OS

Theory

Definition and Purpose

- Manages computer resources(hardware & software)
- Provides environment for program execution
- Hides hardware complexity (Abstraction)
- Acts as Resource Manager (Arbitration)
- Facilitates app execution (Isolation & Protection)

Why OS?

- Made up of System Software Collection
- Without OS:
    - Bulky & complex apps (hardware interaction in app)
    - Resource exploitation by one app
    - No memory protection

OS Goals

- Maximum CPU Utilization
- Less Process Starvation
- Higher priority job execution

Types of Operating Systems

Singe Process OS

- Only 1 process executes at a time from the ready queue

Batch Processing OS

- User prepares jobs (punch cards)
- Operator collects & sorts jobs into batches
- Operator submits batches to processor
- All jobs in batch execute together

- Limitations:
  - Priorities cannot be set (if a job comes with some higher priority)
  - May lead to starvation (A batch may take more time to complete)
  - CPU idle during I/O

Multiprogramming OS

- Increases CPU Utilization
- Keeps multiple jobs in memory
- Single CPU
- Context switching when process waits
- Reduced CPU idle time

Multitasking OS

- Logical extension of multiprogramming
- Single CPU
- Runs more than one task simultaneously
- Context switching & time sharing are used
- Increases responsiveness
- Further reduced CPU idle time

Multi-Processing OS

- More than 1 CPU in single computer
- Increases reliability
- Better throughput (Throughput refers to the **number of processes completed per unit of time** by the system)
- Lesser process starvation

Distributed OS

- Manages many bunches of resources, >=1 CPUs, >=1 memory, >=1 GPUs, etc
- Loosely connected autonomous nodes
- Collection of independent, networked, communicating, and physically separate computational nodes

Real Time OS (RTOS)

- Real-time error-free computations
- Within tight time boundaries
- Examples: Air Traffic Control, Robots

Program, Process, Thread

Program

- It is an Executable file
- Contains set of instructions for specific job
- It's a compiled code, ready to execute
- Stored in Disk

Process

- Program under execution
- Resides in RAM

Thread

- Single sequence stream within a process
- Independent path of execution in a process
- Light-weight process
- Achieves parallelism by dividing process tasks
- Examples: Browser tabs, Text editor features (spell-check, format, save concurrently)

Multitasking vs Multithreading

Multitasking

- Execution of >1 task simultaneously
- Concept of >1 processes context switched
- No. of CPU = 1
- Isolation & memory protection exist
- OS allocates separate memory/resources per program
- Example: Playing music + using MS Word.

Multithreading

- Process divided into sub-tasks (threads). Each thread has own execution path

- Concept of >1 thread, threads context switched
- No. of CPU >=1 (better with >1)
- No isolation & memory protection, resources shared
- OS allocates memory to process, threads share it
- Example: In Word, spell check runs while you type.

Thread Scheduling

- Threads are schedules for execution based on priority. Each thread is assigned a priority level; the OS uses these priorities to decide which thread gets CPU time first. Higher priority threads are scheduled before lower ones.
- The OS allocates small chunks of CPU time (called time slices) to threads, ensuring that even lower-priority threads get some execution time (especially in **preemptive scheduling**)

Context Switching

Thread Context Switching

- OS saves current thread state, switches to another in same process
- Doesn’t includes switching of memory address space. (But Program counter, registers & stack are included.)
- Fast switching
- CPU's cache state preserved

Process Context Switching

- OS saves current process state, switches to another process
- Includes switching of memory address space
- Slow switching
- CPU's cache state flushed.

OS Components

User → Shell → Kernel → Hardware

User Space

- Application software runs here
- Apps don't have privileged hardware access
- Interacts with kernel

- GUI (Graphical User Interface)
- CLI (Command Line Interface)

Shell (Command Interpreter)

- Receives commands from users
- Gets commands executed by OS
- Shell is like the **translator**. You type a command (`ls` , `dir`) → shell takes it → asks the kernel → kernel executes.

Kernel

- Interacts directly with hardware
- Performs critical tasks
- Heart/Core component of OS
- Fist part of OS to load on start-up

Kernel Functions

- **Process Management**
  - Scheduling process threads
  - Creating & deleting user/system processes
  - Suspending & resuming processes
  - Providing mechanisms for sync/communication
- **Memory Management**
  - Allocating & deallocating memory
  - Tracking memory usage
- **File Management**
  - Creating & deleting files
  - Creating & deleting directories
  - Mapping files to secondary storage
  - Backup support
- **I/O Management**
  - Manages & controls I/O operation/devices
  - Buffering (data copy between devices, within one job)
  - Caching (memory, web caching)
  - Spooling (between differing speed jobs, e.x, mail spooling, print spooling (When multiple print requests are received, the OS stores them in a disk

queue and sends them to the printer one by one))

Types of Kernels

Monolithic Kernel

- All functions in kernel
- Bulky in size
- High memory required
- Less reliable (one module crash = whole kernel down)
- High Performance (less user/kernel mode overhead)
- Examples: Linux, Unix, MS-DOS

Microkernel

- Only major functions in kernel (Memory, Process Management)
- File & I/O Management in User-space
- Smaller in size
- More reliable & Stable
- Slower Performance (overhead switching between modes)
- Examples: L4 Linux, Symbian OS, MINIX

