Prathamesh Arvind Jadhav

Operating System for Placement

LEC-1-Introduction to Operating Systems (OS)

Introduction to Operating Systems (OS)

An **Operating System (OS)** is one of the most important system software components in any computer. It acts as a **bridge** between the **user and the computer hardware**, ensuring efficient and convenient usage of the computer's resources.

Key Concepts to Understand

1. Application Software vs. System Software

Feature	Application Software	System Software
Purpose	Performs specific tasks for the user.	Manages and controls hardware resources.
Examples	MS Word, Excel, VLC Player	Windows, Linux, macOS, Android
User Interaction	Direct interaction with the user	Works mostly in the background
Dependency	Depends on system software to run	Independent (runs before apps)

2. What is an Operating System?

- The **Operating System (OS)** is **system software** that:
  - Manages all **hardware** and **software** resources.
  - Acts as an **intermediary** between the user and computer hardware.
  - Provides an **environment** where the user can execute applications **efficiently and safely**.
  - Hides the **complexity** of the hardware using **abstractions** (like virtual memory, file systems, etc.).

Why Do We Need an Operating System?

Case: What if there is No OS?

If we did not have an OS:

1. **Applications would become bulky and complex:**
   - Each application must contain code to manage hardware directly (like memory, CPU, printer, etc.).
   - Leads to duplication of efforts and complexity.
2. **No resource management:**

   - One app could use **all memory**, **CPU time**, or **devices**—leading to **starvation** of others.
3. **No memory protection:**
   - One program could overwrite the memory of another, causing crashes and corruption.

□ What is the OS Made Of?

The **Operating System is a collection of system software** components working together to:

- Manage resources.
- Provide abstraction layers.
- Enable communication between user apps and hardware.

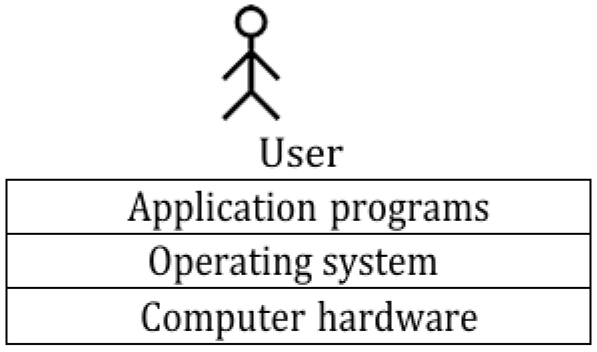
Examples of components:

- **Kernel** (core part managing CPU, memory, devices)
- **File System Manager**
- **Device Drivers**
- **Command Interpreters (Shell)**
- **Security and Access Controls**

□ Key Functions of an Operating System

Let's break it down simply:

OS Function	Description
Hardware Access	Controls and coordinates access to physical hardware devices (I/O, CPU).
Interface for User	Provides a user interface (e.g., GUI, CLI) to interact with the system.
Resource Management (Arbitration)	Manages CPU, memory, files, input/output devices, and security.
Abstraction	Hides hardware complexities. E.g., gives you a "file" instead of block storage.
Execution Support for Apps	Ensures safe execution of user applications with **protection and isolation**.



- **User** interacts with applications.
- **Applications** make system calls to OS for resources.
- **OS** interacts directly with hardware.
- **Hardware** executes the final instructions.

□ Summary of Roles of an Operating System

- **Acts as a manager** for:
  - CPU scheduling
  - Memory allocation
  - File handling
  - Device control
  - User permissions & security
- **Acts as an interface** between:
  - User and machine
  - Application software and hardware
- **Improves efficiency and usability** of the system.

□ Real-Life Analogy

Think of the OS as the **manager of a hotel**:

- Guests (Applications) check-in and request services (CPU, memory).
- The manager (OS) allocates rooms (memory), handles bookings (file systems), and ensures safety and smooth operation.
- Guests don't worry about how the elevator or kitchen works (hardware abstraction).

LEC-2: Types of OS

OS Goals

Before understanding the types of Operating Systems, it’s important to know **what they aim to achieve**. The goals of an OS include:

1. Maximum CPU Utilization

- Ensure the **CPU is never idle** unless absolutely necessary.
- The OS should keep the CPU busy by smartly scheduling tasks.

2. Less Process Starvation

- No task should **wait indefinitely** for resources.
- OS should ensure **fairness** among running processes.

3. Higher Priority Job Execution

- Some tasks are **more critical** (e.g., system tasks, real-time updates).
- The OS must support **priority scheduling**, allowing such jobs to execute before others.

□ Types of Operating Systems

1. **Single-Process Operating System**

- **Only one process** can execute at a time from the ready queue.
- No concurrency or parallelism.
- Earliest and **simplest OS type**.

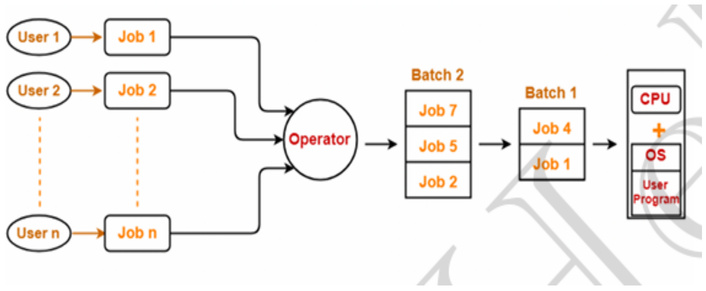
□ **Example:** MS-DOS (1981)
□ **Use case:** Old personal computers and embedded systems.

---

2. Batch Processing Operating System

How it Works:

1. Users **prepare jobs** on punch cards or magnetic tapes.
2. Jobs are **submitted to an operator** (not directly to the CPU).
3. Operator **groups similar jobs into batches**.
4. Entire batch is **executed one after another**.



Advantages:

- Reduces manual setup time for every job.
- Efficient for **large-scale repetitive tasks**.

Disadvantages:

- **No real-time execution**.
- **No priority support** — higher priority jobs can't interrupt.
- If one job **requires I/O**, the CPU may become idle.
- Risk of **starvation** for short or urgent jobs.

Historical Example: IBM's early batch systems.

---

3. Multiprogramming Operating System

Key Idea:

- **Multiple jobs** (code + data) are **loaded into memory**.
- CPU executes one job, and when that job needs to do I/O, the **CPU switches to another**.
- Keeps CPU **utilized** at all times.

Concepts:

- **Context Switching:** Switching CPU from one process to another.
- Efficient **memory usage** and **less idle time**.

Example: ATLAS (Manchester University, 1950s–60s)

---

4. Multitasking Operating System

Logical Extension of Multiprogramming:

- Allows the **user to perform multiple tasks** at once.
    - e.g., Listening to music + Writing a document
- **Time-sharing**: CPU switches between tasks quickly.
- User feels that tasks are executing **simultaneously**.

Benefits:

- Better **user responsiveness**.
- Improves **interactive experience**.

Example: THE (Dijkstra's OS), Windows NT

□ 5. Multiprocessing Operating System

Key Feature:

- A system with **more than one CPU** (multi-core systems).
- All CPUs share memory and devices but may **run different tasks simultaneously**.

Benefits:

- Increased **reliability** — if one CPU fails, the others keep running.
- **Parallel processing** = faster computation.
- Lesser starvation due to **task distribution**.

Example: Windows Server OS, Linux SMP systems

□ 6. Distributed Operating System

What It Is:

- A **network of independent computers (nodes)** connected together.
- Appears to the user as **a single system**.
- OS coordinates **resource sharing and computation** across all nodes.

Key Traits:

- Nodes are **loosely connected**, but work together.
- Improves **scalability** and **resource utilization**.

- Great for **cloud computing**, **grid computing**.

Example: LOCUS (early Distributed OS)

□ 7. Real-Time Operating System (RTOS)

Designed For:

- **Critical systems** where **timing is everything**.
- Must respond to inputs **within strict time constraints**.

Types:

- **Hard Real-Time OS**: Missing a deadline = system failure (e.g., pacemaker).
- **Soft Real-Time OS**: Occasional deadline misses tolerated (e.g., streaming).

Used In:

- Robotics
- Air Traffic Control
- Industrial Automation

Example: ATCS (Air Traffic Control System), VxWorks, RTLinux

□ Summary Table

Type	**CPUs**	**Features**	**Examples**
Single-process OS	1	Only one job at a time	MS-DOS
Batch-processing OS	1	Executes job batches, low CPU utilization during I/O	IBM Batch Systems
Multiprogramming OS	1	Multiple jobs in memory; switches on I/O	ATLAS
Multitasking OS	1	Logical multitasking, improves responsiveness	Windows NT, THE

Type	CPUs	Features	Examples
Multiprocessing OS	>1	True parallelism, multiple CPUs	Linux SMP, Windows Server
Distributed OS	Many	Networked computers working as one system	LOCUS
Real-Time OS (RTOS)	1 or more	Responds within strict timing limits	ATCS, RTLinux, VxWorks

LEC-3: Multi-Tasking vs Multi-Threading

Basic Definitions

1. Program

- A **program** is a **static** set of instructions written to perform a specific task.
- It is a **compiled file** stored on disk (like .exe, .out).
- Not executing until it is loaded into memory.

□ **Example**: A text editor like Notepad is a program stored on disk. When you double-click to open it, it becomes a process.

---

2. Process

- A **process** is a **program in execution**.
- It is **loaded into main memory (RAM)** and managed by the operating system.
- A process has its own:
  - Memory space
  - Process ID (PID)
  - Code, data, stack, and heap

□ **Example**: When you open Chrome, each tab might be a separate process (depending on design).

---

3. Thread

- A **thread** is the **smallest unit of execution** within a process.

- It is a **lightweight subprocess**.
- All threads in a process **share**:
    - The same memory space
    - Same resources like file descriptors, etc.
- But each thread has its **own stack and program counter**.

 Example: While typing in MS Word (one thread), spell checking happens in the background (another thread).

Multi-Tasking vs Multi-Threading

Aspect	Multi-Tasking	Multi-Threading
Definition	Running multiple **processes** at once	Running multiple **threads** of a single process
Unit of Execution	Process	Thread
Memory Isolation	Processes are isolated; each gets separate memory	Threads share the same memory space
Context Switching	OS switches between different processes	OS switches between threads of the same process
Resource Use	Heavyweight: higher memory and resource usage	Lightweight: low resource usage
CPU Requirement	Can run on single CPU (time sharing)	Can also run on single CPU; more efficient on multi-core CPUs
Use Case Example	Running Chrome, Notepad, and Spotify simultaneously	Chrome: each tab/thread handles different tasks like rendering, JS

Context Switching

Thread Context Switching

- Switches from one thread to another **within the same process**.
- Fast and lightweight.
- **Does NOT** switch memory address space (since threads share memory).
- OS only needs to save registers, stack pointer, and program counter.

Process Context Switching

- Switches from one **process** to another.
- Slower and heavier.
- **DOES** involve switching memory address space.

- OS must save:
    - Memory map
    - Process state
    - Cache (which gets flushed)

□ Thread Scheduling

- Threads are scheduled by the OS based on:
    - Priority
    - Availability of CPU time slices
- Even though all threads belong to one process, they can run concurrently (true parallelism on multi-core CPUs or via time-sharing on a single core).

□ Real-Life Example

Scenario: Using Microsoft Word

- You are **typing** (one thread).
- **Spell-check** is happening automatically (second thread).
- **Auto-save** runs in the background (third thread).
- All these threads belong to the same Word process and share memory.

Scenario: On Your Computer

- You're running Chrome (one process), Zoom (second process), and Spotify (third process).
- Each process is managed by OS using multitasking.

□ Summary

- **Multi-Tasking** = Running multiple **processes** = Heavyweight.
- **Multi-Threading** = Running multiple **threads in one process** = Lightweight.
- Threads **share memory**, processes **don't**.
- Context switching is **faster** for threads than for processes.
- Multi-threading increases performance and responsiveness, especially in real-time and user-interactive applications.

LEC-4: Components of OS

1. What is an Operating System (OS)?

An **Operating System (OS)** is system software that acts as an interface between the **user**, **application software**, and **hardware**.

---

2. Two Main Parts of OS

A. Kernel

- The **core component** of the OS.
- Directly interacts with the **hardware**.
- Performs **low-level** tasks like:
    - Process management
    - Memory management
    - File management
    - I/O control
- **Loads first** at system startup.
- Known as the **"heart of the OS."**

B. User Space

- Where **applications** run (e.g., browsers, media players).
- Apps **do not** directly access hardware.
- Interacts with the kernel for services.
- Includes:
    - **GUI (Graphical User Interface)** – e.g., windows, icons.
    - **CLI (Command Line Interface)** – e.g., terminal, command prompt.

---

3. What is a Shell?

- A **shell** is a **command interpreter**.
- It takes **user commands**, interprets them, and forwards them to the **kernel**.
- Can be GUI-based or CLI-based.

---

4. Functions of Kernel

1. Process Management

- Manages all system and user processes.
- Responsibilities:
  - Creating and deleting processes
  - Scheduling tasks on CPU
  - Suspending/resuming processes
  - Managing inter-process communication (IPC)
  - Handling synchronization and deadlocks

2. Memory Management

- Manages the RAM (main memory).
- Responsibilities:
  - Allocating/deallocating memory
  - Keeping track of memory usage
  - Preventing unauthorized access (protection)

3. File Management

- Manages data storage in files/directories.
- Responsibilities:
  - Create/delete files and directories
  - Organize and map files to storage
  - Manage file permissions
  - Support backup and restore

4. I/O Management

- Manages Input/Output devices (keyboard, mouse, printer).
- Includes:
  - **Buffering**: Temporary storage during transfer (e.g., YouTube video buffering)
  - **Caching**: Storing frequent data for faster access (e.g., web cache)
  - **Spooling**: Queuing data for slow devices (e.g., print spooler)

---

5. Types of Kernels

1. Monolithic Kernel

- All OS services are inside the kernel.
- Fast performance due to fewer mode switches
- Bulky, less secure (a single crash can bring down the entire system)
- **Examples**: Linux, Unix, MS-DOS

Feature	Monolithic Kernel
Size	Large
Speed	Fast
Reliability	Low
Memory Usage	High

□ 2. Microkernel

- Only core functionalities (like memory & process management) in the kernel.
- Other services (like file system, drivers) run in **user space**.
- More secure and stable
- Slower due to more user/kernel communication
- **Examples**: MINIX, L4 Linux, Symbian OS

Feature	Microkernel
Size	Small
Speed	Slow
Reliability	High
Modularity	High

□ 3. Hybrid Kernel

- Mix of Monolithic and Microkernel.
- Tries to combine **performance** with **modularity**.
- File management in user space; core tasks in kernel.
- **Examples**: Windows NT/7/10, MacOS

Feature	Hybrid Kernel
Performance	Good

Feature	Hybrid Kernel
Modularity	Medium-High
Reliability	High

---

4. Nano/Exo Kernel

- Even smaller than microkernels.
- Focuses only on **hardware abstraction**.
- Rare in general-purpose systems. Used in embedded and research systems.

