written back to secondary storage (swapped out) before its frame can be reused for a new page. This saves a disk write operation for clean pages.

**🤝 10. Shared Pages & Reentrant Code**

* **Shared Pages**: Two or more processes use the **same code/page** to save memory.
* **Shared pages are useful when:**  
  A) Pages are read-only  
  B) Pages are for different users  
  C) Pages are for I/O  
  D) All pages are private

**Answer:** A  
📘 *Read-only code (like libraries) can be shared across processes safely.*

Example: Two browser tabs using the same shared rendering engine.

* **Reentrant Code**:
  + Also called **pure code**
  + Code that **doesn’t modify itself**
  + Safe to **share across processes**

**🚦 11. Throttling**

* Used when system is **overloaded**.
* OS may:
  + Pause or delay low-priority processes
  + Prevent new processes from starting
* Helps **control CPU/memory usage**
* **throttling** refers to intentionally slowing down or limiting the rate at which a process or resource can be used.

**📤 12. I/O Management (Quick Overview)**

* OS handles **input/output devices** (mouse, keyboard, disk, etc.)
* Includes:

**Spooling (Simultaneous Peripheral Operations On-Line):**

* **Concept:** Using a dedicated disk buffer to handle I/O for devices that cannot handle interleaved data streams.

**DMA (Direct Memory Access):**

* **Concept:** A hardware feature that allows I/O devices to transfer data directly to and from main memory without involving the CPU
  + **Buffering**: Temporarily hold data
  + **Spooling**: Queue of output jobs (e.g., print queue)
  + **Interrupt Handling**: Process notifies CPU when I/O is done

**✅ Summary Table**

| **Concept** | **Meaning** |
| --- | --- |
| Contiguous Alloc | Fixed blocks; simple but causes fragmentation |
| Segmentation | Logical memory division; base & limit used |
| Paging | Logical pages mapped to physical frames |
| Page Table | Holds mapping of pages to frames |
| TLB | Caches page-frame entries for faster access |
| Dirty Bit | Shows if page is modified |
| Reentrant Code | Read-only, shareable code between processes |
| Throttling | Slows down processes to control system load |
| Fragmentation | Wasted memory (internal/external) |
|  |  |

🡺   
📘 *A memory leak occurs when a program doesn't release unused memory*

🡺 *Intel architectures use segmentation + paging for flexible protection and performance.*

🡺 *RAM is volatile and loses data when power is off.*

# ✅ UNIT 6: VIRTUAL MEMORY

Multics was a pioneer in virtual memory concepts.

**🔍 1. What is Virtual Memory?**

Virtual Memory is a technique that allows **execution of processes that may not be completely in memory**. It gives the illusion that each process has **its own large, continuous block of memory** — even if actual RAM is limited.

**🧠 Purpose:**

* **Run large programs** without needing enough RAM
* **Multi-tasking** without overloading RAM
* **Efficient memory use** by loading only what's needed

**🡺** **Memory management unit (MMU) enables virtual memory**

**💡 2. How It Works**

Virtual memory works with:

* **Paging** or **Segmentation**
* **Page tables**
* **Disk storage (swap space)**

Only **active parts** of a program are loaded into memory. Others are kept on **disk** and brought into memory on demand.

📌 **Key Component:** **Demand Paging**

**📦 3. Demand Paging**

Pages are not loaded into memory until **they are needed**.

**Steps:**

1. CPU accesses a page.
2. If page is **not in memory**, it triggers a **page fault**.
3. OS loads the page from **disk (swap)** to **RAM**.
4. Page table is updated.’
5. Paging allows non-contiguous allocation, ideal for virtual memory.
6. Virtual memory maps🡺 Virtual to physical addresses(The MMU translates virtual addresses to physical ones.)

❗ Frequent page faults slow down the system (called **thrashing**).

📘 *Thrashing occurs when the system spends most time handling page faults.*

**Thrashing** occurs when a system spends **more time swapping pages in and out of memory** than executing actual processes — usually due to **overcommitment of memory**.

**Suspend low-priority processes** ✔️

* This is an effective way to **free up memory**.
* By suspending or swapping out **low-priority** or **inactive** processes, more RAM is available for active ones, reducing page faults and thrashing

**🧠 4. Page Fault**

A **page fault** happens when:

* A program tries to access a page that is not in physical memory (RAM).
* *OS is responsible for servicing page faults and managing memory.i,e 🡺* *Page faults are handled by os*
* Page fault service time is: Higher than memory access(*Fetching from disk is much slower than from RAM.*)

**OS must:**

* Pause the process
* Load the page from disk
* Update the page table
* Resume the process

🛑 Too many page faults can drastically reduce performance.

**🔁 5. Page Replacement Algorithms**

When RAM is full, and a new page needs to be loaded, **some page must be removed**.

**Common Algorithms:**

| **Algorithm** | **Description** |
| --- | --- |
| **FIFO** | Removes the oldest page (First In First Out) |
| **LRU** | Removes the **Least Recently Used** page implemented by using stack |
| **Optimal** | Removes the page that **won’t be used for the longest time** in future (not practical, but ideal) |
| **Clock** | Uses a reference bit (like circular FIFO) |
| **Second Chance** | Variation of FIFO that gives a page a second chance if its reference bit is set |
|  |  |

**23. Which algorithm gives best performance but is theoretical?**  
A) FIFO  
B) Optimal  
C) LRU  
D) Clock

**Answer:** B  
📘 *Optimal removes the page that won’t be used for the longest time — ideal but requires future knowledge.*

🡺 Belady’s anomaly is associated with in FIFO, increasing frames can sometimes increase page faults — known as Belady’s anomaly

**30. Second Chance avoids replacing:**  
A) Dirty pages  
B) Recently used pages  
C) Clean pages  
D) Large pages

**Answer:** B  
📘 *It checks if a page was used recently via its reference bit.*

**🧪 Example:**

Let’s say you have 3 frames and the page reference string is:  
**7, 0, 1, 2, 0, 3, 0, 4**

Using **FIFO**, the first pages loaded will be:

* 7, 0, 1 → 3 page faults
* Then 2 replaces 7 → page fault
* 0 is already present → no fault
* 3 replaces 0 → fault
* And so on…

You can calculate total faults and compare algorithms.