Hybrid Kernel

- Combines advantages of both (Monolithic Kernel + Microkernel)
- File Management in User space, rest in Kernel space
- Speed/design of monolithic kernel, modularity/stability of microkernel
- Lesser IPC (Inter-Process Communication) overheads
- Examples: MacOS, Windows 10

Nano/Exo Kernels

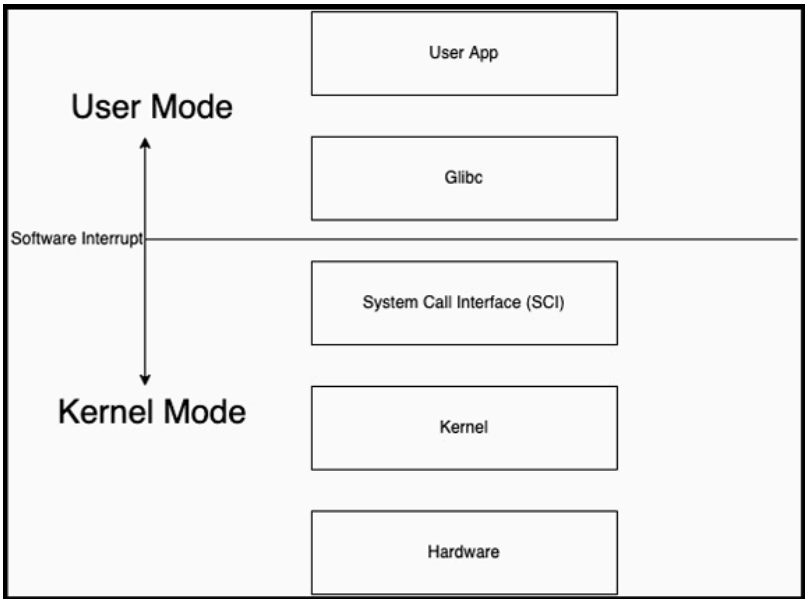
Inter-Process Communication (IPC)

- It's a mechanism used by an operating system to allow processes to interact and exchange data.
- Processes execute independently with memory protection (processes are isolated for memory safety). Some processes need to communicate.
- IPC is how user mode and kernel mode communicate.
    - **User mode** processes can't access hardware directly.

    - They **request services** from the **kernel** via **system calls**.
    - For IPC, system calls like `pipe()`, `shmget()`, `msgget()` etc., **let the kernel set up shared memory or message queues**.
    - Example: When a user mode process wants shared memory, it asks the kernel through `shmget()` → kernel sets it up → returns a reference to the process.
- Done by Shared Memory and Message Passing

System Calls

- Mechanism for apps to interact with Kernel
- User program requests services from kernel (lacks permission)
- Only way process goes from user mode to kernel mode
- Implemented in C
- Example: Mkdir (wrapper for file management system call)
- Transitions from User space to Kernel space via software interrupts



- **Types of System Calls:**
    - Process Control (end, abort, load, execute, create/terminate process, get/set process attributes, wait time/event, allocate/free memory)
    - File Management (create/delete file, open/close, read/write/reposition, get/set file attributes)
    - Device Management (request/release device, read/write/reposition, get/set device attributes, attach/detach devices)
    - Information Maintenance (get/set time/date, get/set system data, get/set process/file/device attributes)

    - Communication Management(create/delete connection, send/receive messages, transfer status, attach/detach remote devices)

Computer Boot Process

- PC On
- CPU initializes, looks for firmware (BIOS/UEFI) in ROM chip. (Firmware is software embedded in hardware devices that provides low-level control, enabling them to operate)
- BIOS/UEFI runs
    - Tests & initializes hardware (This is called: POST - Power on self-test process)
    - Loads configuration settings
    - If something is not appropriate (like missing RAM) error is thrown and boot process is stopped.
    - UEFI (Unified Extensible Firmware Interface) more advanced, tiny OS itself.
- BIOS/UEFI hands off to Bootloader
    - Looks at MBR (Master Boot Record) or EFI system partition. The MBR contains code that loads the rest of the operating system, known as a “bootloader.” MBR is present at the beginning of the disk
    - Executes bootloader code
- Bootloader runs
    - Small program, boots rest of OS
    - Boots Kernel, then User Space
    - Ex: Windows Boot Manager, GRUB (Linux), boot.efi (Mac)
- **Booting** is the sequence of events that occurs when a computer is powered on or restarted

32-Bit vs 64-Bit OS

32-bit OS

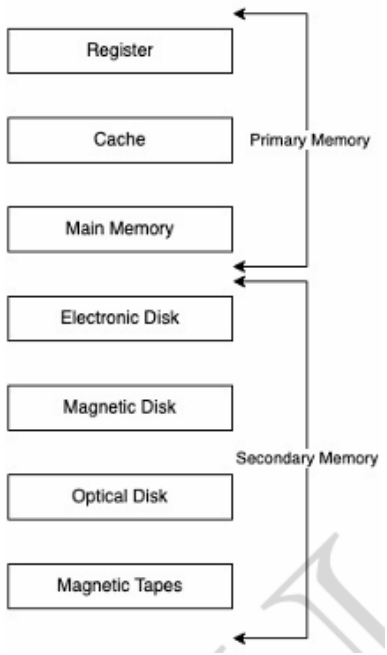
- 32-bit registers
- Accesses 2^32 unique memory addresses (4GB physical memory)
- 32-bit CPU processes 32 bits/instruction cycle