---

6. User Mode vs Kernel Mode Communication

How do User Space & Kernel Space communicate?

Through Inter-Process Communication (IPC):

- Required when processes need to **coordinate or exchange data**.
- Memory isolation means direct sharing isn't allowed; use IPC.

IPC Mechanisms:

1. **Shared Memory**:
    - Both processes access a **common memory area**.
    - Fast, but needs synchronization (e.g., mutex/semaphore).
2. **Message Passing**:
    - Processes send and receive messages via **queues or pipes**.
    - Safer but slower than shared memory.

□ **Summary Table**

Component	Description
Kernel	Core part of OS, manages hardware and low-level operations
User Space	Where apps run; interacts with kernel via system calls

Component	Description
Shell	Interface that takes user commands and communicates with the kernel
IPC	Communication between processes (shared memory / message passing)
Monolithic Kernel	Fast but heavy and less modular
Microkernel	Small and modular but slower
Hybrid Kernel	Combines best of both worlds

LEC-5: System Calls

How Do Applications Interact with the Kernel?

 Through System Calls

System calls are the **gateway between user space (apps)** and **kernel space (OS core)**. They allow user applications to request services from the kernel that they are **not allowed to perform directly** due to **security and stability** reasons.

What Happens Internally When You Run a Command Like mkdir laks?

1. **User Executes mkdir laks (in User Space)**
   - You type this in a shell or terminal.
2. **Wrapper Function Calls System Call**
   - mkdir is not the system call itself—it is a **wrapper** that **internally invokes a system call**, such as mkdir() or sys_mkdir.
3. **System Call Enters Kernel Space**
   - Control shifts from **user mode to kernel mode** using a **software interrupt** or **trap**.
   - The **file management module** in the kernel is now responsible for creating the directory.
4. **Kernel Performs the Task**
   - It accesses the **file system**, creates the directory, and updates necessary tables.
5. **Control Returns to User Space**
   - The operation completes, and execution returns back to the user shell or application.

 User Mode to Kernel Mode Transition

- Transitions happen via **software interrupts/traps**.
- A **system call** acts as a controlled entry point into the kernel.

 Why System Calls Are Needed?

- User apps do **not have permissions** to:
  - Access I/O devices directly.
  - Allocate or free kernel memory.
  - Create or terminate system-level processes.
  - Access hardware or interact with other processes directly.

 So, **system calls** allow users to request these services **safely and securely**.

 Types of System Calls

1) Process Control

Handles process creation, execution, and termination.

- end, abort, load, execute
- create_process, terminate_process
- get/set process attributes
- wait for time/event, signal event
- allocate/free memory

2) File Management

Handles operations on files and directories.

- create, delete, open, close
- read, write, reposition
- get/set file attributes

3) Device Management

Handles interaction with hardware devices.

- request/release device
- read/write, reposition
- attach/detach devices

- get/set device attributes

4) Information Maintenance

Deals with system-level information.

- get/set time or date
- get/set system data
- get/set process/file/device attributes

5) Communication Management

Used for **inter-process communication (IPC)** and networking.

- create/delete communication connection
- send/receive messages
- transfer status
- attach/detach remote devices

Examples of System Calls in Windows vs Unix

Category	Windows	Unix/Linux
Process Control	CreateProcess(), ExitProcess()	fork(), exit(), wait()
File Management	CreateFile(), ReadFile()	open(), read(), write()
Device Management	ReadConsole(), WriteConsole()	ioctl(), read(), write()
Information	GetCurrentProcessID(), Sleep()	getpid(), sleep()
Communication	CreatePipe(), MapViewOfFile()	pipe(), shmget(), mmap()

LEC-6: What happens when you turn on your computer?

i. Power On (PC On)

- When you press the **power button**, electricity is supplied to the computer.
- The **Power Supply Unit (PSU)** sends power to all components (CPU, RAM, motherboard, etc.).

---

ii. CPU Initialization & Firmware Execution

- The **CPU** (Central Processing Unit) is the brain of the computer. It **initializes itself** and looks for a **firmware program**.

BIOS or UEFI:

- The firmware is typically stored in a **ROM chip** on the motherboard.

BIOS (Basic Input Output System):

- Traditional firmware found on older PCs.
- It's a small program that helps start the system by performing basic operations.
- BIOS is non-volatile—it stays stored even when the computer is off.

UEFI (Unified Extensible Firmware Interface):

- Modern replacement for BIOS.
- Offers a **graphical interface**, **mouse support**, **larger disk support**, and **faster boot times**.
- Can include features like **Intel Management Engine** for remote diagnostics, firmware updates, etc.

---

iii. POST (Power-On Self Test)

- The CPU runs the BIOS/UEFI, which starts the **POST process**:
  - **POST** is a diagnostic test to check if hardware components like RAM, keyboard, storage, etc., are connected and working.
  - If something is missing or faulty (e.g., RAM not installed), an **error is displayed**, and the boot process is halted.

---

iv. BIOS/UEFI Loads Bootloader

- Once POST is successful, BIOS/UEFI needs to load the **Operating System**.

What it does:

- It **searches storage devices** (SSD, HDD, USB) for a special area:
  - **Legacy BIOS**: Looks at **MBR (Master Boot Record)**.
  - **UEFI**: Looks at the **EFI System Partition**.

MBR (Master Boot Record):

- A tiny program located at the **first sector of the disk (sector 0)**.
- Contains code to load the OS's **bootloader**.

The BIOS/UEFI hands over control to the **bootloader**.

---

v. Bootloader Executes and Boots the OS

The **bootloader** is the bridge between the firmware and the OS kernel.

What the bootloader does:

1. **Initializes OS Kernel** (Windows, Linux, macOS).
2. Transfers control to the kernel.
3. Kernel initializes the rest of the OS (drivers, memory, user processes).

Examples of Bootloaders:

OS	Bootloader
Windows	bootmgr.exe (Windows Boot Manager)
Linux	GRUB (GRand Unified Bootloader)
Mac	boot.efi

Once the **kernel** is loaded:

- It sets up **user space** (your desktop, login screen, background services).
- Now your system is ready for **user interaction**.

Lec-7: 32-Bit vs 64-Bit OS

1. Definition of Bit Architecture

- The **bit-size** of an operating system or processor defines the amount of data it can handle and the size of memory addresses it can manage.

Bit Size	What it Means
32-bit	Processes 32 bits (4 bytes) of data at a time.
64-bit	Processes 64 bits (8 bytes) of data at a time.

---

2. Memory Addressing and RAM Limits

OS Type	RAM Limit
32-bit	~4 GB max
64-bit	~17 billion GB (theoretical)

- A **32-bit OS** can only use **up to 4 GB** of RAM efficiently.
- A **64-bit OS** can use **way more RAM** (Windows versions support 128 GB to 2 TB depending on edition).

---

3. CPU Instruction Processing

- **32-bit CPU**: Can execute **4 bytes (32 bits)** in a single instruction cycle.
- **64-bit CPU**: Can execute **8 bytes (64 bits)** in one cycle.

The more bits processed at once, the **faster and more efficient** the CPU is for large calculations or data-intensive tasks.

---

4. Advantages of 64-bit OS over 32-bit OS

a. Addressable Memory

- **32-bit**: Limited to 2^{32} memory addresses = 4 GB max RAM.
- **64-bit**: Can access 2^{64} memory addresses = over 17 billion GB of memory.

This is especially important for applications like video editing, gaming, machine learning, etc.

---

b. Resource Usage

- Adding more RAM to a 32-bit OS doesn't improve performance beyond 4GB.
- A **64-bit OS** can **fully utilize additional RAM**, resulting in **better multitasking, faster data processing**, and smoother system performance.

---

c. Performance

- **Registers** are small memory locations in the CPU.
- Larger registers in 64-bit CPUs allow:
    - Bigger numbers in calculations.
    - Faster handling of large data sets.
    - Fewer instructions to process big tasks.

□ Example: A 64-bit system can multiply two large numbers in **one cycle**, while a 32-bit system may require **multiple cycles** to break it into parts.

---

d. Compatibility

CPU Type	Can Run 32-bit OS?	Can Run 64-bit OS?
32-bit CPU	□ Yes	□ No
64-bit CPU	□ Yes	□ Yes

- A **64-bit CPU** is backward compatible, i.e., it can run **both 32-bit and 64-bit OS** and applications.
- A **32-bit CPU** can only run **32-bit OS**.

---

e. Better Graphics Performance

- Many graphics operations require handling large numbers (pixel data, shading, lighting, etc.).
- A 64-bit system can handle **8-byte graphics calculations** directly, leading to:
    - **Faster rendering**
    - **Smoother visuals**
    - **Enhanced gaming or 3D application performance**

---

5. Use Cases and Recommendations

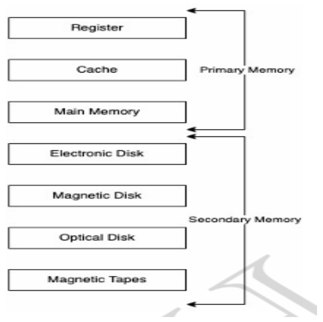
Use Case	Recommended OS
Basic browsing, word processing	32-bit (if hardware is very old)
Gaming, content creation, programming, multitasking	64-bit

Nowadays, **64-bit OS is standard** for almost all modern systems.

□ Summary Table: 32-bit vs 64-bit

Feature	32-bit	64-bit
Max RAM Usage	~4 GB	Theoretically 17.2 Billion GB
Register Size	32-bit	64-bit
Data per Instruction Cycle	4 bytes	8 bytes
Compatibility	Only 32-bit OS	Both 32-bit & 64-bit OS
Performance	Lower for modern apps	Higher and scalable
Graphics	Limited	Enhanced & faster
Ideal For	Older systems	Modern PCs, high-performance tasks

Lec-8: Storage Devices Basics



What are the different memory types present in a computer system?

A computer system uses a **memory hierarchy** that balances **speed, cost, and capacity** to efficiently store and access data. These are:

---

1. Register

- **Location**: Inside the **CPU** itself.
- **Speed**: **Fastest** memory in the computer system.
- **Size**: **Smallest** – stores a few bytes only (typically 32 to 64 bits).
- **Function**:
  - Holds **data**, **instructions**, or **addresses** that are **immediately** required by the CPU.
  - Helps in **arithmetic, logic, and control** operations.

 Example: While adding two numbers, the CPU loads the numbers into registers, performs the operation, and stores the result in a register.

---

2. Cache Memory

- **Location**: Close to the CPU core (either **inside** or **between CPU and RAM**).
- **Purpose**: Speeds up data access by storing **frequently used data/instructions**.
- **Types**:
  - **L1 Cache**: Closest to CPU (fastest but smallest).
  - **L2 Cache**: Bigger but slower.
  - **L3 Cache**: Shared across CPU cores (larger, slower).
- **Function**: Reduces CPU access time to main memory.

---

3. Main Memory (Primary Memory or RAM)

- **Full Form**: **Random Access Memory**
- **Location**: External to CPU, but directly accessible by it.
- **Purpose**: Temporarily stores programs and data being used.
- **Volatility**: **Volatile** – data is lost when power is turned off.
- **Speed**: Slower than cache, but faster than secondary storage.

RAM is where your applications like Chrome, Word, or a game **run while in use**.

---

4. Secondary Memory

- **Examples**: HDD, SSD, USB drives, DVDs.
- **Purpose**: **Permanently stores data** like OS, software, files, media, etc.
- **Volatility**: **Non-volatile** – retains data without power.
- **Speed**: Much slower compared to registers/cache/RAM.
- **Capacity**: Much **larger** than primary memory.

□ Comparison of Memory Types

Feature	Register	Cache	Main Memory (RAM)	Secondary Memory
Location	Inside CPU	Near CPU	Motherboard	Disk/External
Speed	Fastest	Very Fast	Moderate	Slow
Size	Few bytes	KB to MB	GBs	GBs to TBs
Volatility	Volatile	Volatile	Volatile	Non-volatile
Cost	Most Expensive	Expensive	Moderate	Cheapest
Access Time	1 ns	2–10 ns	50–100 ns	ms (milliseconds)
Function	Execute CPU instructions	Temporary fast access	Active data	Permanent storage

---

□ Detailed Comparison

---

1. Cost

- **Registers** are the **most expensive** because:
  - They use **high-performance semiconductors**.
  - They are built directly into the CPU with high precision.
- **Primary memory (RAM, Cache)** is **more expensive** than secondary.
- **Secondary memory** (like HDD/SSD) is **cheaper** due to mass production and lower performance requirements.

---

2. Access Speed

- Fastest to slowest:

Registers > Cache > RAM > SSD > HDD

- CPU accesses **registers in nanoseconds**, but **HDDs in milliseconds**.
- Hence, higher levels of memory are used to **bridge the speed gap** between CPU and disk.

---

3. Storage Size

- **Registers** store **very little** data (a few bytes).
- **Cache** ranges from **kilobytes to megabytes**.
- **RAM** can go from **4 GB to 64 GB** in personal computers.
- **Secondary storage** can be **terabytes** in size.

---

4. Volatility

- **Volatile** memory loses data when power is off:
  - Registers, Cache, RAM.
- **Non-volatile** memory retains data:
  - Hard drives, SSDs, flash drives.

Lec-9: Introduction to Process

1. What is a Program?

- A **program** is a **set of instructions** written in a programming language (like C, Python, Java).
- After **compilation**, it becomes **machine code**.
- It is a **passive** entity — meaning it's just a file stored on disk, **not currently running**.

Example: notepad.exe stored on disk is a program.

---

2. What is a Process?

- A **process** is an **active instance** of a program — it is a **program under execution**.
- Once a program is **loaded into memory** and starts executing, it becomes a process.
- A process is **dynamic** and has resources like CPU time, memory, I/O, etc.

Example: When you **double-click on Notepad**, it becomes a **process** and starts running.

---

3. How does the OS create a process?

The operating system (OS) transforms a **program into a process** in the following steps:

 Steps to create a process:

a. Load Program & Static Data into Memory

- The OS loads the **compiled program** and **read-only/static data** into the system's **RAM**.

b. Allocate Runtime Stack

- The **stack** is created for the process. It stores **function calls, parameters, return addresses, and local variables**.

c. Allocate Heap Memory

- The **heap** section is reserved for **dynamic memory allocation** (e.g., variables created using malloc() in C).

d. Handle I/O Tasks

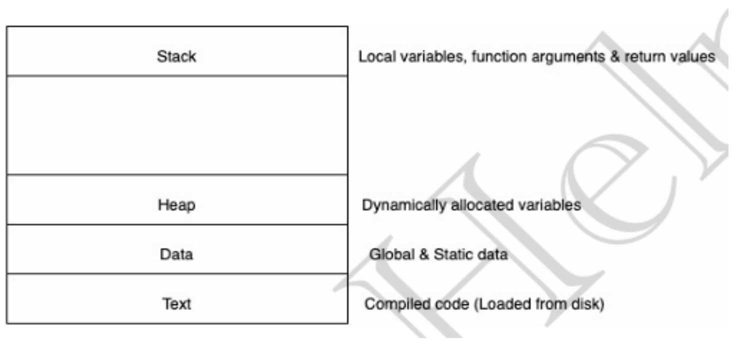
- Setup **I/O connections** like file descriptors, standard input/output, etc.

e. OS Handover to main()

- Finally, the OS passes control to the **starting point** of the program (like main() function), and the **process begins execution**.