**⚠️ 6. Thrashing**

When a system spends **more time swapping pages** than executing processes, it is called **thrashing**.

**Causes:**

* Too many processes
* Too little memory
* Bad page replacement strategy

**Solutions:**

* Reduce number of processes
* Use **working set model**
* Use smarter replacement algorithms

**📚 7. Advantages of Virtual Memory**

* **Enables multitasking**
* **Run large programs** on small RAM
* **Efficient use of memory**
* Process isolation and security
* Faster context switching

**🚫 8. Disadvantages**

* **Slower** than using pure RAM (due to disk I/O)
* Risk of **thrashing**
* Requires complex OS and hardware support

**✅ Summary Table**

| **Concept** | **Explanation** |
| --- | --- |
| Virtual Memory | Illusion of larger memory than physical RAM |
| Demand Paging | Load pages only when needed |
| Page Fault | Page is not in memory; OS loads from disk |
| Page Replacement | Replace a page to make room for a new one |
| FIFO | Oldest page replaced |
| LRU | Least recently used page replaced |
| Optimal | Best theoretical performance |
| Thrashing | Excessive paging slows system |
| Working Set | Set of pages a process actively uses |

🡺 *the MMU converts virtual addresses to actual RAM addresses.*i,e Physical Address

🡺 Working set is the set of pages a process needs in a given time to avoid faults.

🡺 **32. What is the best way to avoid thrashing?**  
A) Reduce RAM  
B) Run more processes  
C) Increase multiprogramming  
D) Adjust working set size

**Answer:** D  
📘 *Keeping the working set in memory prevents frequent page faults>*Excessive paging causes CPU to wait, lowering utilization.

🡺 Excessive paging causes CPU to wait, lowering utilization.

🡺 *Local replacement limits a process to replace its own pages, avoiding system-wide faults.*

🡺 Swap space is disk space used when RAM is full.

# ✅ UNIT 7: DEADLOCKS

**🔍 1. What is a Deadlock?**

A **deadlock** is a situation in a **multiprogramming system** where **two or more processes are waiting indefinitely for resources** held by each other, and **none can proceed**.

The deadlock handling method used by Linux is: 🡺 Ignoring the problem(Linux typically ignores deadlocks (ostrich algorithm))

🡺 Deadlock is a type of Process synchronization problem🡺 *Deadlock is related to resource synchronization among processes.*

**🧷 2. Real-Life Analogy**

Imagine:

* Two people trying to cross a narrow bridge from opposite sides.
* Both are blocking each other.
* Neither can go forward or backward.

➡️ This is a **deadlock**.

**🔗 3. Necessary Conditions for Deadlock(Coffman Conditions)**

According to **Coffman Conditions**, a deadlock **can occur** if all **four** of these conditions hold **simultaneously**:

| **Condition** | **Description** |
| --- | --- |
| **Mutual Exclusion** | Only one process can use a resource at a time 🡺 hardest to eliminate |
| **Hold and Wait** | Process holds at least one resource and is waiting for others |
| **No Preemption** | A resource cannot be forcibly taken; only released voluntarily |
| **Circular Wait** | A circular chain exists where each process waits for a resource held by the next🡺 Happens only in multithreading |

🔁 **All four must be present** for deadlock to occur.

Allowing all resource requests upfront prevents:🡺 Hold and wait

**⚠️ 4. Deadlock vs Starvation**

| **Deadlock** | **Starvation** |
| --- | --- |
| All processes blocked forever | Some processes never get the resource |
| Caused by circular wait | Caused by unfair scheduling |
| No process progresses | Some processes may still progress |

**🛡️ 5. Deadlock Handling Strategies**

1. **Deadlock Prevention**
   * Ensure **one of the four conditions never holds**.
   * Example: Prevent **hold and wait** by requiring all resources to be requested at once.
2. **Deadlock Avoidance**
   * Use algorithms like **Banker's Algorithm**.
   * Requires knowledge of **maximum resource needs in advance**.
   * System checks **before allocating** a resource to ensure safety.
3. **Deadlock Detection and Recovery**
   * Allow deadlocks but detect and **recover**.
   * Use **resource allocation graphs** or **detection algorithms**.
   * Recovery: **Abort** or **rollback** a process.
4. **Ignore the Problem**
   * Used in many operating systems like **Linux** ("Ostrich Algorithm").
   * Assumes deadlocks are **rare** and ignores them.

**💳 6. Banker's Algorithm (Avoidance Strategy)**

* **Similar to a bank** giving out loans — ensures it never gets into a state where it can’t fulfill future requests.
* Works only if:
  + The system knows **maximum resource needs**
  + Resources are allocated only if the system stays in a **safe state**

**Safe State:**

* A state where **all processes can finish** if executed in some order.

📌 If a request leads to an unsafe state, **it is denied**.

🡺 A safe sequence of process execution is used to confirm safe state.

*Unsafe does not mean deadlock, but it’s a warning that deadlock* ***can*** *occur.*

In Banker’s algorithm, what are the key data structures🡺 Allocation, Max, Available, Need matrices

**⚙️ 7. Resource Allocation Graph (RAG)**

* Graph of processes (circles) and resources (squares)
* Edge types:
  + **Request edge:** P → R
  + **Assignment edge:** R → P

**Cycle in RAG:**

* With **1 instance per resource** → cycle means **deadlock**
* With **multiple instances** → cycle **may or may not** mean deadlock

**🔁 8. Recovery from Deadlock**

1. **Process Termination**:
   * Abort all deadlocked processes
   * Abort one at a time until deadlock breaks
2. **Resource Preemption**:
   * Temporarily take resources from processes

⚠️ Care must be taken to avoid **starvation** during recovery.