64-bit OS

- 64-bit registers
- Accesses 2^64 unique memory addresses (17EB physical memory)
- 64-bit CPU processes 64 bits/instruction cycle
- **Advantages of 64-bit**
    - More addressable memory
    - Better resource usage (utilizes more RAM)
    - Improved Performance (larger calculations per cycle)
    - Compatibility (runs both 32-bit & 64-bit OS)
    - Better graphics performance

Storage Devices Basics

Types of Memory

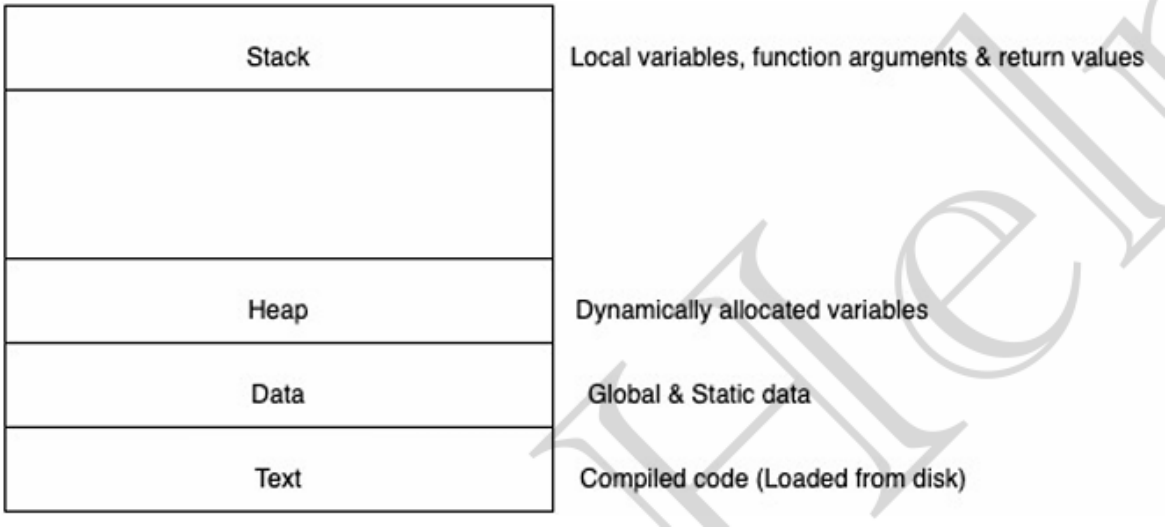
- Registers (part of CPU itself, holds an instruction, a storage address, or any data (such as bit sequence or individual characters), fast, expensive)
- Cache (very fast, stores frequently used data/instruction, faster than main memory)
- Main Memory (RAM, primary, volatile, costly)
- Secondary Memory (storage of data, non-volatile, cheaper, large)
- 

Comparison Metrics

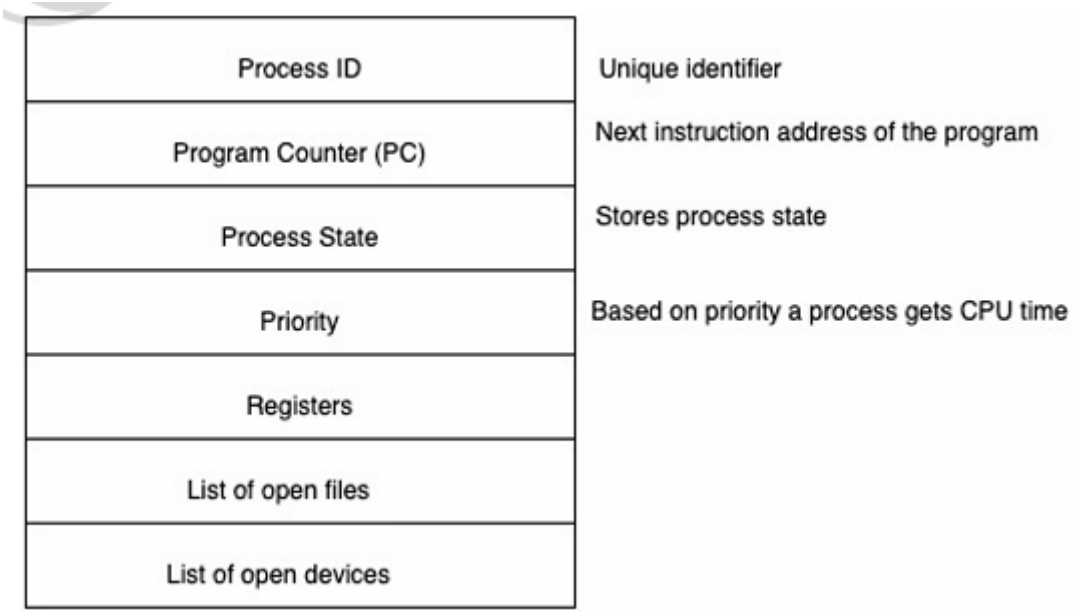
- Cost (Registers > Cache > Main > Secondary)
- Access Speed (Register > Cache > Main > Secondary)
- Storage Size (Secondary > Main > Cache > Register)

- Volatility (Primary-volatile, Secondary-non-volatile)