---

4. Architecture of a Process


Stack
Local variables, function arguments & return values
Heap
Dynamically allocated variables
Data
Global & Static data
Text
Compiled code (Loaded from disk)

A process in memory is divided into 4 sections:

Section	Purpose
Text	Executable code of the program.
Data	Static and global variables.
Heap	Dynamically allocated memory during runtime.
Stack	Function call stack (local variables, return addresses).

---

5. Attributes of a Process

These are **properties/information** the OS uses to **manage** each process.

a. Unique Identification

- Every process is assigned a unique **Process ID (PID)**.

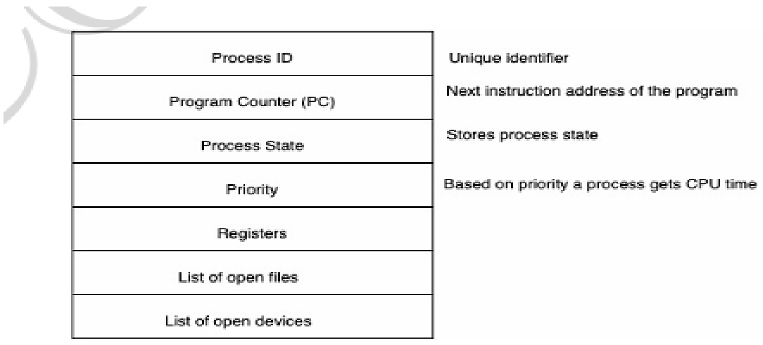
b. Process Table

- OS uses a **data structure** called the **Process Table** to keep track of all active processes.

c. Process Control Block (PCB)

- Each entry in the process table is a **PCB**.
- The **PCB** contains **all the essential information about a process**.

6.PCB (Process Control Block) Structure



A **PCB** is like an **identity card** + **status record** for each process.

What it stores:

- **Process ID** – Unique number for each process.
- **Process State** – Ready, Running, Waiting, etc.
- **Program Counter** – Address of the next instruction to execute.
- **CPU Registers** – Values of registers during context switch.
- **Memory Management Info** – Pointers to stack, heap, page tables.
- **Priority** – Scheduling priority.
- **I/O Status** – List of open files, I/O devices assigned.

 Context Switching & Registers in PCB

- When a process is running and its **CPU time slice ends**, the OS must **pause it** and schedule another.
- This requires **saving the state** of the current process.
- The CPU's **register values (including the program counter)** are **saved into the PCB**.
- When the process is scheduled to run again, its **register values are restored** from the PCB into the CPU.

This mechanism is what enables **multitasking** in modern operating systems.

Lec-10: Process States | Process Queues

1. Process States

As a process goes through its life cycle, it transitions through different **states**. These transitions are managed by the **Operating System (OS)**.

Here are the main **process states**:

State	Description
New	The process is being created. OS is in the process of loading the program into memory.
Ready	The process is loaded into RAM and is waiting to be assigned CPU time.
Running	The CPU is currently executing the instructions of the process.

State	Description
Waiting	The process is waiting for some **I/O operation** (like reading a file, network data).
Terminated	The process has finished execution and its PCB is removed from the process table.

Example Lifecycle:

New → Ready → Running → (Wait or Ready again) → Terminated

---

2. Process Queues

Operating Systems maintain **different queues** to manage the movement of processes across different states:

---

a. Job Queue

- Contains all the processes that are in the **New** state.
- These are stored in **secondary memory** (like HDD or SSD).
- Managed by **Long-Term Scheduler (LTS)**:
    - Picks processes from job queue.
    - Loads them into **main memory** to move them to the **Ready state**.

---

b. Ready Queue

- Contains processes that are in **Ready state**.
- Stored in **main memory (RAM)**.
- Managed by **Short-Term Scheduler (STS)** or **CPU Scheduler**:
    - Picks one process from the ready queue and **dispatches** it to CPU for execution.

---

c. Waiting Queue

- Holds processes that are **waiting for I/O operations** to complete.
- When I/O is completed, they are moved back to the **Ready queue**.

---

3. Degree of Multi-Programming

- It refers to the **number of processes in main memory** (i.e., how many processes are "active" at once).
- Controlled by the **Long-Term Scheduler (LTS)**:
    - If the degree is too high → system can become overloaded.
    - If too low → CPU might be idle.

Higher degree of multi-programming = Better CPU utilization (but up to an optimal point).

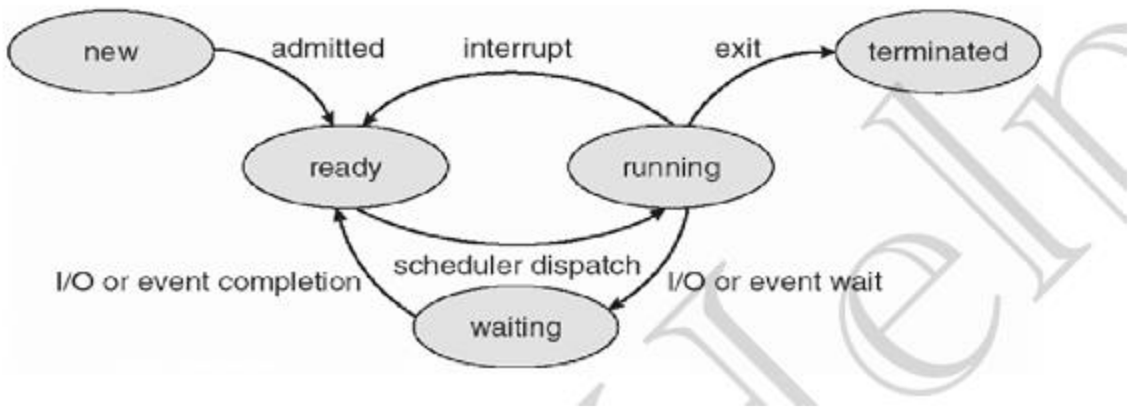
---

4. Dispatcher

- A special module in the OS responsible for **giving control of the CPU to the process selected by the STS**.

Key roles:

- **Context switching**: Saves the state of the old process and loads the state of the new one.
- **Switching modes**: From user mode to kernel mode and vice versa.
- **Jump to the correct program counter** to resume execution.



LEC-11: Swapping | Context-Switching | Orphan process | Zombie process

1. Swapping

Definition:
Swapping is a memory management technique where a **process is temporarily moved from main memory (RAM) to secondary storage (disk)** and brought back when needed. This helps **free up space** in RAM for other processes.

How It Works:

- Managed by the **Medium-Term Scheduler (MTS)**.
- Helps control the **degree of multi-programming** by removing (swap-out) and later reloading (swap-in) processes.
- When system memory is **overloaded**, swapping is used to maintain efficiency.

Steps:

1. A process is **swap-out** from RAM to disk.
2. RAM space becomes available for other processes.
3. Later, the process is **swap-in** back from disk to RAM.
4. It resumes execution from **exact point** where it was paused.

Purpose:

- Improve **process mix** (better variety of processes in memory).
- Handle **memory overcommitment**.
- Maintain good **CPU utilization** by balancing active processes.

---

2. Context Switching

Definition:
Context switching is the process of **saving the state of a currently running process** and **loading the saved state of the next scheduled process**.

Steps Involved:

1. **Save** current process state in its **Process Control Block (PCB)**.
2. **Load** the next process's saved state from its PCB.
3. CPU resumes execution of the new process.

Key Points:

- Performed by the **Operating System Kernel**.
- It is a **pure overhead** (no real work is done during switching).
- Time taken depends on:
    - CPU and memory speed,
    - Number of registers to be saved/restored.

Why it's important:

- Enables **multitasking**.

- Helps OS implement **CPU scheduling**.
- Allows **interrupt handling** and **preemptive execution**.

---

3. Orphan Process

Definition:
An **orphan process** is a child process **whose parent process has terminated** before the child finishes its execution.

How it is handled:

- Such processes are **adopted by the 'init' process** (process ID = 1).
- init is the **first process** started by the OS at boot time.
- It **monitors and cleans up** orphan processes to avoid resource leaks.

Example:
Parent starts Child
Parent exits early
Child is now orphan
Init adopts the child

---

4. Zombie Process / Defunct Process

Definition:
A **zombie process** is a process that has **completed execution**, but **still has an entry in the process table**.

Why does this happen?

- When a child process finishes execution, it still leaves an **exit status** for the parent to read.
- The parent process must call **wait() system call** to collect this status.
- Until this is done, the child remains in the **"zombie" state**.

Technical Details:

- Zombie processes don't use RAM or CPU (they are dead), but they **occupy a slot** in the **process table**.
- Once the parent **reads the exit status**, the OS **cleans it up** (this is called **reaping**).

Risks:

- Too many zombie processes can **fill up the process table**, leading to system instability.

LEC-12: Intro to Process Scheduling | FCFS | Convoy Effect

1. Process Scheduling

Definition:
Process scheduling is the activity of the operating system that **decides which process in the ready queue gets the CPU next**.

Purpose:

- Core concept behind **multi-programming**.
- Increases **CPU utilization** by **switching** CPU among multiple processes.
- Ensures multiple processes remain active even if some are waiting (e.g., for I/O).

---

2. CPU Scheduler (Short-Term Scheduler - STS)

- **Responsibility:** Selects one process from the **ready queue** for CPU allocation **when the CPU becomes idle**.
- Makes the **decision quickly and frequently** (milliseconds).
- Ensures **efficient use of the CPU** by not letting it stay idle.

---

3. Non-Preemptive Scheduling

Definition:
Once a process is given the CPU, it **retains control** until it:

- Terminates, or
- Voluntarily switches to a wait state (e.g., for I/O).

Drawbacks:

- Can lead to **starvation**: Short processes may wait a long time if a long process is running.
- **Low CPU utilization** if the current process is slow or blocks often.

---

4. Preemptive Scheduling

Definition:
The OS can **take the CPU away** from a running process after:

- Time quantum (time slice) expires,
- Or the process is moved to waiting state.

Advantages:

- Better **CPU utilization**.
- Reduces **starvation** by allowing fair CPU access.
- Ideal for **time-sharing** and **real-time systems**.

---

5. Goals of CPU Scheduling

Goal	Meaning
Maximum CPU Utilization	Keep the CPU busy all the time.
Minimum Turnaround Time (TAT)	Complete tasks in shortest time from arrival to finish.
Minimum Waiting Time (WT)	Reduce time spent in the ready queue.
Minimum Response Time	Respond to user interaction as quickly as possible.
Maximum Throughput	Maximize number of processes completed per time unit.

---

6. Key Terminologies

Term	Description
Arrival Time (AT)	Time when a process enters the **ready queue**.
Burst Time (BT)	Time required by the process for execution on the CPU.

Term	Description
Completion Time (CT)	The time at which a process **finishes execution**.
Turnaround Time (TAT)	Total time from arrival to completion. **TAT = CT - AT**
Waiting Time (WT)	Time spent waiting in the ready queue. **WT = TAT - BT**
Response Time	Time between **arrival** and **first CPU allocation**.

---

7. FCFS (First Come First Serve) Scheduling

Definition:
In FCFS, the process that **arrives first** is served first, just like a **queue in a bank**.

Characteristics:

- **Non-preemptive**.
- Simple and easy to implement using **FIFO queue**.
- **Execution order** is based only on **arrival time**.

Drawback: Convoy Effect

- **Convoy Effect** occurs when a **long burst time process** arrives first.
- All the **short processes** have to wait until the long process completes.
- Leads to:
    - **Poor CPU utilization**
    - **Long average waiting time**
    - **Inefficient resource usage**

Example of Convoy Effect:

Process	Arrival Time	Burst Time
P1	0	10
P2	1	2
P3	2	1

- P1 executes first (10 units), so P2 and P3 wait unnecessarily.
- Even though P2 and P3 are short, they get delayed by P1 → **Convoy Effect**.

LEC-13: CPU Scheduling | SJF | Priority | RR

1. Shortest Job First (SJF) Scheduling

a. SJF Non-Preemptive:

- The process with the **shortest burst time (BT)** is selected next for CPU.
- Process runs to completion once it gets CPU.
- Requires an **estimate of BT** for each process before scheduling.
- Accurate BT estimation is difficult (often based on past history or heuristics).
- This version can suffer from the **Convoy Effect** if the first process has a large BT, delaying shorter jobs.
- **Starvation** can occur for longer processes if short jobs keep arriving.
- Scheduling depends on both **Arrival Time (AT)** and **Burst Time (BT)**.

b. SJF Preemptive (also called Shortest Remaining Time First - SRTF):

- The CPU can be taken from the current process if a new process arrives with a **shorter remaining BT**.
- Has **less starvation** than non-preemptive SJF.
- **No Convoy Effect** because shorter jobs get scheduled immediately.
- Generally gives a **lower average waiting time** than non-preemptive SJF and FCFS.
- Efficiently prioritizes shorter jobs without significant delays.

---

2. Priority Scheduling

a. Priority Scheduling Non-Preemptive:

- Each process is assigned a **priority when created**.
- The CPU is given to the process with the **highest priority** (lowest numeric value or highest priority value depending on implementation).
- **SJF is a special case** of priority scheduling where priority is inversely proportional to BT (shorter BT = higher priority).
- Once CPU is allocated, process runs till completion or wait state.

b. Priority Scheduling Preemptive:

- If a new process arrives with **higher priority** than the currently running process, it **preempts** the CPU.
- Can cause **indefinite waiting (starvation)** for lower priority processes.

- **Starvation problem** exists for both preemptive and non-preemptive priority scheduling.