**✅ Summary Table**

| **Concept** | **Description** |
| --- | --- |
| Deadlock | Processes waiting indefinitely |
| 4 Coffman Conditions | Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait |
| Prevention | Deny at least one condition |
| Avoidance | Predict and avoid unsafe states |
| Detection | Let it happen and detect cycles |
| Recovery | Abort or preempt processes |
| Starvation | Process waits indefinitely due to unfairness |

🡺 *Priority inversion is a scheduling issue, not related to circular wait.*

**25. Which recovery strategy reclaims resources without killing processes?**  
A) Rollback  
B) Restart  
C) Suspend  
D) Preemption

**Answer:** D  
📘 *Preemption reclaims a resource by suspending the holding process.*

*Preemption must ensure the system stays in a safe state.*

🡺 **41. A wait-for graph is used to:**  
A) Allocate resources  
B) Replace Banker’s algorithm  
C) Detect cycles in deadlock detection  
D) Track CPU time

**Answer:** C  
📘 *Wait-for graph simplifies cycle detection by focusing only on waiting relationships.*

🡺 Lock ordering **reduces** deadlock chance in multithreading

🡺 If resources are requested in a fixed order, circular wait can be avoided. 🡺

🡺 **44. Preemption during recovery can cause:**  
A) Process starvation  
B) Automatic resolution  
C) Better performance  
D) Lock inversion

**Answer:** A  
📘 *If a process keeps losing resources, it may starve.*

**Which OS component is responsible for resource allocation and deadlock handling?**  
A) Scheduler  
B) MMU  
C) File System  
D) Kernel

**Answer:** D  
📘 *The kernel manages resources, including deadlock detection and handling.*

**🧠 1. Semaphore**

**🔹 What is it?**

A **semaphore** is a synchronization primitive used to control **access to shared resources** by multiple processes/threads in a concurrent system.

**🔹 Types:**

* **Counting Semaphore** – Allows a resource to be used by **multiple processes** (value > 1).
* **Binary Semaphore (similar to mutex)** – Allows only one process at a time (value is 0 or 1).

**🔹 Basic Operations:**

* wait(S) or P(S): Decrements the value. If it's < 0, process waits.
* signal(S) or V(S): Increments the value. If processes are waiting, one is unblocked.

**✅ Use Case:**

Used in **critical section problems**, **bounded buffer**, **readers-writers**, etc.

**🔒 2. Mutex (Mutual Exclusion Object)**

**🔹 What is it?**

A **mutex** is a lock that allows **only one thread/process** to access a **critical section** at a time.

A critical section is a code segment where shared resources are accessed.

**🔹 Difference from Semaphore:**

| **Feature** | **Semaphore** | **Mutex** |
| --- | --- | --- |
| Value | Integer | Binary (locked/unlocked) |
| Ownership | Not owned | Owned by a thread |
| Usage | Signaling + mutual exclusion | Only mutual exclusion |

**✅ Use Case:**

* Thread-safe code sections
* Prevent race conditions

**🍞🥛 3. Producer-Consumer Problem (Bounded Buffer Problem)**

**🔹 Description:**

Two types of processes:

* **Producer**: Produces items and puts them in a **buffer**.
* **Consumer**: Takes items from the buffer and consumes them.

**🔹 Problem:**

Synchronize access to ensure:

* **Producer** doesn't produce when the buffer is **full**.
* **Consumer** doesn't consume when the buffer is **empty**.

**🔹 Solution:**

Use **semaphores**:

* mutex: For mutual exclusion.
* empty: To track **empty slots** in buffer.
* full: To track **filled slots**.

**⚔️ 4. Deadlock vs Starvation**

**🔹 Deadlock:**

A condition where **a set of processes are blocked** because **each process is holding a resource and waiting for another**.

**Four necessary conditions (Coffman conditions)**:

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

➡️ **Example**: Process A holds Resource 1, waits for Resource 2. Process B holds Resource 2, waits for Resource 1.

**🔹 Starvation:**

Occurs when a process waits **indefinitely** to get a resource because **other higher-priority processes keep getting it first**.

➡️ **Example**: A low-priority process never gets CPU time because high-priority tasks keep coming.

**🔄 Key Difference:**

| **Feature** | **Deadlock** | **Starvation** |
| --- | --- | --- |
| Blocked? | All involved processes blocked | Only one or few processes starve |
| Cause | Circular wait and resource holding | Biased scheduling or priority |
| Recovery | Needs detection and recovery | Fixed with aging or fairness policy |

**What is Swap Space?**

* **Swap space** is a portion of the **hard disk (secondary storage)** used as **virtual memory**.
* When **RAM (primary memory)** is full, inactive pages are moved to swap space to free up physical memory.

**🔹 Why is it on Secondary Memory?**

* Because **disks (HDD/SSD)** offer **larger but slower** storage.
* The OS uses disk space as an **overflow** area when RAM is insufficient.

**🔹 Relation to Virtual Memory:**

* **Virtual memory** is an abstraction that combines RAM + swap.
* But swap space **resides physically on secondary memory**, not in virtual memory itself.

**🔹 Clustered System:**

* A **clustered system** connects **multiple independent computers (nodes)** to work together as a **single system**.
* These systems share storage and are **tightly coupled** for **high availability, load balancing, and parallel processing**.
* Each node has **its own CPU and memory**, making it a **type of multiple-CPU system**.

**🔍 Why not others?**

**A) Mini Computer ❌**

* Older term for mid-range computers; may or may not have multiple CPUs.

**B) Super Computer ❌**

* Very powerful machines with **many CPUs**, but usually a **single system**, not **multiple systems working together** like clusters.

**D) Network Computer ❌**

* Basic computers that rely on servers over a network; not designed for multi-CPU computation.