Introduction to Process

- Program: Compiled code, ready to execute
- Process: Program under execution
- **OS creates Process steps:**
  - Loads program & static data into memory
  - Allocate runtime stack
  - Heap memory allocation
  - I/O tasks
  - OS hands off control to main()
  - 

- **Attributes of Process**
  - Uniquely identifies process
  - Process Table
    - OS tracks all processes
    - Each entry in Process Control Block (PCB)
  - PCB (Process Control Block)
    - Data structure stores process info (ID, PC, state, priority)
    - Stores register values during context switch

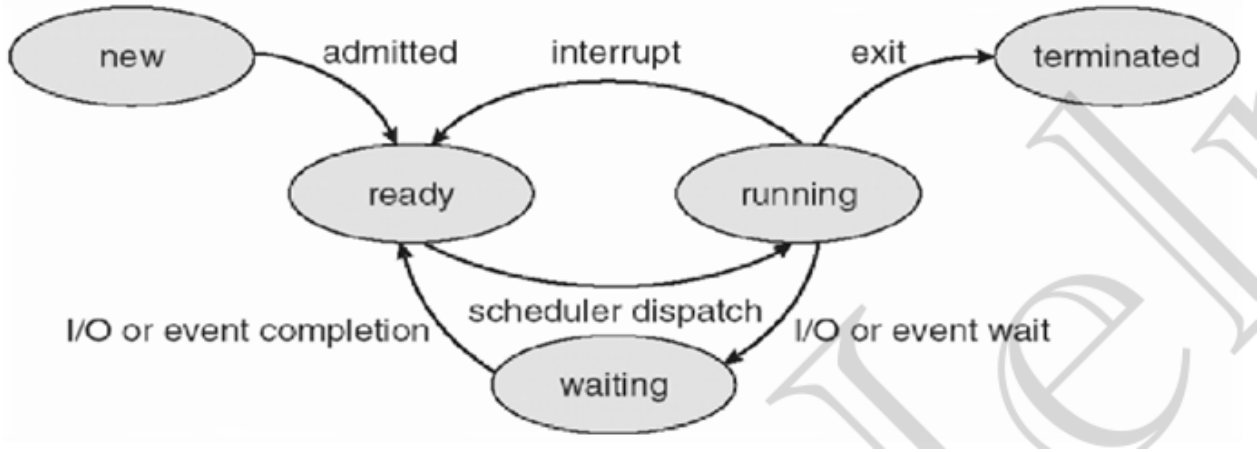


- **Explanation:**
  - So, in every operating system, there are multiple processes, like little mini-programs, doing their thing, right? But the OS isn't just winging it. It needs to keep track of all these moving pieces in a super-organized way.
  - That's where the **Process Table** steps in. Think of it like an Excel sheet the OS maintains where each row holds info about one process. And each of those rows? That's called a **PCB**, short for **Process Control Block**.
  - Now the PCB is kind of the process's personal diary. What does it store?
    - Its **unique ID** (so the OS can recognize it),
    - Its **current state** (like, is it running or just chilling waiting in line),
    - The **Program Counter** (i.e., what line of code it's on),
    - And even **CPU register values**, which is like remembering where it left off before going for a coffee break (aka context switch).
  - Context switching is when the OS has to pause one process and switch to another, and it needs to save all the info in the PCB to come back to it later without messing up.
  - So basically, each PCB helps the OS multitask without getting confused. It's like juggling multiple tabs in your browser and remembering exactly where you left off on each one.

Process States & Queues

Process States

- New (process being created)
- Run (instructions executed, CPU allocated)
- Waiting (waiting for I/O)
- Ready (in memory, waiting for processor)
- Terminated (finished execution, PCB removed)



Process Queues

- **Job Queue:**
  - Processes in New state
  - Present in Secondary Memory
  - Job Scheduler (Long-Term Scheduler) loads to memory
- **Ready Queue:**
  - Processes in Ready state
  - Present in Main Memory
  - CPU Scheduler (Short-Term Scheduler) dispatches to CPU
- **Waiting Queue (Processes in Wait state)**

Schedulers

- In an operating system, scheduling is all about deciding **which process gets to run when**. There are different types of schedulers for different stages of this decision-making.
- First, there's the **Long-Term Scheduler**. You can think of it as the gatekeeper. It controls which processes are allowed into the system for execution. Its main job is to **manage the degree of multiprogramming**, i.e., how many

processes get to hang out in memory at once. Since this decision doesn't need to be made super often, the long-term scheduler runs **infrequently**.
- Then comes the **Short-Term Scheduler**, also called the CPU scheduler. This one's all about **who gets the CPU next**. It picks one process from the ready queue and gives it access to the CPU. And because processes finish or pause often, this scheduler works **very frequently**, handling **context switches** and making sure CPU time is used efficiently.
- The **Medium-Term Scheduler** acts like a **balancer** between the long-term and short-term schedulers. Its main job is to **temporarily suspend processes** to improve CPU and memory efficiency. This is called **swapping**.