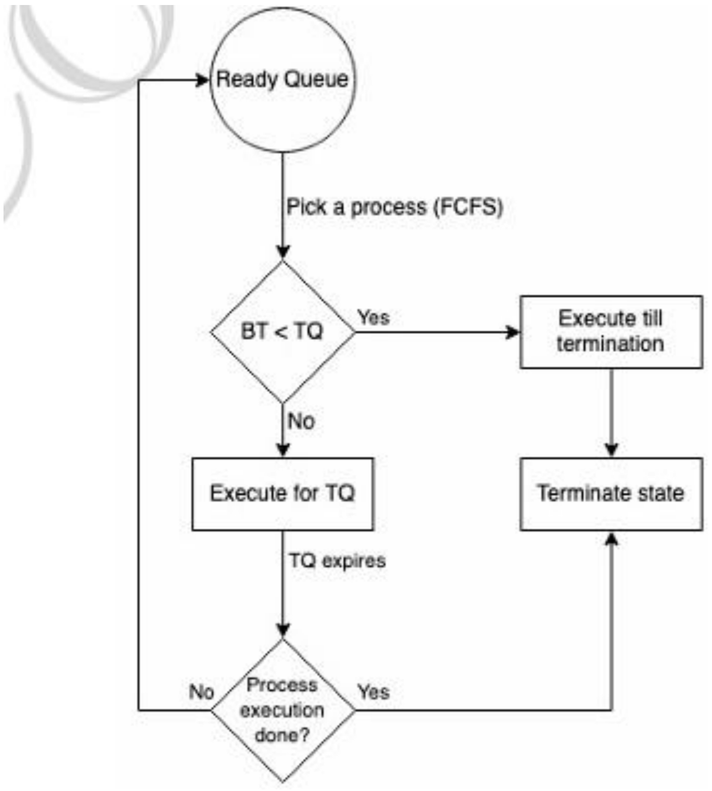
Solution to Starvation: Aging

- Gradually **increase the priority** of waiting processes over time.
- Example: Increase priority by 1 every 15 minutes.
- Ensures all processes eventually get CPU time.

---

3. Round Robin (RR) Scheduling

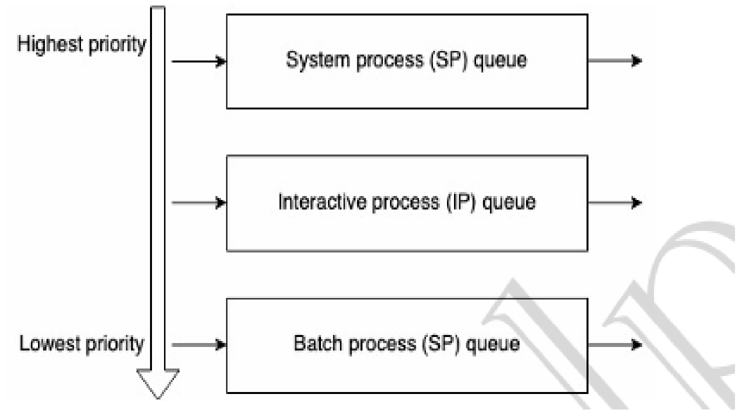
- **Most popular** scheduling method for **time-sharing systems**.
- Similar to FCFS but **preemptive**.
- Processes are assigned CPU for a fixed time slice called **Time Quantum (TQ)**.
- Criteria for scheduling: **Arrival Time (AT)** + Time Quantum.
- Unlike SJF or Priority, RR does **not depend on Burst Time**.
- Guarantees **very low starvation** because no process waits indefinitely.
- **No Convoy Effect** since CPU is shared fairly among processes.
- Simple and **easy to implement**.
- If Time Quantum is very small, the CPU spends too much time in **context switching**, causing overhead.
- If Time Quantum is very large, RR behaves like FCFS (non-preemptive).



LEC-14: MLQ | MLFQ

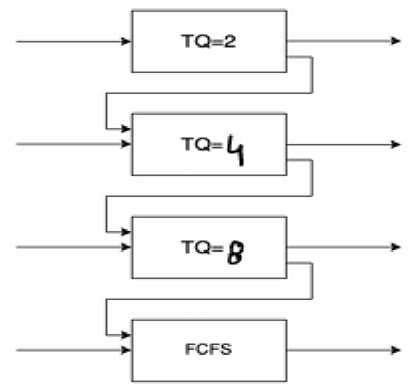
1. Multi-Level Queue Scheduling (MLQ)

- The **ready queue is divided into multiple separate queues**, each representing a different priority or class of processes.
- **Each process is permanently assigned to one queue**, based on a fixed attribute such as:
  - Process type (system, interactive, batch)
  - Memory size
  - Process priority or class
- **Queues have their own scheduling algorithms** — for example:
  - System processes queue (SP) → Round Robin (RR)
  - Interactive processes queue (IP) → Round Robin (RR)
  - Batch processes queue (BP) → First Come First Serve (FCFS)
- **Types of processes:**
  - **System processes**: Created by the OS; highest priority.
  - **Interactive processes**: Need user input or I/O (foreground).
  - **Batch processes**: Run silently without user input (background).
- **Scheduling among these queues uses fixed priority preemptive scheduling**:
  - Higher priority queues are served before lower priority queues.
  - If a higher priority queue process arrives, it **preempts** the currently running lower priority process.
- **Example scenario:**
  - If an interactive process (high priority) arrives while a batch process (low priority) is running, the batch process will be preempted immediately.
- **Problems with MLQ:**
  - **Starvation for lower priority queues**: Processes in lower priority queues may never get CPU time if higher priority queues keep getting new processes.
  - **Convoy effect**: If a long process is running in a high priority queue, shorter processes in lower priority queues get delayed.



2. Multi-Level Feedback Queue Scheduling (MLFQ)

- Similar to MLQ, but **processes can move between queues** based on their CPU usage and behavior — this is called "feedback."
- **Purpose:** Separate processes based on their burst time characteristics dynamically.
    - Processes using a lot of CPU time get moved **down to lower priority queues**.
    - I/O-bound or interactive processes tend to remain in **higher priority queues** (since they usually need less CPU).
- **Aging mechanism:** If a process waits too long in a lower priority queue, it can be moved **up to a higher priority queue**.
    - This helps **prevent starvation** that exists in MLQ.
- **Advantages of MLFQ:**
    - More **flexible** than MLQ.
    - Better handles a mix of CPU-bound and I/O-bound processes.
    - Can be **customized/configured** based on system needs (e.g., number of queues, scheduling policies in each queue, aging intervals).
- **Typical MLFQ design:**
    - Processes start in the highest priority queue.
    - If they use too much CPU time, they move down to a lower priority queue.
    - If they wait too long without getting CPU, they move up to prevent starvation.
    - Each queue might use a different scheduling algorithm (like RR for high priority, FCFS for low priority).



LEC-15: Introduction to Concurrency

1. What is Concurrency?

- **Concurrency** means executing multiple sequences of instructions *at the same time*.
- It occurs in operating systems when **multiple processes or threads run in parallel**, giving the appearance of simultaneous execution.
- This is essential for efficient utilization of CPU resources and better system performance.

2. What is a Thread?

- A **thread** is a *single sequence stream* within a process.
- It is an **independent path of execution inside the same process**.
- Threads are sometimes called **light-weight processes** because they share the process resources but have their own execution flow.
- Threads are used to achieve **parallelism** by splitting tasks of a process into independent, concurrently executing sub-tasks.

Example:
In a web browser with multiple tabs or a text editor:

- When typing, one thread handles the keystrokes.
- Another thread performs spell checking.
- Another manages formatting.
- Yet another thread saves the document.
  All these threads run concurrently, improving responsiveness.

---

3. Thread Scheduling

- Threads are scheduled for CPU time **based on their priority**.
- Even though multiple threads are running inside a process, the operating system allocates **processor time slices** to each thread.
- Thread scheduling is managed by the OS's **Thread Scheduling System (STS)**.

---

4. Thread Context Switching

- When switching from one thread to another, the OS:
  - Saves the **current state** of the running thread.
  - Switches to the **next thread** in the same process.
- This switch **does not include changing the memory address space** because threads share the same process memory.
- Only the thread-specific data such as:
  - Program Counter (PC)
  - Registers
  - Stack

    are switched.
- This makes thread context switching **faster than process switching**.
- The CPU cache state is preserved, which further improves performance.

---

5. How Does Each Thread Access the CPU?

- Each thread maintains its own **program counter (PC)** which keeps track of the next instruction to execute.
- The OS uses a **thread scheduling algorithm** to decide which thread to run next.
- The CPU fetches instructions from the address pointed to by the thread's PC and executes them.

---

6. Context Switching Triggers

- Threads are switched based on **I/O events** or **time quantum (TQ)** expiration.
- The OS maintains a **Thread Control Block (TCB)**, similar to the Process Control Block (PCB), to store thread state information during context switching.

---

7. Does Multi-threading Benefit a Single CPU System?

- No, a single CPU system **does not gain from multi-threading** because:
  - Only one thread can execute at a time.
  - Threads have to perform context switches, which adds overhead.
  - So, the benefit of concurrency is limited on a single CPU without parallel execution.

---

8. Benefits of Multi-threading

- **Responsiveness:** Applications remain responsive because threads can run independently (e.g., UI thread and background task thread).
- **Resource Sharing:** Threads share resources like memory and open files within the same process, making resource sharing efficient.
- **Economy:** Creating and switching between threads is cheaper than creating and switching between processes.
  - Creating a new process requires allocating separate memory and resources, which is costly.
- **Better Multiprocessor Utilization:** Threads can run on multiple processors/cores simultaneously, greatly improving performance on multiprocessor systems.

LEC-16: Critical Section Problem and How to address it

1. Why is Process Synchronization Important?

- When multiple processes or threads run concurrently, they often need to access **shared data** (like variables, files, or devices).
- To maintain **data consistency** and avoid errors, processes must be synchronized properly.
- **Process synchronization techniques** help prevent conflicts and ensure correct results when accessing shared resources.

---

2. What is a Critical Section (C.S)?

- The **Critical Section** is the part of the program where a process/thread accesses **shared resources** and performs operations like reading or writing.
- Because multiple threads/processes run concurrently, **a process can be interrupted at any time**, possibly causing inconsistent or incorrect data.
- So, **only one process should execute its critical section at a time** to avoid interference.

---

3. Major Thread Scheduling Issue: Race Condition

- A **race condition** happens when **two or more threads/processes access and try to modify shared data at the same time**.
- The exact order of execution is unpredictable because thread scheduling can switch between threads at any moment.
- Since the result depends on the timing/order of threads’ access, the program’s behavior becomes unpredictable.
- Both threads are "racing" to read/write shared data, leading to inconsistent or corrupted results.

---

4. Solutions to Race Condition

- **Atomic operations**: Make the entire critical section execute as a single atomic (indivisible) operation that cannot be interrupted.
- **Mutual exclusion using locks**: Use locks or mutexes to ensure that only one thread/process can enter the critical section at a time.
- **Semaphores**: Use signaling mechanisms like semaphores to control access and coordinate among processes.

---

5. Can a Simple Flag Variable Solve Race Condition?

- No.
  Using a simple flag to indicate whether a process is in the critical section does **not work reliably** because checking and setting the flag are not atomic operations.
  This can still lead to race conditions.

---

6. Peterson's Solution

- Peterson's algorithm is a classic **software-based solution for mutual exclusion**.
- It works by using two variables to indicate intent and turn, ensuring **mutual exclusion for exactly 2 processes/threads**.
- **Limitation:** It only works for **two processes** and is mostly of theoretical interest.

---

7. Mutexes / Locks

- **Locks** or **Mutexes (Mutual Exclusion objects)** are used to implement mutual exclusion.
- They allow only **one thread/process to enter the critical section at a time**, avoiding race conditions.
- When a thread acquires a lock, all others must wait until it releases the lock.

---

8. Disadvantages of Mutexes / Locks

- **Contention:**
  If one thread holds the lock, others must wait (busy wait or block). This can lead to performance bottlenecks.
- **Deadlocks:**
  If threads wait forever for locks held by each other, the system halts — a situation called deadlock.
- **Debugging:**
  Locks make debugging difficult because race conditions and deadlocks are often timing-dependent and hard to reproduce.
- **Starvation:**
  High-priority threads might be starved if lower-priority threads continuously acquire the lock before them.

LEC-17: Conditional Variable and Semaphores for Threads synchronization

1. Conditional Variable

- **What is it?**
  A **conditional variable** is a synchronization primitive used in multithreaded programming that **allows a thread to wait until a specific condition is true** or an event occurs.
- **How does it work?**
  - It **works together with a lock (mutex)** to ensure proper synchronization.
  - A thread must **acquire the lock** before it can wait on the conditional variable.
  - When the thread waits, it **releases the lock and goes into a waiting state**, freeing the resource for other threads.
  - Another thread, after changing some shared state, **signals or notifies** the conditional variable to indicate the condition has occurred.
  - The waiting thread is then **woken up, re-acquires the lock, and resumes execution**.
- **Why use conditional variables?**
  - To **avoid busy waiting**, where a thread wastes CPU cycles repeatedly checking a condition.
  - This mechanism makes thread synchronization more efficient by **blocking the thread** until the condition is true.
- **No contention here:**
  Because the thread releases the lock while waiting, other threads can progress without blocking on this condition variable.

---

2. Semaphores

- **What are Semaphores?**
  Semaphores are another synchronization method used to control access to shared resources in concurrent programming.
- **Key properties:**
  - Represented as an **integer value** that indicates the number of available resources.
  - Allow **multiple threads to access shared resources concurrently**, up to a limit defined by the semaphore's value.
- **Difference between Mutex and Semaphore:**
  - **Mutex (binary semaphore):** Allows **only one thread at a time** to access a single resource (value is either 0 or 1).
  - **Counting Semaphore:** Can have a value greater than 1, allowing **multiple instances** of a resource to be accessed concurrently.

---

3. Types of Semaphores

- **Binary Semaphore:**
  - Value is either 0 or 1.
  - Also called a **mutex lock** because it allows mutual exclusion for one thread at a time.
- **Counting Semaphore:**
  - Value can range over an unrestricted domain (0 to N).
  - Controls access to resources that have **multiple identical instances** (like a pool of printers or database connections).

---

4. Wait() and Signal() Operations

- **Traditional wait() (P) operation:**
  - Decreases the semaphore value.
  - If the semaphore value is positive, the thread proceeds.
  - If the semaphore value is zero or negative, the thread **busy waits** (repeatedly checks) until it becomes positive.
- **Signal() (V) operation:**
  - Increases the semaphore value.
  - If there are threads waiting, signals them to proceed.

---

5. Avoiding Busy Waiting

- To prevent CPU waste due to busy waiting, **wait() and signal() operations are modified**:
  - When a thread calls wait() and finds the semaphore value is not positive, instead of busy waiting, it **blocks itself**.
  - The blocked thread is placed in a **waiting queue** associated with the semaphore and its state is changed to **Waiting**.
  - The CPU scheduler then runs another thread.
- When another thread calls **signal()**, a blocked thread is **woken up (wakeup())**, changed from Waiting to Ready state, and placed in the ready queue to run.

Lec-18: Producer-Consumer Problem

What is the Producer-Consumer Problem?

- The **Producer-Consumer problem** (also called the **bounded-buffer problem**) is a classic example of a **synchronization problem** in concurrent programming.
- It deals with **process synchronization** when two or more processes (or threads) share a common, finite-sized buffer.

Components

- **Producer:**
    - A process or thread that **produces data items** and puts them into the buffer.
    - It cannot add data if the buffer is full.
- **Consumer:**
    - A process or thread that **consumes data items** from the buffer.
    - It cannot consume if the buffer is empty.
- **Buffer:**
    - A fixed-size queue or array where produced items are stored temporarily.
    - It acts as the shared resource between producers and consumers.