**21. Fastest form of Inter-Process Communication (IPC):**

**A) Signals**  
**B) Shared Memory** ✅  
**C) Message Queues**  
**D) Pipes**

**✅ Correct Answer: B) Shared Memory**

**➡️ Explanation:**

**🔹 Shared Memory:**

* It's the **fastest IPC mechanism** because:
  + Data is shared **directly in memory**.
  + There is **no need to copy data** between processes via the kernel.
  + Once the memory is mapped, processes can **read/write instantly** without syscalls.

⚡ Speed comes from **direct access**, avoiding the overhead of context switching and copying.

**🔍 Why not others?**

**A) Signals ❌**

* Lightweight, but used only to **notify** events (not for data transfer).
* Minimal data can be passed, mostly just flags.

**C) Message Queues ❌**

* Go through the **kernel**, require **copying** of data.
* More overhead compared to shared memory.

**D) Pipes ❌**

* Also involve **kernel intervention**.
* Data is transferred in **stream format**, requiring **read/write syscalls** — slower than direct access.

PREVIOUS YRER

**3. Kernel generates signals to:**

* A) Notify processes of events
* B) Call user space
* C) Stop execution of ended process
* D) Kernel does not generate signals

**Answer:** A) Notify processes of events  
**Explanation:** Signals are used by the kernel to notify a process about system events.

**4. Which one of the following is not a part of main thread libraries in use today?**

* A) POSIX Pthreads
* B) Win32 threads
* C) Java threads
* D) HPUX threads

**Answer:** D) HPUX threads  
**Explanation:** HPUX is an OS, not a widely used thread library.

**6. Where is Swap Space located?**

* A) Primary Memory
* B) Secondary Memory
* C) Registers
* D) Virtual Memory

**Answer:** B) Secondary Memory  
**Explanation:** Swap space resides on disk (secondary memory) and extends virtual memory.

🡺 Time sharing and Multi-tasking🡺 Logical extension of multiprogramming

🡺

# Shell

**PS1 (Prompt String 1):**

* **This environment variable defines the format of the primary shell prompt you see in the terminal.**
* **Example:**

**PS1="\u@\h:\w\$ "**

**This would display:  
username@hostname:current-directory$**

* **You can customize it to include username, hostname, time, etc.**

**🔍 What do the other options mean?**

**B) PS2 ❌**

* **This defines the secondary prompt, shown when a command is not complete.  
  Example: when you type an unmatched quote:**

**> echo "hello**

**> ⬅️ this is PS2 prompt**

**C) SHELL ❌**

* **Contains the path of the default shell, e.g., /bin/bash.**

**D) $HOME ❌**

* **Stores the home directory path of the current user.**

**To properly process regex in shell:**

**A) Parentheses**  
**B) Backslashes** ✅  
**C) Double quotation marks**  
**D) Single quotation marks**

**✅ Correct Answer: B) Backslashes**

**➡️ Explanation:**

In shell (like **bash**), when you're using **regular expressions**, you often need to **escape special characters** to make them behave correctly. This is where **backslashes (\)** come into play.

**🔍 Example:**

Suppose you're using grep to match a pattern with parentheses:

bash

CopyEdit

grep "\(hello\)" file.txt

* Here, ( and ) are **special characters in regex**, used for grouping.
* In **basic regular expressions (BRE)** used by default in grep, these must be **escaped with backslashes**.

So, to make regex patterns **work as intended**, we often use:

✅ **Backslashes** → to **escape special regex characters** in shell.

**🔍 Why not others?**

**A) Parentheses ❌**

* They are **part of regex**, not what processes it.
* Also need to be escaped or quoted.

**C) Double quotation marks ❌**

* Used to **preserve spaces and variable expansion**, not specifically for regex.
* They don't escape special regex characters.

**D) Single quotation marks ❌**

* Prevent all expansions (including variables), but again **not sufficient alone** for regex processing.
* You often still need backslashes **inside** quotes for regex to work.

### 🟩 Summary:

To ensure regular expressions are processed properly in shell, **backslashes (\)** are used to **escape special regex characters**.

In Unix/Linux systems, **signals** are used to notify a process that a particular event has occurred (like termination, pause, etc.).

Most signals can be **caught**, **ignored**, or **handled** by processes using custom signal handlers — but **some signals are so critical**, they **cannot be blocked, caught, or ignored**.

**🔥 These uncatchable signals are:**

1. **SIGKILL (signal 9)** – Immediately **terminates** a process.
   * **Cannot be caught or ignored**.
   * Bypasses all cleanup.
   * Used by the OS or root to **forcefully kill** misbehaving processes.
2. **SIGSTOP (signal 19)** – **Pauses (stops)** a process execution.
   * Also **cannot be caught or blocked**.
   * Can be resumed with SIGCONT.

**Windows NT Kernel:**

* The Windows NT family (including Windows 10, 11, and Server editions) uses a **Hybrid Kernel architecture**.

**🔹 What is a Hybrid Kernel?**

* A **hybrid kernel** is a blend of:
  + **Monolithic kernel**: where most OS services run in **kernel space** (fast performance).
  + **Microkernel**: where most services run in **user space** (better modularity and security).
* It tries to take **the performance of monolithic** and **modularity of microkernel**.

**🔹 Windows NT Design:**

* Core services (like memory manager, scheduler, etc.) run in **kernel mode**.
* Some modular components (like drivers) are designed to be more **replaceable or isolated**, inspired by microkernel ideas.

**🔍 Why not others?**

| **Option** | **Reason it’s incorrect** |
| --- | --- |
| **A) Monolithic** ❌ | Used by Linux. All services in kernel space. |
| **B) Micro** ❌ | Used by Minix, QNX. Most services in user space. |
| **D) Nano** ❌ | Not a kernel type. Could refer to "Nano Server" in Windows, which is unrelated here. |

**🟩 Summary:**

The **Windows NT kernel** (used in modern Windows OSes) is a **Hybrid Kernel**, combining features of both monolithic and microkernel designs.