Degree of Multi-programming

- Number of processes in memory
- Controlled by Long Term Scheduler

Dispatcher

- It is an OS module
- It gives CPU control to process selected by Short-Term Scheduler

Swapping, Context Switching, Orphan/Zombie Process

Swapping

- Medium Term Scheduler (MTS) removes processes from memory
- Reduces degree of multi-programming
- Reintroduces processes, continues execution
- Swap-out (memory to disk) & swap-in (disk to memory) by MTS
- Improves process mix, frees memory

Context Switching

- State save of current process, state restore of new process
- Kernel saves old PCB context, loads new PCB context
- Pure overhead (no useful work is being done)
- Speed varies by machine (memory speed, registers copied)

Orphan Process

- Parent process terminated, child still running

- Adopted by init process (first OS process)

Zombie Process (Defunct Process)

- Execution completed, but entry still in process table
- Child processes (parent needs to read exit status via wait() call)
- Eliminated (reaped) after parent reads exit status

Process Scheduling & Algorithms

Process Scheduling

- It is an OS's mechanism for deciding **which process gets the CPU** and **when** especially when there are multiple processes in the ready queue.
- Basis of Multi-programming OS
- OS makes computer productive by switching CPU-using processes

CPU Scheduler (Short-Term Scheduler)

- Selects process from ready queue when CPU idle

Scheduling Types

- **Non-Preemptive Scheduling**
  - Process keeps CPU until it terminates or waits
  - Can cause starvation for short jobs
  - Low CPU utilization
- **Preemptive Scheduling**
  - CPU taken away after time quantum expires, or wait/terminate
  - Less starvation
  - High CPU utilization

Goals of CPU Scheduling

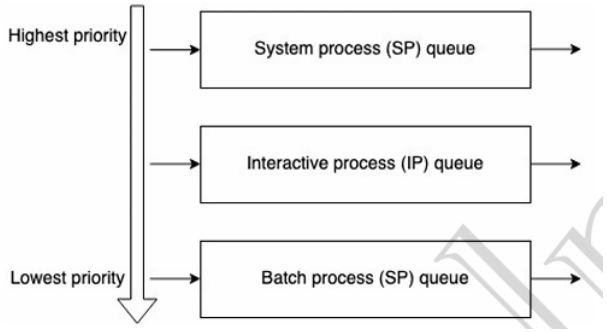
- Maximum CPU Utilization
- Minimum Turnaround Time (TAT = CT - AT)
- Minimum Wait Time (WT = TAT - BT)
- Minimum Response Time
- Maximum Throughput (processes completed per unit time)

Key Metrics

- **Throughput:** No. of processes completed per unit time.
- **Arrival time (AT):** Time when process is arrived at the ready queue.
- **Burst time (BT):** The time required by the process for its execution.
- **Turnaround time (TAT):** Time taken from first time process enters ready state till it terminates. (CT - AT)
- **Wait time (WT):** Time process spends waiting for CPU. (WT = TAT – BT)
- **Response time:** Time duration between process getting into ready queue and process getting CPU for the first time.
- **Completion Time (CT):** Time taken till process gets terminated

Scheduling Algorithms

- **FCFS (First Come-First Serve)**
  - Process first in queue gets CPU first
  - **Convoy Effect**
    - Process with long BT (Burst Time) blocks short ones
    - Poor resource management
- **Shortest Job First (SJF)**
  - Least BT (Burst Time) process dispatched first
  - Requires BT (Burst Time) estimation (difficult)
  - Non-preemptive: suffers convoy effect, starvation
  - Preemptive: less starvation, no convoy effect, lower average WT (Waiting Time)
  - Criteria: Arrival Time + Burst Time
- **Priority Scheduling**
  - Priority assigned at creation
  - SJF (Shortest Job First) is a special case with priority inversely proportional to BT (Burst Time)
  - Preemptive: higher priority job preempts current job
  - Indefinite waiting (starvation) for lower priority
    - **Solution:** Aging (gradually increase priority of waiting processes)
- **Round Robin (RR)**
  - Most popular, like FCFS but preemptive
  - Designed for time-sharing systems
  - Considers AT (Arrival Time) + Time Quantum (TQ), not BT (Burst Time) dependent

    - Very low starvation (no process waits forever)
    - No convoy effect
    - Easy to implement
    - Small TQ (Time Quantum) -> frequent context switches (more overhead)
- **Multi-Level Queue (MLQ)**
    - Ready queue divided into multiple queues by priority
    - Processes permanently assigned to a queue (inflexible)
    - Each queue has its own scheduling algorithm (e.g., RR, SJF, FCFS)
    - Scheduling among sub-queues is fixed priority preemptive
    - Problems: Starvation for lower priority processes, Convoy effect present
    - 

- **Multi-Level Feedback Queue (MLFQ)**
    - Multiple sub-queues
    - Processes can move between queues (flexible)
    - Separates processes by BT (Burst Time) characteristics (long BT (Burst Time) to lower priority)
    - I/O bound interactive processes stay in higher priority because they can complete their tasks, and return to waiting for I/O, **freeing up CPU for others**
    - Aging: Process waiting too long moves to a higher priority (prevents starvation)
    - Less starvation than MLQ
    - Configurable to system design

Concurrency

- Execution of multiple instruction sequences at the same time
- Several process threads running in parallel
- **Benefits of Multi-threading:**
    - Responsiveness
    - Resource Sharing (efficient)

    - Economy (cheaper to create/switch threads vs processes)
    - Utilizes multiprocessor architectures better
- **How each thread get access to the CPU?**
    - Each thread has its own program counter.
    - Depending upon the thread scheduling algorithm, OS schedule these threads.
    - OS will fetch instructions corresponding to PC of that thread and execute instruction
- Single CPU system does not gain from multi-threading (As two threads have to context switch for that single CPU. This won't give any gain)
- **Thread Control Block (TCB)** for state storage during context switching