The Problem

- Both producer and consumer **share access to the buffer**, so their actions must be synchronized:
    - **Producer must wait** if the buffer is **full** (no space to add data).
    - **Consumer must wait** if the buffer is **empty** (no data to consume).
- The problem is to **avoid race conditions and ensure mutual exclusion** while accessing the buffer.
- Also, avoid **deadlocks**, **starvation**, and ensure **proper synchronization**.

Why is it Important?

- Demonstrates the need for synchronization primitives like **mutexes**, **semaphores**, and **condition variables**.
- Illustrates how to coordinate multiple threads/processes sharing resources in a controlled manner.

Common Synchronization Tools Used

- **Mutex (Lock):**
  To ensure **mutual exclusion** while accessing the buffer so only one process modifies the buffer at a time.
- **Semaphores:**
  Two counting semaphores are typically used:
    - **empty semaphore:** Counts the number of empty slots in the buffer. Initialized to the buffer size.
    - **full semaphore:** Counts the number of filled slots in the buffer. Initialized to 0.

How it works (Stepwise)

1. **Producer:**
    - Waits (decrements) on empty semaphore to check if there is space in the buffer.
    - Acquires the mutex lock to get exclusive access to the buffer.
    - Adds an item to the buffer.
    - Releases the mutex lock.
    - Signals (increments) the full semaphore to indicate new data is available.
2. **Consumer:**
    - Waits (decrements) on full semaphore to check if there is an item to consume.
    - Acquires the mutex lock to get exclusive access to the buffer.
    - Removes an item from the buffer.
    - Releases the mutex lock.
    - Signals (increments) the empty semaphore to indicate a free slot is available.

Pseudocode Example

```
semaphore empty = BUFFER_SIZE
semaphore full = 0
mutex lock

Producer() {
  while (true) {
    produce_item()
    wait(empty)        // Wait if no empty slot
    wait(lock)         // Acquire exclusive access
    add_item_to_buffer()
    signal(lock)       // Release access
    signal(full)       // Signal that item is added
  }
}

Consumer() {
  while (true) {
    wait(full)         // Wait if buffer empty
    wait(lock)         // Acquire exclusive access
    remove_item_from_buffer()
    signal(lock)       // Release access
    signal(empty)      // Signal that slot is free
    consume_item()
  }
}
```

LEC-19: Reader-Writer Problem

The **Reader-Writer Problem** is a classical **synchronization problem** that arises when multiple processes try to access **shared data** concurrently, with **readers** only reading the data and **writers** modifying it.

□ Objective:

- Allow **multiple readers** to read the shared data **at the same time**.
- Ensure that **only one writer** can write at a time.
- Prevent **readers and writers** from accessing the shared resource **simultaneously** to avoid **data inconsistency**.

□□ Processes Involved:

1. **Reader:**
   - A process that **reads** the shared data.
   - **Does not modify** the data.
   - **Multiple readers** can access the data **simultaneously**, if **no writer** is writing.
2. **Writer:**
   - A process that **writes or modifies** the shared data.
   - **Exclusive access** is required. Only **one writer** can access the data at a time, and **no readers** should access during writing.

□ Problem:

- **Race conditions** occur if readers and writers access the data simultaneously.
- Need **synchronization mechanisms** to prevent conflicts.

□ Constraints to Ensure:

1. No reader should be allowed to read **while a writer is writing**.
2. No writer should write **while any reader is reading** or **another writer is writing**.

□ Types of Reader-Writer Problems:

1. First Readers-Writers Problem (Reader Priority)

- Multiple readers can read simultaneously.
- If a writer is waiting, new readers may still enter.
- May cause **starvation of writers**.

2. Second Readers-Writers Problem (Writer Priority)

- If a writer is waiting, **no new readers** are allowed to enter.
- Gives priority to writers.
- May cause **starvation of readers**.

□ **Variables Used for Synchronization:**

```
int readcount = 0   // Number of active readers
semaphore mutex     // To protect readcount variable
semaphore wrt       // To allow one writer or multiple readers
```

Solution to Reader-Writer Problem (Reader Priority Example):

□ ***Reader Process:***

```
wait(mutex);
readcount++;
if (readcount == 1)
   wait(wrt);     // First reader locks writer
signal(mutex);

// --- critical section (read the data) ---

wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);   // Last reader unlocks writer
signal(mutex);
```

□ ***Writer Process:***

```
wait(wrt);        // Wait until no readers or writers
// --- critical section (write the data) ---
signal(wrt);      // Release lock
```

□ **Flow Explanation:**

- **Multiple readers** can increase readcount, allowing concurrent access.
- **First reader** locks the writer to prevent modification.
- **Last reader** unlocks the writer.
- **Writer** acquires wrt lock to ensure **exclusive access**.

□ **Why Mutex and Semaphore?**

- mutex ensures **mutual exclusion** for modifying readcount.
- wrt ensures **writer exclusion** and **reader coordination**.

□ **Starvation Issues:**

- **Reader-priority solution** may starve writers if readers keep coming.
- **Writer-priority solution** uses an additional semaphore to block new readers when a writer is waiting.

□ **Applications in Real World:**

- Database systems: Reading and updating a table.
- File access: Multiple users reading a document vs. one editing it.
- Cache systems: Reading from memory while updating content.

Lec-20: The Dining Philosophers problem

Goal:

The **Dining Philosophers Problem** is a classic example of **process synchronization** used to illustrate the issues of **deadlock, starvation, and concurrency control** in a multi-process system.

□ **Problem Setup:**

1. There are **5 philosophers** sitting around a **circular table**.
2. In the center is a **bowl of noodles**, and on the table are **5 forks** — one **between each pair** of philosophers.
3. Each philosopher alternates between two activities:
    - **Thinking**

- **Eating**

 Philosopher's Life Cycle:

- **Thinking:**
  A philosopher thinks quietly and doesn't require any forks.
- **Hungry:**
  When hungry, the philosopher attempts to **pick up the two forks adjacent to them** — one on the **left** and one on the **right**.
- **Eating:**
  A philosopher can eat **only when they have both forks**.
  After eating, they **put down both forks** and go back to thinking.

 Fork Access:

- Each fork is a **shared resource** between two neighboring philosophers.
- A philosopher can **pick up only one fork at a time**.
- If a fork is already taken, the philosopher must **wait**.

Problem: Deadlock

- If all 5 philosophers become hungry **simultaneously** and each picks up **only their left fork**, no one can pick up the right fork (all are held).
- This leads to a **circular wait** condition → **Deadlock**.

 Using Semaphores for Synchronization:

Semaphores can help **control fork access**, but **on their own**, they **do not prevent deadlock**.

Semaphore Representation:

```
semaphore fork[5] = {1, 1, 1, 1, 1};  // Each fork is a binary semaphore
```

Fork Usage:

```
// To pick up a fork
wait(fork[i]);

// To put down a fork
```

```
signal(fork[i]);
```

Why Deadlock Happens:

If **all philosophers** pick up their **left fork at the same time**, then:

- Each has **one fork**, and
- Tries to pick up their **right fork**, which is already taken.

All philosophers **wait forever** — no one can proceed.
This is deadlock.

Solutions to Prevent Deadlock:

1. Allow Only 4 Philosophers to Sit at a Time

- At most **4** philosophers can attempt to eat at any one time.
- Ensures **at least one fork** is always free, avoiding circular wait.

2. Atomic Fork Picking (Critical Section)

- Philosopher **picks both forks together** in a critical section.
- If both forks are not available, **don't pick any**.
- Avoids holding one fork and waiting for another.

3. Odd-Even Rule

- **Odd-numbered** philosophers: Pick **left fork first**, then right.
- **Even-numbered** philosophers: Pick **right fork first**, then left.
- Breaks the **symmetry** of fork acquisition → avoids circular wait.

 Improved Pseudocode Example (Odd-Even Rule):

```
// Assume fork[i] is a binary semaphore

philosopher(i):
    while (true) {
        think();

        if (i % 2 == 0) {
```

```
        wait(fork[(i+1)%5]);  // right fork
        wait(fork[i]);        // left fork
    } else {
        wait(fork[i]);        // left fork
        wait(fork[(i+1)%5]);  // right fork
    }

    eat();

    signal(fork[i]);
    signal(fork[(i+1)%5]);
}
```

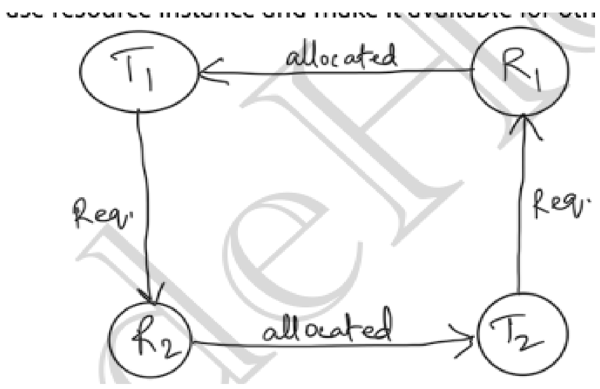
 Key Concepts Highlighted by the Problem:

Concept	Explanation
Mutual Exclusion	Only one philosopher can use a fork at a time.
Deadlock	Circular waiting for forks leads to a freeze in the system.
Starvation	A philosopher may never get a chance to eat if others dominate.
Concurrency	Multiple philosophers can eat if they don’t share forks.

LEC-21: Deadlock Part-1

1–5. Introduction to Deadlock

1. In a **multiprogramming environment**, multiple processes run simultaneously.
2. These processes compete for **limited system resources** like CPU, memory, files, etc.
3. A process may **request a resource** (e.g., a file or printer). If the resource is **already taken**, the process enters a **waiting state**.
4. In some cases, the **resource may never become available**, causing the process to **wait forever**. This situation is called a **Deadlock**.
5. **Deadlock** occurs when **two or more processes** are waiting **indefinitely** for resources **held by each other**. None can proceed, resulting in a complete **standstill**.



 Concept Summary

Deadlock is a situation where a group of processes are **blocked** because each process is holding a resource and **waiting for another resource** that is held by some other process in the group.

6–7. Examples and Nature of Resources

6. **Examples of resources**:
   - Memory (RAM)
   - CPU cycles
   - Files
   - Locks
   - Sockets
   - I/O Devices (printers, keyboards)
7. A resource may have **multiple instances**.
   - E.g., CPU is a resource; a system may have 2 CPUs (instances).

---

8. Lifecycle of Resource Utilization by Process

A process utilizes a resource in **three stages**:

a. **Request**:

- Process requests the resource.
- If available → allocate it.
- If not → process waits.

b. **Use**:

- Process uses the allocated resource.

c. **Release**:

- Process releases the resource after using it.
- Resource becomes available for others.

---

9. Necessary Conditions for Deadlock

Deadlock **can occur** only if **all four** of the following conditions are **true simultaneously**:

a. Mutual Exclusion

- Only **one process** can use a resource at a time.
- If another process requests it, it must wait.

b. Hold and Wait

- A process is **holding at least one resource** and waiting to acquire **more resources**.

c. No Preemption

- A resource **cannot be forcibly taken** from a process.
- It must be **voluntarily released**.

d. Circular Wait

- A set of processes {P0, P1, ..., Pn} exist such that:
    - P0 waits for a resource held by P1,
    - P1 waits for a resource held by P2,
    - ...
    - Pn waits for a resource held by P0.
- Forms a **cycle**, leading to **circular waiting**.

---

10. Methods for Handling Deadlocks

There are **3 common strategies** to deal with Deadlocks:

a. Prevention or Avoidance

- Use specific protocols to **prevent deadlock from occurring** at all.

b. Detection and Recovery

- Allow the system to **enter deadlock**, then **detect** it and **recover** by terminating processes or preempting resources.

c. Deadlock Ignorance (Ostrich Algorithm)

- **Ignore** the problem, assuming deadlocks **rarely occur**.
- Many OSes like UNIX use this due to the **cost of prevention**.

---

11. Prevention vs Avoidance

To **ensure deadlocks never occur**, two approaches:

- **Prevention**: Design the system so that **one of the four necessary conditions cannot occur**.
- **Avoidance**: Dynamically analyze resource-allocation requests to **avoid unsafe states**.

---

12. Deadlock Prevention in Detail

a. Prevent Mutual Exclusion

- Not always possible: Some resources (e.g., printer) are **inherently non-shareable**.
- But for sharable resources (e.g., **read-only files**), allow **multiple accesses**.

b. Prevent Hold and Wait

- Protocol A:
    - A process must **request all required resources** at once before execution.
- Protocol B:
    - A process can request a resource **only if it is holding no other resource**.

Downside: This may cause **low resource utilization** and **starvation**.

c. Prevent No Preemption

- If a process requests a resource not available, **preempt all its resources**.
- Restart the process once **all resources are available**.
- Another option: Preempt resource from another **waiting process** and give it to the **requesting process**.

 This may lead to **Live Lock** (continuous resource switching, no progress).

d. Prevent Circular Wait

- Impose a **total ordering** on resources (e.g., R1 < R2 < R3).
- A process must request resources in **increasing order only**.
- This **breaks the cycle** of dependency.

LEC-22: Deadlock Part-2

1. Deadlock Avoidance

Definition:
Deadlock avoidance is a strategy where the system is **aware of all future resource requests** a process may make and carefully schedules resource allocation **to avoid entering an unsafe state** (which may lead to deadlock).

Key Concepts:

- The **kernel/system must know** in advance:
    - Maximum resource demand of each process
    - Current resource allocation
    - Remaining available resources
- When a process requests a resource, the system checks whether **granting the request keeps the system in a safe state**.

---

a. Scheduling with Deadlock Avoidance

- The OS **schedules resource allocation** in such a way that **deadlocks never occur**.
- If a request causes an unsafe state, it is **delayed**.

---

b. Safe State

- A **safe state** means that there is a way (a **safe sequence**) in which **all processes can complete**.
- Safe sequence: A sequence of all processes such that each process can get the resources it needs to complete, even if all of them request their maximum.

---

c. Unsafe State

- An **unsafe state** is **not necessarily a deadlock**, but it may lead to a deadlock.
- If a resource is allocated without checking, and a deadlock eventually occurs, it means the system entered an unsafe state earlier.

---

d. Key Principle

- A resource request is **approved only** if the resulting state is **safe**.
- Otherwise, the request is **delayed** until the system returns to a safe state.

---

e. When System Cannot Fulfill All Requests

- If the system **cannot satisfy** the maximum possible resource requests of all processes, the system is in an **unsafe state**.

---

f. Banker's Algorithm

- An algorithm used to **check whether granting a request leads to a safe state**.
- Named because it simulates a bank ensuring that it **never runs out of resources** (money) even after giving loans (resource allocations).