**Important Concept: Order of static blocks in Java**

In Java, **all static blocks run in the order they appear in the code**, **before the main() method is executed**.

**2. Commands Associated with Files/Directories & Other Basic Commands**

The power of Linux often comes from its robust command-line interface. Here are essential commands you'll use frequently:

* ls (list): Lists the contents of a directory.
  + ls: Lists contents of the current directory.
  + ls -l: Long listing format (more details like permissions, owner, size, date).
  + ls -a: Lists all files, including hidden ones (starting with .).
  + ls -lh: Long listing with human-readable file sizes.
* cd (change directory): Changes your current working directory.
  + cd ~: Go to your home directory.
  + cd /path/to/directory: Go to a specific absolute path.
  + cd ..: Go up one directory level.
  + cd -: Go back to the previous directory you were in.
* pwd (print working directory): Displays the absolute path of your current directory.
* mkdir (make directory): Creates new directories.
  + mkdir my\_new\_folder
  + mkdir -p path/to/nested/folder: Creates parent directories if they don't exist.
* rmdir (remove directory): Removes *empty* directories.
  + rmdir empty\_folder
* touch: Creates an empty new file or updates the timestamp of an existing file.
  + touch newfile.txt
* cp (copy): Copies files or directories.
  + cp file.txt /path/to/destination/: Copies file.txt to the destination.
  + cp -r folder/ /path/to/destination/: Recursively copies a directory and its contents.
* mv (move/rename): Moves files or directories, or renames them.
  + mv oldname.txt newname.txt: Renames oldname.txt to newname.txt.
  + mv file.txt /path/to/destination/: Moves file.txt to the destination.
* rm (remove): Removes files or directories. **Be careful with rm as it permanently deletes!**
  + rm file.txt
  + rm -r folder/: Recursively removes a directory and its contents.
  + rm -f file.txt: Forces removal, ignores non-existent files and arguments, never prompts.
  + rm -rf folder/: **Extremely dangerous!** Recursively and forcefully removes a directory. Use with extreme caution.
* cat (concatenate and display): Displays the content of files.
  + cat file1.txt file2.txt: Displays content of both files.
  + cat > newfile.txt: Creates a new file and allows you to type content directly (press Ctrl+D to save and exit).
* more/less: Used to view large files page by page. less is generally preferred as it allows scrolling both forward and backward.
  + more long\_document.txt
  + less another\_long\_document.txt
* head/tail: Displays the beginning (head) or end (tail) of files.
  + head -n 5 file.txt: Shows the first 5 lines.
  + tail -n 10 file.txt: Shows the last 10 lines.
  + tail -f logfile.log: "Follows" the file, displaying new lines as they are added (useful for logs).
* man (manual): Provides documentation (manual pages) for commands.
  + man ls: Shows the manual page for the ls command. Press q to exit.

**Operators like Redirection, Pipe**

These operators are fundamental for combining commands and managing input/output in the shell.

* **Redirection:** Changes where a command's standard output or standard input goes.
  + > (Redirect Standard Output): Sends the output of a command to a file, overwriting the file if it exists.
    - ls -l > file\_list.txt: Puts the ls -l output into file\_list.txt.
  + >> (Append Standard Output): Sends the output of a command to a file, appending to it if it exists, or creating it if it doesn't.
    - echo "Hello" >> greeting.txt: Adds "Hello" to the end of greeting.txt.
  + < (Redirect Standard Input): Takes input for a command from a file instead of the keyboard.
    - sort < unsorted\_names.txt: Sorts the content of unsorted\_names.txt.
  + 2> (Redirect Standard Error): Redirects error messages to a file.
    - command\_that\_might\_fail 2> errors.log: Sends error messages to errors.log.
  + &> (Redirect Standard Output and Standard Error): Redirects both to a file.
    - command\_output &> all\_output.log
* **Pipe (|):** Connects the standard output of one command to the standard input of another command. This allows you to chain commands together, where the result of one becomes the starting point for the next.
  + ls -l | less: Pipes the long listing of files to less, allowing you to scroll through it.
  + ps aux | grep firefox: Lists all running processes (ps aux) and then filters (grep) for lines containing "firefox".
  + cat file.txt | wc -l: Counts the number of lines (wc -l) in file.txt.

**4. What are File Permissions and How to Set Them?**

Linux is a multi-user system, and file permissions are crucial for controlling who can do what with files and directories.

Each file/directory has permissions for three categories of users:

1. **User (u):** The owner of the file/directory.
2. **Group (g):** Members of the group that owns the file/directory.
3. **Others (o):** Everyone else on the system.

And three types of permissions:

1. **Read (r):**
   * **Files:** Can view the file's content.
   * **Directories:** Can list the directory's contents (ls).
2. **Write (w):**
   * **Files:** Can modify or delete the file.
   * **Directories:** Can create, delete, or rename files within the directory (even if you don't have write permission on the files themselves!).
3. **Execute (x):**
   * **Files:** Can run the file as a program/script.
   * **Directories:** Can enter (cd) into the directory and access its contents.

When you run ls -l, you'll see something like: -rwxr-xr--

* The first character indicates the file type (- for regular file, d for directory, l for symbolic link).
* The next nine characters are the permissions, grouped into threes for User, Group, and Others:
  + rwx (User): Read, Write, Execute
  + r-x (Group): Read, Execute (no write)
  + r-- (Others): Read only