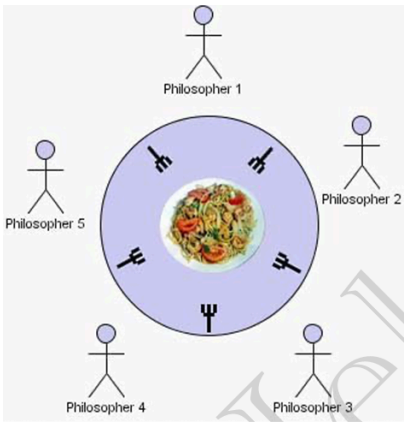
Critical Section Problem & Solutions

- Process synchronization maintains shared data consistency
- **Critical Section (C.S.):**
    - Code segment where processes/threads access/write shared resources
    - Concurrent execution means interruption is possible
- **Major Thread Scheduling Issue: Race Condition**
    - Two or more threads access/change shared data simultaneously
    - The result depends on the thread scheduling order
- **Solutions to Race Condition:**
    - Atomic operations (C.S (Critical Section) executed in one CPU cycle)
    - **Mutual Exclusion** using Locks (Mutex)
        - Ensures **only one** thread/process can access the C.S (Critical Section) at a time.
        - **Disadvantages:**
            - **Contention**: If one thread holds the lock, others must wait. If that thread crashes or never releases it, everyone else is stuck forever.
            - **Deadlocks**: Two or more threads wait on each other’s locks, and no one proceeds.
            - **Debugging issues**: Hard to trace and reproduce lock-related bugs.
            - **Starvation**: High-priority threads may wait indefinitely if lower-priority threads keep getting the lock.
    - **Semaphores**

        - A semaphore is a **counter** that helps manage access to resources in a multi-threaded program.
        - The counter's value = **how many resources are available**.
        - If the value > 1 → **multiple threads** can enter the critical section at the same time.
        - **Binary semaphore** (0 or 1) → works like a lock (mutex) for one-at-a-time access.
        - **Counting semaphore** (>1) → lets several threads use the resource together, up to the limit.
        - To avoid wasting CPU (busy waiting), if a thread can't enter, it's **blocked** and put in a **waiting queue**.
        - When another thread finishes and calls `signal()`, the waiting thread is **woken up** and moved to the **ready queue** so it can run.
    - Peterson's solution (for 2 processes/threads only)
    - Cannot use a simple flag variable
- **Conditional Variable:**
    - A **synchronization primitive** used for waiting until a certain condition is true. (Think of a **synchronization primitive** as the **most basic "tool"** your system gives you to control how multiple threads or processes interact with shared data, so they don't mess each other up. Ex: Mutex, Semaphore, etc.)
    - Always works **together with a lock**.
    - Flow: Thread gets the lock → checks condition → if not met, calls `wait()` (releases lock + goes to sleep) → wakes up when `notify()` / `signal()` is called → re-acquires the lock and continues.
    - **Avoids busy waiting** – thread sleeps instead of looping.
    - **No contention time** – lock is free for others while the thread waits.

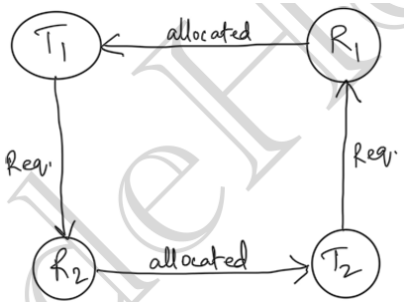
Dining Philosophers Problem:

- **Setup:** 5 philosophers, 5 chairs, a bowl of noodles, 5 single forks (arranged circularly).

- 

- **States:** Thinking, Eating.
- **Eating Process:**
  - Needs 2 adjacent forks (left & right).
  - Picks one at a time.
  - Cannot pick up a fork that is already taken.
  - Eats without releasing forks once acquired.
- **Solution using Semaphores:**
  - Each fork is a binary semaphore.
  - Uses `wait()` and `signal()` operations.
- **Problem:** Can lead to **Deadlock**.
  - All 5 philosophers pick up their left fork, then wait forever for their right fork (a circular wait).
- **Methods to Avoid Deadlock:**
  - Allow at most 4 philosophers to eat simultaneously.
  - Pick up both forks in a critical section (atomically).
  - Odd-even rule: Odd picks left then right, even picks right then left.
- **Semaphores alone are not enough, need enhancement rules.**
  - This means even though semaphores can **control access** to forks (so that no two philosophers use the same fork at the same time), they **don't automatically prevent deadlock**.
  - Why? If every philosopher uses a `wait()` on their left fork first, then tries for the right, all can get stuck in a **circular wait**. This is a *deadlock*.
  - The semaphore enforces mutual exclusion on each fork, but **doesn't control the overall sequence** in which forks are picked up.
  - So, to actually make the dining philosophers problem safe: You **still use semaphores**, but you **add rules** like "maximum 4 philosophers at the table" or "odd-even picking order" to break at least one of the **necessary conditions for deadlock** (like circular wait).