---

2. Banker's Algorithm — How It Works:

- Each process declares its **maximum resource requirement** at the start.
- When a process requests a resource:
  - The system checks whether it has **enough resources** to fulfill it.
  - Then it simulates allocating the resources and checks if the system would still be in a **safe state**.
  - If safe → Allocate
    If unsafe → Process waits

---

3. Deadlock Detection

If deadlock prevention/avoidance is **not used**, the system may enter a deadlock. In that case, **deadlock detection** is used.

a. Single Instance per Resource (Wait-For Graph)

- **Wait-for graph (WFG)**:
  - Nodes → Processes
  - Edge from P1 → P2 if P1 is waiting for a resource held by P2
- **Deadlock exists** ⇨ if and only if there is a **cycle** in the WFG
- The system periodically runs **cycle detection algorithms** on the WFG to find deadlocks.

---

b. Multiple Instances per Resource

- Banker’s algorithm is **used again** to check if a deadlock is present with multiple instances.
- System simulates allocations to check for possibility of all processes completing.

---

4. Recovery from Deadlock

If a deadlock is **detected**, the system must recover from it using one of these methods:

a. Process Termination

1. **Abort All Deadlocked Processes**
   - Simple but can lead to **loss of work/data**.
2. **Abort One Process at a Time**
   - Abort one process, release its resources, recheck for deadlock
   - Repeat until cycle is broken

---

b. Resource Preemption

- **Temporarily take resources** from some processes and give them to others
- Continue doing this until deadlock is resolved

□ Challenge: Difficult to decide:

- Which process to preempt?
- What resources?
- Rollback state of the process?

LEC-24: Memory Management Techniques | Contiguous Memory Allocation

1. Introduction to Memory Management in Multiprogramming Environment

- In a **multiprogramming** system, multiple processes reside in **main memory** to keep the CPU busy and improve responsiveness.
- To achieve high performance:
  - Multiple processes must be kept in memory.
  - Memory must be **shared efficiently** among all processes.
  - The **Operating System (OS)** must manage this sharing securely and fairly.

---

2. Logical vs Physical Address Space

a. Logical Address (aka Virtual Address)

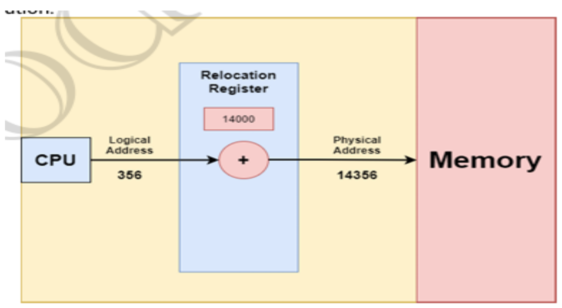
- **Generated by the CPU** during process execution.
- Used by the **process** to reference memory locations.
- **Does not physically exist**; it's an abstract address.
- **User interacts** with logical addresses, not physical ones.
- Logical address space: **range from 0 to max** (as defined by the program).
- **Example**: Think of it like a seat number on a ticket — it's valid only after it's mapped to an actual physical seat.

b. Physical Address

- Actual **location in main memory (RAM)**.
- Generated by **Memory Management Unit (MMU)**.
- Range: **(R + 0) to (R + max)**, where **R = base (relocation) address**.
- Not directly accessible by user programs.

c. Role of MMU (Memory Management Unit)

- MMU dynamically translates **logical → physical address** at runtime.
- It adds the **relocation register value** to the logical address.
- Protects OS and other user processes by enforcing bounds.



3. Memory Mapping & Protection

a. Why Protection?

- Prevents a process from accessing memory that doesn't belong to it.
- Ensures **isolation** between processes.

b. Mechanism

- OS uses:
  - **Relocation Register (Base address)**: smallest physical address a process can access.
  - **Limit Register**: max range for the process's memory access.

c. Enforcement

- Every time a memory address is accessed:
  - Logical address is **checked** against the limit register.
  - If valid → passed to MMU for conversion.
  - If invalid → **trap** is raised → OS handles it as a **fatal error**.

d. During Context Switch

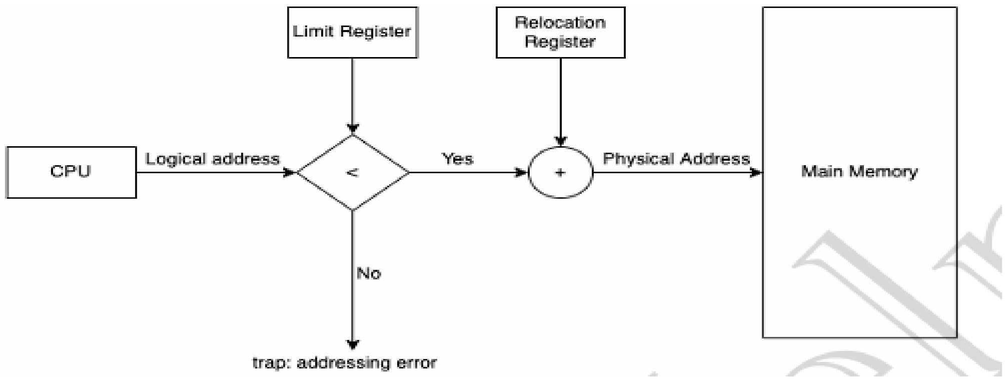
- Dispatcher loads the correct relocation and limit registers for the selected process.

---

4. Address Translation

- **Logical Address + Base (Relocation Register) = Physical Address**
- Happens **dynamically** during program execution.
- Ensures **process isolation and memory protection**.

h. **Address Translation**



---

5. Allocation Methods in Physical Memory

- Two primary memory allocation methods:
    1. **Contiguous Allocation**
    2. **Non-Contiguous Allocation** (like paging/segmentation – covered later)

---

6. Contiguous Memory Allocation

a. Definition

- Each process occupies a **single, continuous block** of memory.
- Simple to implement but has **fragmentation issues**.

---

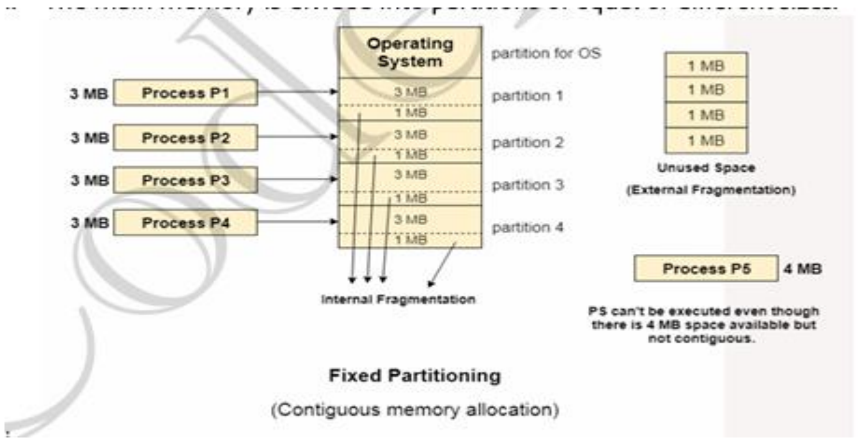
b. Fixed Partitioning

i. What is it?

- Divide memory into fixed-size blocks (partitions) **before execution begins**.
- Each block holds one process.

ii. Limitations

1. **Internal Fragmentation**
    - Occurs when a process is smaller than the allocated partition.
    - Unused space within a partition is wasted.
2. **External Fragmentation**
    - Enough total free memory exists, but **not in one contiguous block**.
    - Cannot load new processes even though space is available.
3. **Limitation on Process Size**
    - If a process is **larger than any partition**, it can’t be loaded.
    - Must be smaller than or equal to the **largest partition**.
4. **Low Multiprogramming Degree**
    - Number of concurrent processes = number of partitions.
    - Limited flexibility → poor memory utilization.

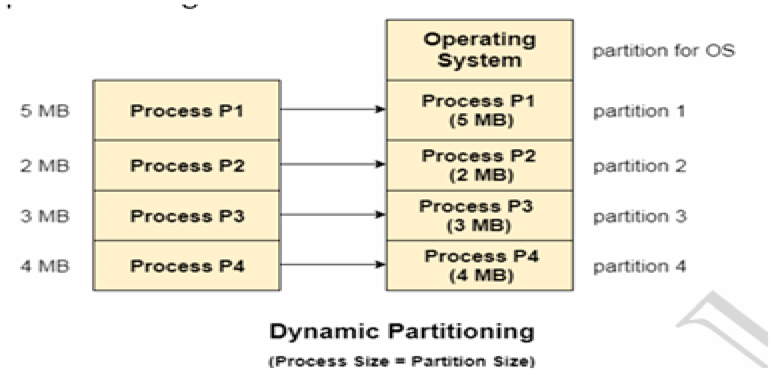


---

c. Dynamic Partitioning

i. What is it?

- Partitions are **created dynamically** at runtime to fit the process size.
- More flexible than fixed partitioning.



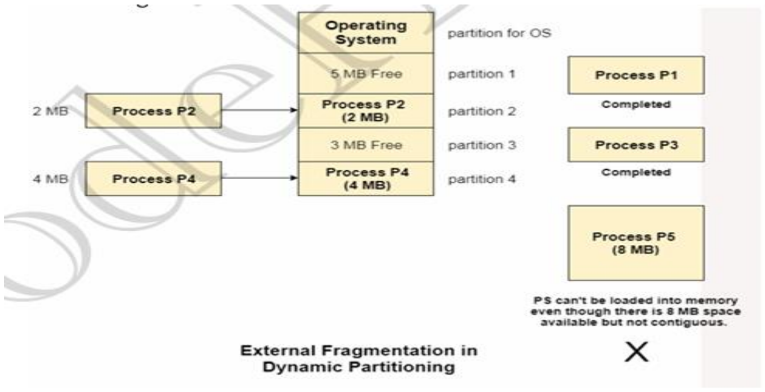
ii. Advantages

1. **No Internal Fragmentation**
   - Since partition size = process size.
2. **No Limit on Process Size**
   - As long as there's enough total free memory.
3. **Better Multiprogramming**
   - Utilizes memory more efficiently.

iii. Limitations

1. **External Fragmentation**
   - As processes are loaded and removed, free memory gets scattered.

- Still a problem because memory is not compacted.



LEC-25: Free Space Management

1. Defragmentation / Compaction

Dynamic Partitioning is flexible in terms of process sizes, but it **suffers from external fragmentation**—free memory is scattered in small chunks across the memory. To handle this, **compaction** (also known as **defragmentation**) is used.

a. Why Compaction is Needed

- In **dynamic partitioning**, memory gets fragmented externally as processes load and unload.
- Even if total free memory is enough for a new process, it may not be **contiguous**, so allocation fails.

b. What is Compaction

- **Compaction merges all scattered free holes** into a **single large contiguous block** of memory.
- This is done by **shifting the occupied blocks together** toward one side of the memory and **merging** the empty holes.

c. Example Scenario

Imagine memory like this:

[Process1][Free][Process2][Free][Process3][Free]

After compaction, it becomes:

[Process1][Process2][Process3][FreeFreeFree]

Now, the free space is large and contiguous, allowing new large processes to be loaded.

d. Advantages

- Reduces **external fragmentation**.
- Allows **larger processes** to be loaded in memory.
- Improves **memory utilization**.

e. Disadvantages

- **Consumes CPU time**.
- Temporarily **degrades system performance** during compaction.
- Processes need to be moved, which may be **slow**.

---

2. How Free Space is Represented in OS

The **Operating System** maintains a **Free List** to keep track of all **unallocated memory blocks (holes)**.

a. Free List

- Implemented using a **Linked List** data structure.
- Each node stores the:
  - Starting address of the hole
  - Size of the hole
  - Pointer to the next hole

This allows the OS to **dynamically manage** memory and find free spaces for new processes.

---

3. How to Satisfy a Request for n Size from Free Holes

To allocate memory efficiently, the OS uses **memory allocation algorithms** to select a suitable hole from the free list based on the requested size.

---

b. First Fit

- **Strategy:** Scan the list from beginning and **allocate the first hole** that is big enough.
- **Advantages:**
  - Simple and **fast**
  - **Low overhead** in searching
- **Disadvantages:**
  - May cause **external fragmentation**

 - Tends to leave **small unusable holes** at the beginning

---

c. Next Fit

- **Strategy:** Similar to First Fit, but instead of starting from the beginning each time, it starts from the **last allocated hole**.
- **Advantages:**
  - Avoids clustering of small holes at the start
  - Performance can be better in some workloads
- **Disadvantages:**
  - May still cause fragmentation
  - Slightly **more complex** to implement than First Fit

---

d. Best Fit

- **Strategy:** Allocate the **smallest hole** that is **just big enough** for the request.
- **Advantages:**
  - Reduces **internal fragmentation**
- **Disadvantages:**
  - **Slow**—must **search the entire list**
  - Leaves many small holes → leads to **external fragmentation**

---

e. Worst Fit

- **Strategy:** Allocate the **largest available hole**.
- **Advantages:**
  - Leaves behind relatively **larger holes** that may be useful for other processes
- **Disadvantages:**
  - **Slow**—requires **scanning entire list**
  - Might not always lead to efficient space utilization

LEC-26: Paging | Non-Contiguous Memory Allocation

1. Problem with Dynamic Partitioning

a. Main Disadvantage: External Fragmentation

- In **dynamic partitioning**, memory is divided based on process requirements.
- As processes arrive and leave, **free spaces are scattered** (external fragmentation).

- Even if total free memory is enough, **a large process might not fit** if memory is not **contiguous**.

b. Compaction: A Partial Solution

- **Compaction** rearranges memory by bringing all **used blocks together** and **merging free spaces**.
- Though it helps eliminate fragmentation, it adds **overhead** due to data movement.
- Hence, we need a **more flexible and efficient mechanism** to load processes: ➡ **Paging**.