**5. Permissions (chmod, chown, etc); Access Control List**

* **chmod (change mode):** Used to change file/directory permissions.
  + **Symbolic Mode:** Using u, g, o, a (all) with + (add), - (remove), = (set exactly).
    - chmod u+x myscript.sh: Add execute permission for the owner.
    - chmod go-w myfile.txt: Remove write permission for group and others.
    - chmod a=rw newfile.txt: Set read and write permissions for all.
  + **Octal (Numeric) Mode:** Each permission (r, w, x) has a numeric value:
    - r = 4
    - w = 2
    - x = 1
    - Combine values for each category: rwx = 4+2+1=7, rw- = 4+2+0=6, r-x = 4+0+1=5, r-- = 4+0+0=4.
    - chmod 755 myscript.sh: Owner gets rwx, Group gets r-x, Others get r-x. (Common for executable scripts/programs).
    - chmod 644 myfile.txt: Owner gets rw-, Group gets r--, Others gets r--. (Common for data files).
* **chown (change owner):** Changes the owner of a file or directory. Only root or sudo users can typically use this.
  + chown newuser file.txt
  + chown newuser:newgroup file.txt: Changes both owner and group.
* **chgrp (change group):** Changes the group ownership of a file or directory.
  + chgrp newgroup file.txt
* **Access Control Lists (ACLs):** (Briefly mention)
  + ACLs provide a more granular way to set permissions than traditional Unix permissions. They allow you to define permissions for specific users or groups beyond the owner, group, and others.
  + Commands: getfacl (view ACLs), setfacl (set ACLs). We won't go deep into this in this intro.

**6. Network Commands (telnet, ftp, ssh, sftp, finger)**

These commands are used for network communication, primarily for remote access and file transfer.

* **telnet (Teletype Network):**
  + An old, insecure protocol for remote login. Sends data (including passwords) in plain text. **Avoid for sensitive operations.**
  + telnet hostname port
* **ftp (File Transfer Protocol):**
  + An old, insecure protocol for transferring files between computers. Also sends credentials in plain text. **Avoid for sensitive file transfers.**
  + ftp hostname
* **ssh (Secure Shell):**
  + **The secure standard for remote login and executing commands on a remote server.** Encrypts all communication.
  + ssh username@hostname
  + ssh -p 2222 username@hostname: Connects to a specific port.
* **sftp (SSH File Transfer Protocol):**
  + A secure file transfer protocol that runs over SSH. Preferred over FTP.
  + sftp username@hostname (then use put, get commands inside the sftp prompt)
* **finger:**
  + (Mostly legacy) Used to get information about users on a system. Often disabled for security reasons.
  + finger username

**7. System Variables like – PS1, PS2 etc. How to set them**

**Environment variables** are dynamic-named values that affect the way running processes behave. They are essentially key-value pairs.

* **echo $VARIABLE\_NAME:** Displays the value of an environment variable.
* **export VARIABLE\_NAME=value:** Sets an environment variable for the current session and any child processes.

**System variables like – PS1, PS2 etc. How to set them:**

* Explain environment variables.
* PS1: Primary prompt string.
* PS2: Secondary prompt string.
* How to view (echo $VARIABLE) and set (export VARIABLE=value) them.
* **PS1 (Prompt String 1):**
  + Controls the primary command prompt you see in your terminal. You can customize its appearance.
  + *Example:* PS1="\u@\h:\w\$ " (shows username@hostname:current\_directory$ )
* **PS2 (Prompt String 2):**
  + Controls the prompt shown when a command is incomplete and the shell expects more input (e.g., when you open a quote but don't close it).
  + *Default:* >
* **Other common variables:**
  + PATH: A list of directories the shell searches for executable commands.
  + HOME: Your home directory.
  + USER: Your username.

To make variable changes permanent, you typically add export commands to your shell's configuration file (e.g., ~/.bashrc, ~/.zshrc).

**Shell Programming Lecture**

Now, let's look at how to automate tasks and write scripts in the Linux shell.

* **What is Shell; What are different shells in Linux?**
  + **Shell:** A command-line interpreter that provides an interface for the user to interact with the operating system. It reads commands from the user or from a file (a shell script) and executes them.
  + **Different Shells in Linux:**
    - **Bash (Bourne Again SHell):** The most common and default shell on many Linux distributions.
    - **Zsh (Z Shell):** A modern shell with many features like advanced tab completion and plugin support.
    - **Ksh (Korn Shell):** An older shell, but still powerful.
    - **Csh/Tcsh (C Shell):** Shells with C-like syntax.
    - *Most of what we cover will be applicable to Bash.*
* **Shell variables; Wildcard symbols:**
  + **Shell Variables:** Used to store data within a script or shell session.
    - my\_variable="Hello World" (no spaces around =)
    - echo $my\_variable (to access the value, use $)
  + **Wildcard Symbols (Globbing):** Used to match filenames and paths.
    - \*: Matches zero or more characters.
      * ls \*.txt: Lists all files ending with .txt.
    - ?: Matches exactly one character.
      * ls file?.txt: Matches file1.txt, fileA.txt, etc.
    - []: Matches any one of the characters inside the brackets.
      * ls [abc]\*.txt: Matches files starting with 'a', 'b', or 'c' and ending with .txt.
      * ls [0-9].txt: Matches files named 0.txt, 1.txt, etc.
* **Shell meta characters; Command line arguments; Read, Echo:**
  + **Shell Meta Characters:** Characters that have special meaning to the shell and need to be quoted or escaped if you want them to be treated literally.
    - ;: Command separator (run commands sequentially).
    - &: Run a command in the background.
    - &&: Logical AND (run next command only if previous succeeded).
    - ||: Logical OR (run next command only if previous failed).
    - (): Group commands, create subshells.
    - #: Start a comment (rest of the line is ignored).
    - ": Double quotes (preserve most special characters, but allow variable expansion).
    - ': Single quotes (preserve *all* special characters, no variable expansion).
    - ` or $(...): Command substitution (execute a command and substitute its output).
      * my\_date=$(date)
  + **Command Line Arguments:** When you run a script, you can pass values to it as arguments.
    - $0: The name of the script itself.
    - $1, $2, $3, ...: The first, second, third, etc., argument passed to the script.
    - $#: The total number of arguments.
    - $@: All arguments as separate strings (best for loops).
    - $\*: All arguments as a single string.
  + **read:** Reads input from the user (standard input) and stores it in a variable.
    - read -p "Enter your name: " name
  + **echo:** Prints text or variable values to the standard output (terminal).
    - echo "Hello, world!"
    - echo "Your name is $name."