Deadlock

- Process compete for finite resources
- Process waits for resource, never changes state (resource busy forever)
- Two or more processes waiting on resources indefinitely
- Bug in process/thread synchronization
- Process never finish, resources tied up
- Examples of resources: memory, CPU cycles, files, locks, I/O devices
- 

- **Resource Utilization Steps:**
  - Request (resource, if free, lock; else wait)
  - Use
  - Release (make available)
- **Deadlock Necessary Conditions (All must hold simultaneously)**
  - Mutual Exclusion (only 1 process uses resource at a time)
  - Hold & Wait (process holds >=1 resource, waits for more held by others)
  - No-Preemption (resource voluntarily released by process)
  - Circular Wait ({P0, Pn}, P0 waits for P1, P1 for P2, ...Pn for P0)
- **Methods for Handling Deadlocks**
  - Prevention or Avoidance (system never enters deadlocked state)
  - Detection & Recovery (system enters, then detects/recovers)
  - Ignore problem (Ostrich algorithm / Deadlock ignorance)

Deadlock Prevention (ensure at least one condition cannot hold)

- **Mutual Exclusion:** Apply locks only to resources that are inherently non-shareable, since mutual exclusion cannot be avoided for such fundamentally non-shareable resources.
- **Hold & Wait:**
  - Request/allocate all resources before execution.
  - Request resources only when none held, release all current first.
- **No Preemption:**

    - If a process can't get new resources, all its held resources are preempted.
    - If a resource is allocated to a waiting process, preempt and give to a requesting process.
- **Circular Wait:** Impose a proper ordering of resource allocation.

Deadlock Avoidance

- **General Approach:**
    - OS needs prior knowledge of each process's maximum resource needs.
    - Before granting a request, the system checks current availability, existing allocations, and potential future needs.
    - Resources are allocated in a way that keeps the system in a **safe state**.
- **Safe State:**
    - There exists some order in which all processes can obtain the resources they need (up to their maximum) and finish execution.
    - Being in a safe state guarantees no deadlock will occur.
- **Unsafe State:**
    - May result in deadlock, but deadlock is not certain.
- **Banker's Algorithm:**
    - Evaluates if a resource request will keep the system in a safe state.
    - If granting the request would lead to an unsafe state, the process must wait.

Deadlock Detection

- Used in systems that do **not** implement deadlock prevention or avoidance.
- **Single instance per resource type:**
    - Use a *Wait-For Graph*. If a cycle exists, deadlock has occurred.
- **Multiple instances per resource type:**
    - Use a variation of the *Banker's Algorithm* to detect deadlock.

Recovery from Deadlock

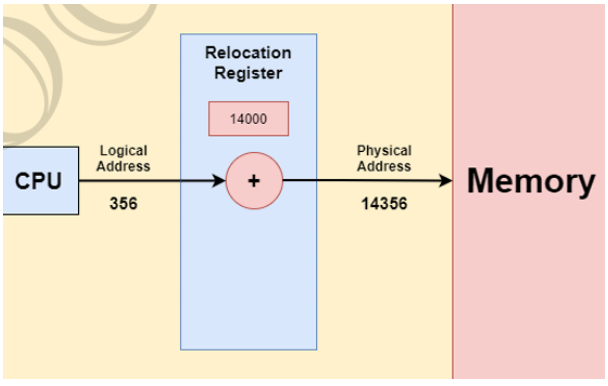
- **Process Termination:**
    - Abort all deadlocked processes.
    - Abort one process at a time until the deadlock cycle is eliminated.
- **Resource Preemption:**

- Successively preempt resources from processes.
- Give resources to others until the deadlock cycle is broken.

Memory Management Techniques

- Responsible for handling main memory allocation among multiple processes.

Logical vs Physical Address Space

- **Logical Address (Virtual Address):**
  - Generated by the CPU and used within a process.
  - Accessed indirectly by the user/program.
  - The collection of all logical addresses forms the *Logical Address Space*.
- **Physical Address:**
  - Actual location in main memory, stored in the memory address register.
  - Never accessed directly by the user.
  - Determined by the Memory Management Unit (MMU).
  - The collection of all physical addresses forms the *Physical Address Space*.
- **MMU (Memory-Management Unit):**
  - Hardware component that performs runtime mapping from logical (virtual) addresses to physical addresses.
- 

- **Explanation:**
  - **Scenario:** You run a program, and it wants to read the value at **logical address 1200**.
  - **Step-by-step:**
    1. **Logical Address:**
       - The CPU generates `1200` as the logical address.
       - This is the address *as seen by your program*, it doesn't correspond to an actual hardware location yet.
    2. **MMU Mapping:**

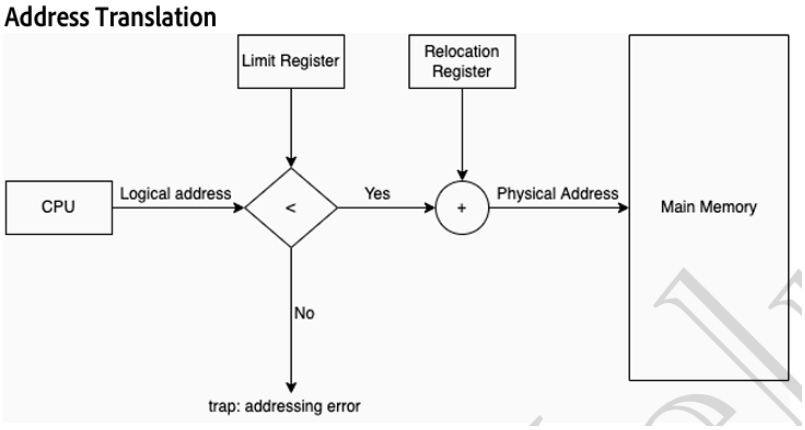
- The **Memory Management Unit** maps this logical address to a **physical address** in RAM.
- Suppose the base address assigned to the process in RAM is **30000**.
- MMU computes:

```
Physical Address = Base Address + Logical Address
Physical Address = 30000 + 1200 = 31200
```

3. **Physical Address:**
   - The data is fetched from **RAM location 31200**.
   - The program never knows this actual address, it only knows “I read from address 1200.”