---

2. Idea Behind Paging

a. Problem Example

- Suppose two non-contiguous **free holes of 1KB each** exist.
- A process of **2KB cannot be allocated** in contiguous memory → fails in dynamic partitioning.

b. Solution: Break the Process

- **Split the process into smaller blocks**, e.g., 1KB + 1KB.
- Allocate them **non-contiguously** across memory.
- This is the **core idea of Paging**.

c. Paging Concept

Paging allows **non-contiguous memory allocation** by dividing:

- **Physical Memory → Fixed-size blocks = Frames**
- **Logical (process) Memory → Same-size blocks = Pages**
- **Page size = Frame size**

---

3. Paging in Detail

a. Avoids External Fragmentation

- Pages and frames are of **equal, fixed size**.
- Any free frame can hold **any page**, no need for continuous memory.
- No need for **compaction**.

b. Common Page Sizes

- Traditional size: **4KB**
- Modern processors support **multiple page sizes** (e.g., 4KB, 2MB, 1GB) for performance.

c. Page Table

A **data structure** maintained by the OS.

i. Purpose:

- Keeps track of which **logical page** maps to which **physical frame**.

ii. Working:

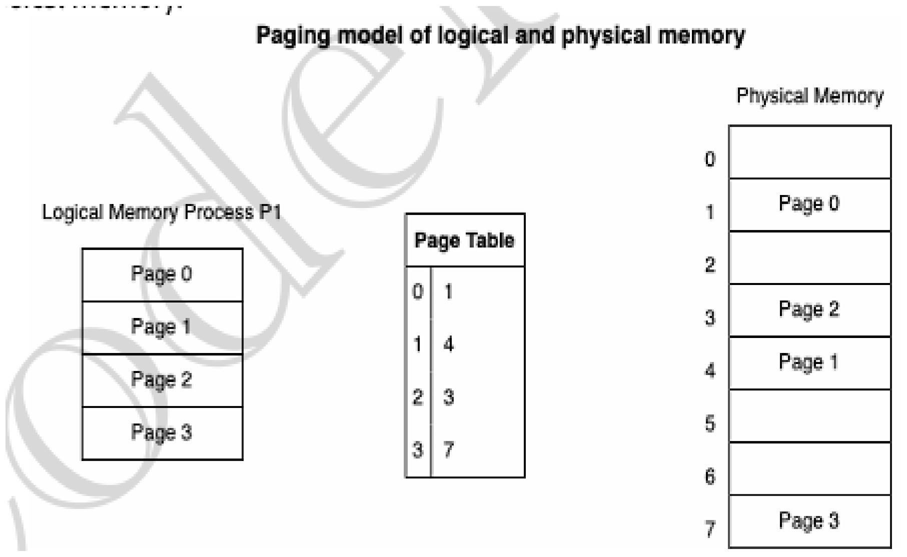
- Each **logical address** generated by the CPU is split into:
  - **Page number (p)**
  - **Page offset (d)**

iii. Usage:

- **Page number (p)** → Index in **Page Table** → Get **Frame Number**
- Combine **Frame Base Address + Offset (d)** → Get **Physical Address**

iv. Storage:

- Page table is kept in **main memory**.
- A register called **Page Table Base Register (PTBR)** points to the page table.
- During **context switching**, only PTBR needs to be updated.



4. How Paging Avoids External Fragmentation

- Pages can be placed **anywhere in physical memory**.
- Memory can be filled with **non-contiguous frames**.
- Because there's **no requirement for contiguous allocation**, **external fragmentation is eliminated**.

---

5. Why Paging is Slow & How to Speed It Up

a. Problem: Too Many Memory Accesses

- To access data:
    1. Use logical address to **look up page table** (1 memory access)
    2. Use result to **access actual data** (2nd memory access)
- Thus, **2 memory accesses** per instruction → **slow performance**.

---

6. Translation Lookaside Buffer (TLB)

a. What is TLB?

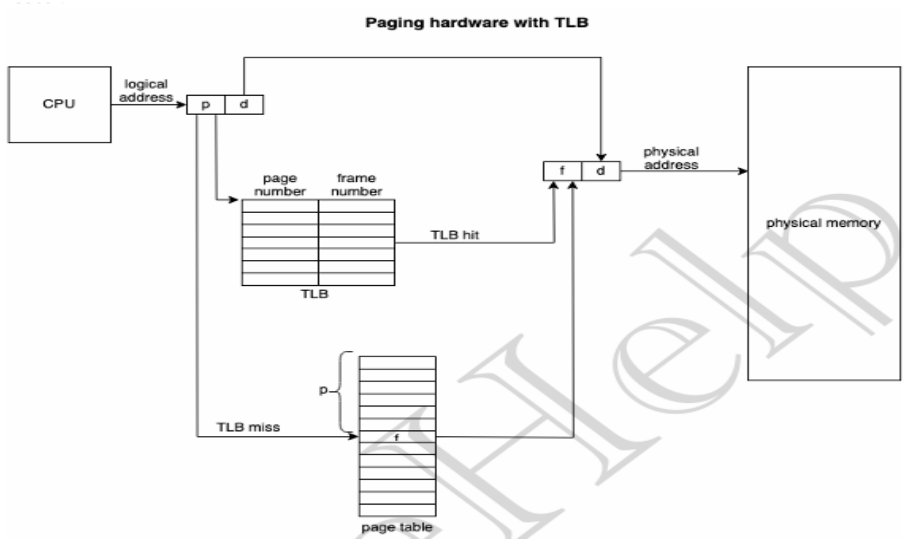
- A **hardware cache** that stores recent **page table entries**.
- Works like a **key-value store**:
    - **Key** = Page number
    - **Value** = Frame number

b. How TLB Works

- When CPU generates a logical address:
    - **Check TLB first** for page number.
    - If found (□ **TLB Hit**) → Use cached frame number → **Fast translation**
    - If not found (□ **TLB Miss**) → Access page table → Load into TLB for future

c. Role of ASID (Address Space Identifier)

- Each TLB entry stores an **ASID**.
- ASID uniquely identifies the **process** that owns the page.
- This allows:
    - **Multiple processes** to share the TLB
    - Prevents wrong mappings (i.e., page table of process A used by process B)



LEC-27: Segmentation | Non-Contiguous Memory Allocation

Introduction

 Problem with Paging

- **Paging** divides memory into **fixed-size blocks (pages)**.
- It **ignores the logical structure** of a program.
- OS divides process memory arbitrarily into pages without considering **how users or compilers group code** (e.g., functions, variables).
- This causes **inefficiencies**, such as:
  - Related functions/data may get split into different pages.
  - Pages may load independently, affecting performance and consistency.

---

1. What is Segmentation?

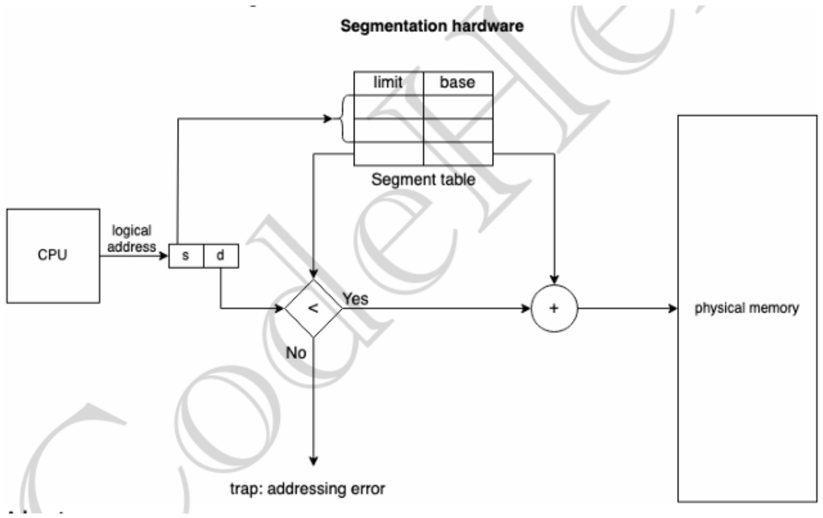
 Definition:

- **Segmentation** is a **memory management technique** that supports the **user's view of memory**.
- It **divides the logical address space into variable-sized segments** based on the **logical divisions** of a program.

 Each Segment Contains:

- Related components like:
  - **Main function**

  - **Stack**
  - **Library code**
  - **Global variables**
- These are grouped meaningfully rather than uniformly (as in paging).



---

2. Logical Address in Segmentation

Each logical address is represented as:

<segment number, offset> → {s, d}

- **Segment number (s)**: Identifies which segment we're accessing.
- **Offset (d)**: Specifies how far into that segment we want to go.

---

3. Key Differences: Paging vs Segmentation

Feature	Paging	Segmentation
Division type	Fixed-size pages	Variable-size segments
Based on	OS's view (physical memory management)	User's view (logical memory organization)
Logical relation	Not preserved (function/data may split)	Preserved (related data grouped in segments)
Fragmentation type	**Internal fragmentation** possible	**External fragmentation** possible
Address format	<Page number, offset>	<Segment number, offset>

Feature	Paging	Segmentation
Efficiency	Less efficient in maintaining logical groups	More efficient for user/program structure

---

4. Why Segmentation?

- To preserve the **logical structure** of the program in memory.
- Ensures that:
  - **Functions and related data are kept together**.
  - The OS can load segments in ways that preserve **logical dependencies**.

---

5. Advantages of Segmentation

a. No Internal Fragmentation

- Since segments are **variable-sized**, only exact space is allocated.
- No unused space within a segment like in paging.

b. Contiguous Allocation Within Segment

- Each segment is stored **contiguously**, making access within segment efficient.

c. Smaller Segment Table

- Typically smaller than the page table since fewer segments are needed than pages.

d. Logical Grouping Helps Efficiency

- The **compiler** can keep similar code/data in the same segment.
- Results in **better locality of reference** and **efficient execution**.

---

6. Disadvantages of Segmentation

a. External Fragmentation

- Since segments are variable-sized and stored contiguously, **external fragmentation** occurs (like dynamic partitioning).

b. Swapping is Complex

- Different segment sizes make **swapping segments** in and out of memory more difficult.
- It's harder to find the right spot to place a segment.

---

7. Modern Systems Use Hybrid: Segmentation + Paging

 Why Combine Both?

- **Segmentation** preserves user's view.
- **Paging** simplifies memory allocation and avoids external fragmentation.

 Hybrid Approach:

- **Each segment** is divided into **pages**.
- Virtual address = **<Segment number, Page number, Offset>**
- Helps:
  - Maintain logical structure (segmentation)
  - Efficient memory use (paging)
  - Avoid external fragmentation while preserving logical grouping

LEC-28: What is Virtual Memory? || Demand Paging || Page Faults

1. Virtual Memory – Definition & Purpose

- **Virtual memory** is a memory management technique that allows the **execution of processes not completely in main memory**.
- It gives the **illusion** of a very large main memory to the user/programmer, even though **physical memory is limited**.
- It does so by **using secondary memory (e.g., hard disk/SSD)** as an extension of RAM (this portion is called **swap space**).

---

2. Why is Virtual Memory Useful?

- Instructions must reside in **main memory to execute**, which **limits the size of programs**.
- In reality, **not all parts** of a program are needed at the same time (e.g., error-handling routines).
- **Advantages**:
  - a. A program is no longer constrained by the **size of physical memory**.
  - b. Since each program occupies **less RAM**, the **number of concurrent processes increases**, improving **CPU utilization and throughput**.
  - c. It improves **system efficiency** and **user experience** by allowing large applications to run seamlessly.

---

3. Demand Paging – A Key Virtual Memory Technique

- **Demand paging** is a lazy loading strategy where **only required pages** of a process are loaded into RAM.
- Pages are brought from **secondary memory** (disk) **only when accessed** → helps reduce memory usage.
- **Page fault** occurs if the process tries to access a page not in memory.

---

4. Lazy Swapper vs Pager

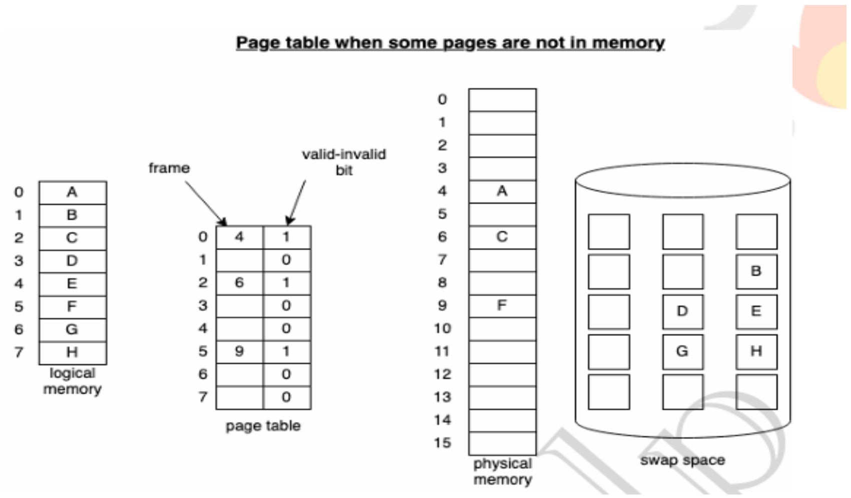
- **Swapper** traditionally deals with **entire processes** (used in older systems).
- In virtual memory, we use a **Pager** instead:
    - Handles **individual pages**.
    - **Lazy swapper** delays loading a page until it's absolutely necessary.

---

5. Working of Demand Paging

- a. When a process is scheduled to run, the **Pager predicts required pages**.
- b. **Only those pages are loaded** into memory → avoids unnecessary memory consumption.
- c. Helps reduce:
    - Swap time.
    - Physical memory usage.
- d. Uses a **valid-invalid bit** in the page table:
    - 1: Page is **valid and in memory**.
    - 0: Page is either **invalid** (illegal) or **on disk**.

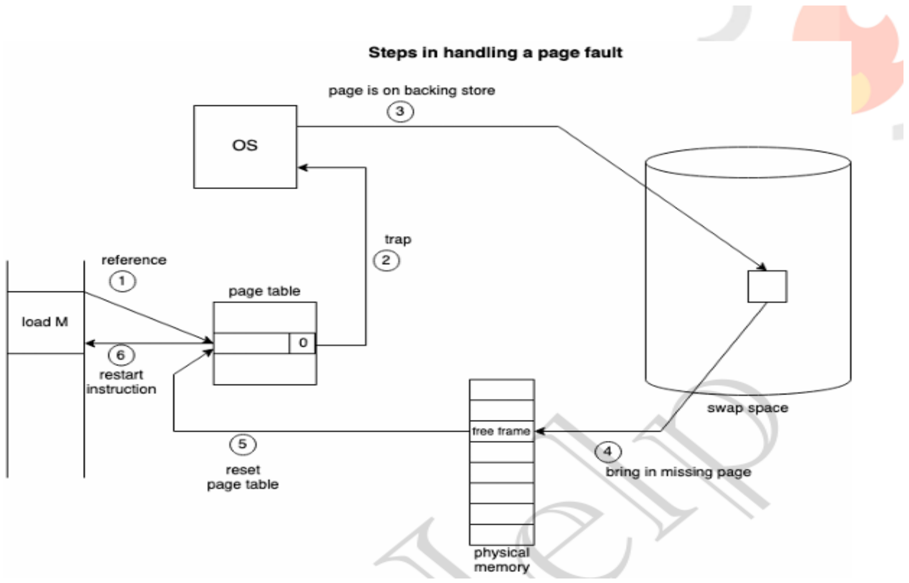
e.



6. Handling a Page Fault

When the process tries to access a page that's not in memory:

- a. **Trap to OS** due to invalid bit.
- b. OS checks if the memory reference is:
  - Valid → continue.
  - Invalid → throw **segmentation fault** or similar error.
- c. If valid:
  - i. Locate a **free frame** in memory.
  - ii. Schedule **disk read** to bring the page into that frame.
  - iii. Update **page table** to mark page as valid.
  - iv. **Restart the instruction** that caused the page fault.



i.