**Session 3: Shell Programming (Advanced)**

Welcome to Session 3! We'll now explore more powerful constructs in shell scripting, allowing you to create more intelligent and dynamic programs.

**1. Decision Loops**

Decision loops are control flow statements that allow your script to make choices and repeat actions based on conditions.

* **if else (Conditional Execution):** The most basic form of decision making. It executes a block of code if a condition is true.

Bash

#!/bin/bash

read -p "Enter a number: " num

if [ "$num" -gt 10 ]; then

echo "$num is greater than 10."

elif [ "$num" -eq 10 ]; then # 'elif' is optional, for multiple conditions

echo "$num is equal to 10."

else

echo "$num is less than 10."

fi

* **test (or [ ]):** The test command (or its shorthand [ ]) is used to evaluate conditions. It returns an exit status of 0 (true) or 1 (false).
  + **Numeric Comparisons:**
    - -eq: Equal to
    - -ne: Not equal to
    - -gt: Greater than
    - -ge: Greater than or equal to
    - -lt: Less than
    - -le: Less than or equal to
    - *Example:* [ "$var1" -eq "$var2" ]
  + **String Comparisons:**
    - =: Equal to (string)
    - !=: Not equal to (string)
    - -z: String is empty (zero length)
    - -n: String is not empty (non-zero length)
    - *Example:* [ -z "$name" ]
  + **File Comparisons:**
    - -e file: File exists
    - -f file: File exists and is a regular file
    - -d dir: Directory exists
    - -r file: File is readable
    - -w file: File is writable
    - -x file: File is executable
    - *Example:* if [ -f "myfile.txt" ]; then ... fi
  + **Logical Operators (within [ ]):**
    - -a: Logical AND
    - -o: Logical OR
    - *Example:* if [ -f "file1.txt" -a -f "file2.txt" ]; then ... fi
  + **Newer [[ ]] for Advanced Conditions:**
    - Provides more powerful features, including regex matching.
    - [[ "$name" == "Alice" ]]
    - [[ "$file" =~ \.txt$ ]] (regex match)
    - *Preferred in modern Bash scripting due to fewer parsing issues.*
* **case controls (Multi-way Branching):** Useful when you have many possible values for a variable and want to execute different code blocks based on those values.

Bash

#!/bin/bash

read -p "Enter a color (red, green, blue): " color

case "$color" in

"red")

echo "You chose red."

;; # End of a case block

"green" | "emerald") # Multiple patterns can be matched

echo "You chose green or emerald."

;;

"blue")

echo "You chose blue."

;;

\*) # Default case, matches anything else

echo "Unknown color."

;;

esac # End of case statement

* **while…until (Looping):**
  + **while loop:** Executes a block of code as long as a condition is true.

Bash

#!/bin/bash

counter=1

while [ "$counter" -le 5 ]; do

echo "Counter: $counter"

counter=$((counter + 1)) # Arithmetic expression

done

* + **until loop:** Executes a block of code *until* a condition becomes true (i.e., as long as the condition is false).

Bash

#!/bin/bash

countdown=5

until [ "$countdown" -eq 0 ]; do

echo "Countdown: $countdown"

countdown=$((countdown - 1))

sleep 1 # Pause for 1 second

done

echo "Blast off!"

* **for loop (Iterating):** Iterates over a list of items or a sequence of numbers.
  + **Iterating over a list of items:**

Bash

#!/bin/bash

for fruit in apple banana orange; do

echo "I like $fruit."

done

* + **Iterating over files (using wildcards):**

Bash

#!/bin/bash

for file in \*.txt; do

echo "Processing text file: $file"

# Add commands to process each file here

done

* + **C-style for loop (for numeric ranges):**

Bash

#!/bin/bash

for (( i=1; i<=5; i++ )); do

echo "Number: $i"

done

**2. Regular Expressions; Arithmetic Expressions**

* **Regular Expressions (Regex):**
  + Powerful patterns used for searching and manipulating text. They define a search pattern for strings.
  + Often used with commands like grep, sed, awk, and in [[ ]] conditional statements.
  + **Basic Regex (BRE) / Extended Regex (ERE):** Many tools support ERE by default or with an option (grep -E).
  + **Common Regex Elements:**
    - .: Matches any single character (except newline).
    - \*: Matches zero or more occurrences of the preceding character/group.
    - +: Matches one or more occurrences of the preceding character/group.
    - ?: Matches zero or one occurrence of the preceding character/group.
    - ^: Matches the beginning of a line.
    - $: Matches the end of a line.
    - []: Matches any single character within the brackets. (e.g., [aeiou])
    - [^]: Matches any single character *not* within the brackets. (e.g., [^0-9])
    - |: OR operator (matches A or B). (e.g., (cat|dog))
    - (): Grouping for applying operators.
    - \d: Matches any digit ([0-9]).
    - \w: Matches any word character (alphanumeric + underscore).
    - \s: Matches any whitespace character.
  + **Example with grep:**
    - grep "^[0-9]" log.txt: Finds lines starting with a digit.
    - grep "error.\*failed" app.log: Finds lines containing "error" followed by "failed" with anything in between.
    - grep -E "color|colour" text.txt: Finds lines with either "color" or "colour".
* **Arithmetic Expressions:**
  + Shell scripts primarily handle strings. To perform arithmetic operations, you need special constructs.
  + **$((expression)):** The most common and recommended way for integer arithmetic.
    - result=$((10 + 5))
    - x=5; y=2; quotient=$((x / y)) (integer division)
    - increment=$((counter + 1))
    - Supports +, -, \*, /, % (modulo), \*\* (exponentiation).
  + **expr command:** Older way, less flexible.
    - result=$(expr 10 + 5) (note spaces are required)
  + **bc (arbitrary precision calculator):** For floating-point arithmetic.
    - echo "scale=2; 10 / 3" | bc (output: 3.33)

**3. More Examples in Shell Programming**

Let's look at how to combine these concepts into more practical scripts:

* **Example 1: Backup Script**

Bash

#!/bin/bash

SOURCE\_DIR="/home/user/documents"

BACKUP\_DIR="/mnt/backup/my\_docs\_$(date +%Y%m%d)" # Date in filename

if [ ! -d "$SOURCE\_DIR" ]; then

echo "Error: Source directory '$SOURCE\_DIR' does not exist."

exit 1 # Exit with error

fi

echo "Creating backup directory: $BACKUP\_DIR"

mkdir -p "$BACKUP\_DIR"

if [ -d "$BACKUP\_DIR" ]; then

echo "Copying files from $SOURCE\_DIR to $BACKUP\_DIR..."

cp -r "$SOURCE\_DIR"/\* "$BACKUP\_DIR"/

echo "Backup completed successfully!"

else

echo "Error: Failed to create backup directory."

exit 1

fi

* **Example 2: User Management (Simplified)**

Bash

#!/bin/bash

read -p "Enter new username: " new\_user

if id "$new\_user" &>/dev/null; then # Check if user exists

echo "User '$new\_user' already exists."

else

echo "Adding user '$new\_user'..."

sudo useradd -m "$new\_user" # Requires sudo privilege

if [ $? -eq 0 ]; then # Check exit status of previous command (0 means success)

echo "User '$new\_user' added successfully."

else

echo "Failed to add user '$new\_user'."

fi

fi

* **Example 3: Log File Analyzer (using grep and pipes)**

Bash

#!/bin/bash

LOG\_FILE="apache\_access.log"

if [ ! -f "$LOG\_FILE" ]; then

echo "Error: Log file '$LOG\_FILE' not found."

exit 1

fi

echo "--- Log Analysis for $LOG\_FILE ---"

echo "Total lines: $(wc -l < "$LOG\_FILE")"

echo "Number of GET requests: $(grep -c "GET" "$LOG\_FILE")"

echo "IP addresses that accessed the server:"

grep -oE "([0-9]{1,3}\.){3}[0-9]{1,3}" "$LOG\_FILE" | sort -u

echo "Top 5 most frequent requests:"

awk '{print $7}' "$LOG\_FILE" | sort | uniq -c | sort -nr | head -n 5

**Lab: (4 hours)**

This lab will solidify your understanding of decision loops, regular expressions, and arithmetic expressions.

**Task 1: Shell Programs related to Session 3 - Decision Loops**

1. **Number Comparison Script:**
   * Create a script compare\_numbers.sh.
   * Prompt the user to enter two numbers.
   * Use if, elif, and else with numeric test operators (-gt, -lt, -eq) to compare the numbers and print whether the first is greater than, less than, or equal to the second.
2. **File Type Checker:**
   * Create a script check\_file\_type.sh.
   * Prompt the user to enter a filename.
   * Use if with file test operators (-f, -d, -r, -w, -x) to:
     + Check if the file exists.
     + If it's a regular file, print "It's a regular file."
     + If it's a directory, print "It's a directory."
     + Check if it's readable, writable, or executable and print relevant messages.
     + Use an else block if the file doesn't exist.
3. **Basic Menu Script (using case):**
   * Create a script menu.sh.
   * Display a simple menu:
   * 1. List files
   * 2. Show current directory
   * 3. Exit
   * Enter your choice:
   * Use read to get the user's choice.
   * Use a case statement to execute ls -l, pwd, or exit the script based on the choice. Include a default case for invalid input.
4. **Countdown Timer (using while or until):**
   * Create a script countdown.sh.
   * Prompt the user for a starting number for the countdown.
   * Use a while or until loop to count down from that number to 0, printing each number.
   * Add sleep 1 inside the loop to pause for one second.
5. **Directory Looper (using for):**
   * Create a few empty directories in your current location (e.g., dir1, dir2, another\_dir).
   * Create a script loop\_dirs.sh.
   * Use a for loop to iterate through all directories in the current working directory (e.g., for d in \*/; do ... done).
   * Inside the loop, echo the name of each directory found.

**Task 2: Shell Programs related to Session 3 - Regular and Arithmetic Expressions**

1. **Simple Calculator Script:**
   * Create a script calculator.sh.
   * Prompt the user to enter two numbers and an operator (+, -, \*, /).
   * Use if / elif / else or case and arithmetic expressions ($((...))) to perform the calculation.
   * Print the result.
   * Handle division by zero if applicable. (Hint: if [ "$num2" -eq 0 ]; then ... fi)
2. **Log File Filtering (using grep with Regex):**
   * Create a dummy log file named sample.log with a few lines containing:
     + "ERROR: something went wrong."
     + "WARNING: low disk space."
     + "INFO: successful operation."
     + "DEBUG: variable x is 10."
     + "ERROR: another critical issue."
   * Create a script log\_filter.sh.
   * Use grep with regular expressions to:
     + Find all lines containing "ERROR".
     + Find all lines starting with "WARNING".
     + Find lines that contain either "ERROR" or "WARNING" (using grep -E).
     + Find lines that contain a specific IP address pattern (e.g., 192.168.1.1 - if you add some to your sample log).
3. **Text Search and Replace (conceptual, using sed or awk - minimal intro):**
   * (Optional, as sed/awk are topics on their own) Briefly demonstrate sed for a simple replacement:
     + echo "This is a test." | sed 's/test/example/' (Replaces first occurrence)
     + echo "one one one" | sed 's/one/two/g' (Replaces all occurrences 'g' for global)
   * Create a file data.txt with some text.
   * Write a script that uses sed to replace a specific word in data.txt and save the result to a new file.

**Important Notes for Lab:**

* Always start your shell scripts with #!/bin/bash (shebang line).
* Make your scripts executable: chmod +x your\_script\_name.sh.
* Run your scripts using ./your\_script\_name.sh.
* Use echo frequently to print messages and debug variable values.