7. Pure Demand Paging

- In extreme implementations:
  - Start with **zero pages in memory**.
  - First instruction causes a **page fault**, and the required page is loaded.
- Never load a page until it is **actually needed**.
- Relies on the principle of **locality of reference**:
  - Temporal locality → recently used pages will be used again.
  - Spatial locality → nearby pages will likely be used soon.

8. Advantages of Virtual Memory

- a. Increases **degree of multiprogramming**.
- b. Users can run **large programs** on systems with **limited RAM**.
- c. Memory is used **more efficiently**.
- d. Reduces load time (load only parts of programs).

---

9. Disadvantages of Virtual Memory

- a. **Slower performance** compared to actual RAM (disk access is slow).
- b. **Thrashing** may occur:
    - If too many page faults happen, system spends more time swapping than executing.
    - Occurs due to **poor locality of reference** or **overloaded RAM**.

LEC-29: Page Replacement Algorithms

Page Fault & Page Replacement

- **Page Fault occurs** when a process tries to access a page that is **not currently loaded in a physical frame**.
- The Operating System (OS) must **bring the needed page from the swap-space (secondary storage)** into a free frame in physical memory.
- If **all frames are busy (no free frames available)**, the OS needs to perform **page replacement** — it selects one of the currently loaded pages to be swapped out (removed) to make space.
- The **page replacement algorithm** is responsible for deciding **which page to remove** and **which new page to load**.

Page Replacement Algorithms (Goal: Minimize Page Faults)

a. FIFO (First In First Out)

- **How it works:** Replace the oldest page in memory — the one that came in first.
- **Advantages:**
    - Simple and easy to implement.
- **Disadvantages:**
    - Sometimes replaces pages still heavily used, causing unnecessary page faults.
    - The page replaced might be an old initialization module (good candidate) or a heavily used variable (bad candidate).
    - Subject to **Belady's Anomaly** — increasing the number of frames can sometimes increase page faults, which is counterintuitive.

b. Belady's Anomaly

- Normally, increasing the number of frames reduces page faults.
- However, in FIFO, **adding more frames can increase the number of page faults** in some cases — this is called **Belady's Anomaly**.
- This anomaly **does not occur in LRU or Optimal algorithms**.

c. Optimal Page Replacement

- **How it works:** Replace the page that will **not be used for the longest time in the future**.
- If a page will **never be referenced again**, replace that one.
- Has the **lowest possible page fault rate** (optimal).
- **Disadvantage:** Requires **future knowledge of reference strings**, which is impossible in practice.
- Useful as a benchmark to compare other algorithms.

d. Least Recently Used (LRU)

- **How it works:** Replace the page that **has not been used for the longest time in the past**, assuming the recent past predicts the near future.
- **Implementations:**
    1. **Counters:**
        - Each page table entry has a time stamp updated on every access.
        - Replace the page with the smallest (oldest) time value.
    2. **Stack:**
        - Keep pages in a stack.
        - When a page is accessed, move it to the top.
        - The bottom of the stack is the least recently used page.
        - Since pages can be removed from the middle, a **doubly linked list** is often used for efficient updates.

e. Counting-Based Algorithms

- **Reference Counting:** Keep track of how many times each page is referenced.

1. **Least Frequently Used (LFU):**
    - Pages with **lowest reference counts** are replaced.
    - Pages actively used have high counts, so they stay in memory longer.
2. **Most Frequently Used (MFU):**
    - Replace pages with the **highest reference counts**, assuming that a page with low usage was just brought in and not yet used.
    - Neither MFU nor LFU are very common due to implementation complexity and poor performance in some scenarios.

Summary Table

Algorithm	Principle	Advantages	Disadvantages
FIFO	Replace oldest page	Simple	Belady's Anomaly, poor in some cases
Optimal	Replace page not used for longest future time	Lowest page faults (optimal)	Impossible to implement fully (needs future knowledge)
LRU	Replace least recently used page	Good approximation of optimal	Overhead in tracking usage
LFU	Replace least frequently used	Keeps actively used pages	Complexity, poor for some patterns
MFU	Replace most frequently used	Based on assumption of new pages	Rarely used, poor practical performance

LEC-30: Thrashing

1. Thrashing

a. What is Thrashing?

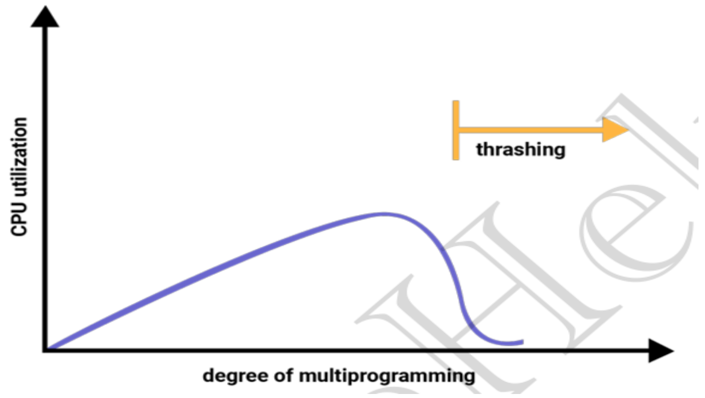
- Thrashing happens when a process **does not have enough frames** to hold all the pages it actively needs.
- Because of this shortage, it keeps causing **page faults very frequently**.
- Each page fault causes the OS to replace some page in memory with a new one from disk.
- However, since **all pages are actively used**, the replaced page is needed again immediately.
- This leads to a **loop of continuous page faults**, with pages being swapped out and brought back in repeatedly.

b. Why does Thrashing happen?

- When the working set (the set of pages actively used by the process) **doesn't fit into the allocated frames**, the process keeps faulting.
- This results in excessive paging activity.

c. Effects of Thrashing

- The system spends **more time handling page faults** (swapping pages in and out) than actually executing process instructions.
- Overall system performance **drops drastically**.
- The CPU utilization decreases as the system is busy in paging rather than useful work.



---

2. Techniques to Handle Thrashing

a. Working Set Model

- Based on the **Locality Model** of program behavior.
- **Locality** means a process tends to access a set of pages repeatedly for some time before moving on to another set.
- The **Working Set** of a process is the set of pages that the process is currently using or will likely use shortly.
- The idea: **Allocate enough frames to hold the current working set**.
  - If a process gets enough frames to hold its working set, page faults reduce drastically.
  - If allocated frames are less than the working set size, thrashing is likely.

b. Page Fault Frequency (PFF) Scheme

- **Monitor the page fault rate** of a process.
- Thrashing leads to a **high page fault rate**.
- We want to **control this page fault rate** within upper and lower bounds.
- **If page fault rate is too high:**
  - The process is given more frames to reduce faults.
- **If page fault rate is too low:**
  - The process might have more frames than necessary, so remove some frames.
- By **dynamically adjusting the number of frames** based on the page fault rate, thrashing can be minimized or avoided.

Cache Memory Organization

What is Cache Memory?

- Cache memory is a small, very fast memory located close to the CPU.
- It stores copies of data from frequently accessed main memory locations.
- The main purpose is to **reduce the average time to access data from the main memory**.

Why Cache is Needed?

- Main memory (RAM) is slower compared to the CPU.
- To avoid CPU waiting for data, cache stores frequently used data and instructions.
- If data is found in cache (**cache hit**), CPU accesses it quickly.
- If data is not found (**cache miss**), it must be fetched from slower main memory.

Cache Memory Organization Components:

1. **Cache Lines or Blocks:**
    - Cache is divided into small blocks called lines.
    - Each cache line stores a block of memory from main memory.
2. **Mapping Techniques:**
   How the CPU finds which cache line corresponds to which main memory block:
    - **Direct Mapped Cache:**
        - Each memory block maps to exactly one cache line.
        - Simple and fast but can cause more cache misses if multiple blocks map to the same line.
    - **Fully Associative Cache:**
        - Any memory block can be stored in any cache line.
        - More flexible but expensive and slower to search.
    - **Set Associative Cache:**
        - Compromise between the two.
        - Cache is divided into sets; each set contains several lines.
        - A memory block maps to exactly one set, but can go into any line within that set.
3. **Cache Controller:**
    - Manages reading and writing between CPU, cache, and main memory.
    - Decides cache hits/misses and replacement policy for evicting cache lines.
4. **Write Policies:**
    - **Write-through:** Data is written to both cache and main memory simultaneously.
    - **Write-back:** Data is written only to cache, and main memory is updated later when cache line is replaced.

Locality of Reference

Locality of Reference is the principle that programs tend to use data and instructions **close to those recently accessed**. It helps caches work efficiently.

Types of Locality:

1. **Temporal Locality:**
    - If a memory location is accessed, it is likely to be accessed again soon.
    - Example: Loop variables or repeatedly used instructions.
    - Cache benefits by keeping recently accessed data.
2. **Spatial Locality:**
    - If a memory location is accessed, locations nearby it are likely to be accessed soon.
    - Example: Accessing elements of an array sequentially.
    - Cache benefits by loading blocks of contiguous memory (cache lines).

How Locality Helps Cache:

- Because of locality, cache can **predict what data will be used soon** and keep it ready.
- Programs that exhibit strong locality perform better with caching.
- Cache size and block size are optimized based on locality patterns.

LEC-31: Disk Management

Disk Management

Disk management refers to the way an operating system controls and uses storage disks to store and retrieve data efficiently.

---

1. Disk Basics

- **Disk:** A storage device that stores data magnetically on rotating platters.
- **Components:**
    - **Platters:** Circular disks coated with magnetic material.
    - **Tracks:** Concentric circles on each platter.
    - **Sectors:** Each track is divided into sectors (smallest unit of storage).
    - **Cylinders:** A set of tracks vertically aligned across platters.
    - **Read/Write Head:** Mechanism to read or write data on a platter.
    - **Arm:** Moves heads over the platters.
- **How Data is Accessed:**
    - The read/write head moves to the right track (seek).
    - Waits for the sector to rotate under the head (rotational latency).
    - Reads or writes data.

---

2. Disk Storage and Disk Scheduling

Disk Storage

- Disks store data in blocks called sectors.
- OS manages files by allocating these sectors.
- Data on disks is accessed by specifying cylinder, track, and sector.

Disk Scheduling

- When multiple I/O requests come, the OS must decide in which order to service them.
- Efficient disk scheduling minimizes seek time and latency, improving performance.

Common Disk Scheduling Algorithms:

- **FCFS (First Come First Serve):**
    - Serve requests in order they arrive.
    - Simple but can cause long delays.
- **SSTF (Shortest Seek Time First):**
    - Serve the request closest to current head position.
    - Reduces seek time but can cause starvation.
- **SCAN (Elevator Algorithm):**
    - Head moves in one direction servicing requests until it reaches the end, then reverses.
    - Provides more uniform wait times.
- **C-SCAN (Circular SCAN):**
    - Head moves in one direction servicing requests; after reaching the end, jumps to start.
    - Provides more uniform wait times than SCAN.
- **LOOK and C-LOOK:**
    - Variants of SCAN and C-SCAN where the head only goes as far as the last request in that direction before reversing or jumping.

---

3. Total Transfer Time

The total time to transfer data from disk includes:

- **Seek Time:** Time taken for the read/write head to move to the correct track.
- **Rotational Latency:** Time waiting for the desired sector to rotate under the head.
- **Transfer Time:** Time to actually read/write the data once the head is in place.

Formula for Total Transfer Time:

$$\text{Total Transfer Time} = \text{Seek Time} + \text{Rotational Latency} + \text{Data Transfer Time}$$

-

- **Seek Time:** Usually a few milliseconds.
- **Rotational Latency:** Average is half the time for one full rotation of the disk.
- **Data Transfer Time:** Depends on the amount of data and disk transfer rate.

LEC-32: File System

File System

A **File System** is a method and data structure that an operating system uses to control how data is stored and retrieved on a storage device like a disk. It organizes files and directories to allow efficient access, management, and security.

---

1. File Allocation Methods

How files are stored on disk blocks/sectors:

- **Contiguous Allocation:**
  - Files are stored in consecutive blocks.
  - Easy and fast access (simple arithmetic to find data).
  - Problem: External fragmentation and difficulty in file size growth.
- **Linked Allocation:**
  - Each file is a linked list of disk blocks; blocks can be scattered.
  - No external fragmentation.
  - Sequential access only; random access is difficult.
  - Overhead of storing pointers.
- **Indexed Allocation:**
  - Uses an index block containing pointers to all file blocks.
  - Supports direct access (random access).
  - More flexible, but index block overhead.

---

2. Free-space Management

How the OS keeps track of free (unused) disk blocks:

- **Bit Vector (Bitmap):**
  - A bit array representing disk blocks.

  - 0 = free, 1 = allocated.
  - Efficient for large disks.
- **Linked List:**
  - Free blocks are linked together.
  - Simple but slower to allocate blocks.
- **Grouping:**
  - Groups of free blocks are linked together instead of individual blocks.
  - Reduces overhead.
- **Counting:**
  - Keep count of contiguous free blocks.
  - Faster allocation for large files.

---

3. File Organization and Access Mechanism

How data within files is structured and accessed:

- **File Organization:**
  - **Sequential:** Data accessed in order.
  - **Direct/Random:** Data accessed directly via an address or offset.
  - **Indexed:** Access through index tables.
- **Access Mechanisms:**
  - **Sequential Access:** Read/write data in sequence (like tape).
  - **Direct Access:** Jump directly to a specific point in the file (like disk).
  - **Indexed Access:** Use an index to find blocks quickly.

---

4. File Directories

- Directory is a structure that stores file metadata and organizes files.
- Types:
  - **Single-level Directory:** All files in one directory.
  - **Two-level Directory:** Separate directory for each user.
  - **Tree-structured Directory:** Hierarchical, allows subdirectories.
  - **Acyclic-graph Directory:** Allows shared files via links.
  - **General Graph Directory:** Allows cycles, requires special handling.
- Directory entries store:
  - File name
  - Pointer to file location
  - File attributes (size, timestamps, permissions)

---

5. File Sharing

- Multiple users or processes accessing files simultaneously.
- Requires:
  - **Access control:** Define who can read, write, or execute.
  - **Consistency control:** Manage concurrent access (locking, versioning).
  - Sharing can be **read-only**, **write-only**, or **read-write**.

---

6. File System Implementation Issues

- Efficient **mapping** of files to disk blocks.
- Managing **metadata** (file attributes, directory info).
- Handling **disk fragmentation** and **free space**.
- Providing **reliability** and **recovery** mechanisms.
- Balancing **performance** with **complexity**.

---

7. File System Protection and Security

- Protect files from unauthorized access or modification.
- Mechanisms include:
  - **Permissions:** Read, write, execute permissions for user, group, others.
  - **Access Control Lists (ACLs):** Fine-grained permissions per user.
  - **Encryption:** Secure data on disk.
  - **Authentication:** Verify user identity before access.
  - **Auditing:** Track who accessed or modified files.