Memory Mapping & Protection

- OS provides a **Virtual Address Space (VAS)** to each process.
- Defines the **legal address range** a process can access.
- Uses a **Relocation Register** (base address) and a **Limit Register** (size range).
- A logical address is valid only if it is **less than the limit register** value.
- The **MMU** (Memory Management Unit) maps the logical address to a physical address by **adding the relocation register value**.
- On a context switch, the **dispatcher** updates the relocation and limit registers for the incoming process.
  - **Explanation:**
    - **Process P1** is running.
      - Relocation register = base address where P1 is loaded in RAM (e.g., 5000).
      - Limit register = size of P1’s allocated memory.
    - **Context switch** happens (P1 → P2).
      - Dispatcher saves P1’s CPU state.
      - Dispatcher **loads P2’s relocation and limit registers** with P2’s memory mapping info.
    - When **P2** runs:
      - Any logical address it generates (like 100, 200…)
      - Goes through the **MMU**, which adds the relocation register value (say 8000 for P2) to get the physical address.

    - So, the dispatcher's job is to **set the mapping parameters** (relocation/limit registers) so the MMU can translate addresses correctly for the currently running process.
- Prevents a process from accessing OS memory or other processes' memory. Illegal access triggers a **trap**.
- 

Allocation Methods on Physical Memory

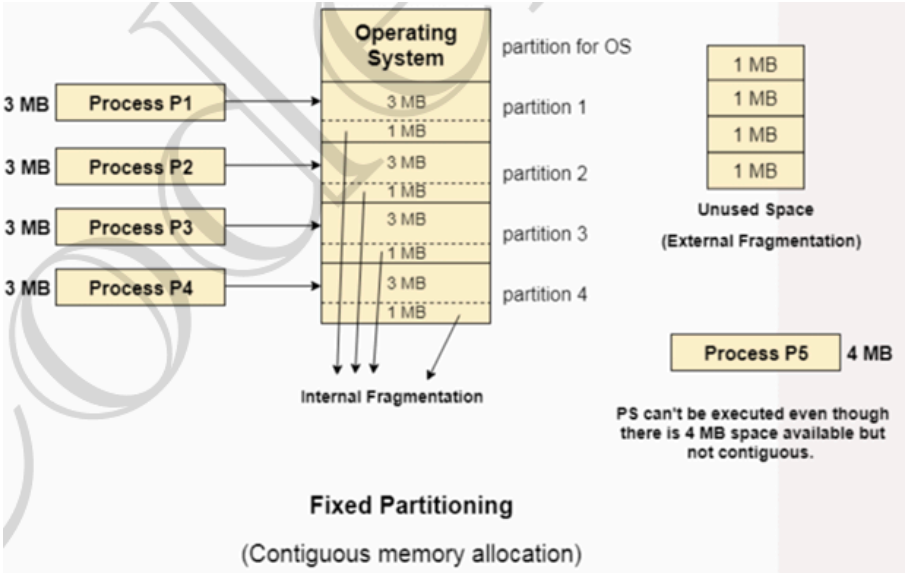
- Contiguous Allocation
- Non-contiguous Allocation

Contiguous Memory Allocation

- Each process gets **one continuous block** of memory in RAM.

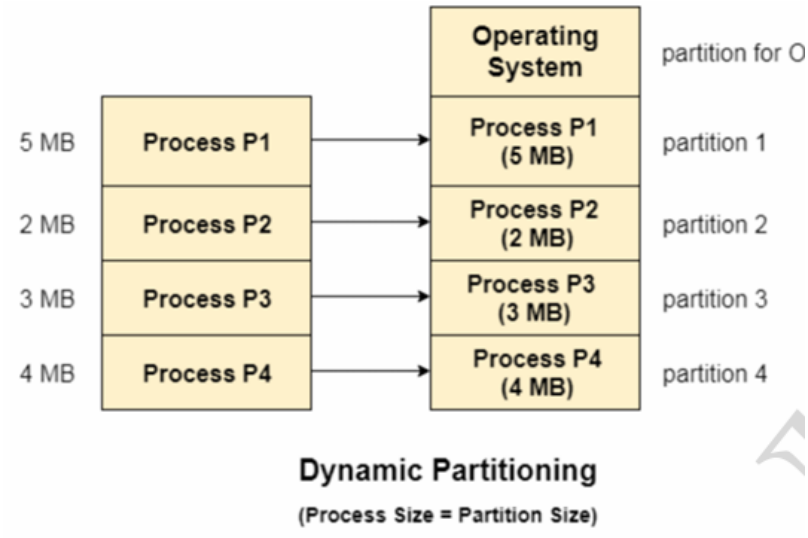
Fixed Partitioning

- Main memory is divided into **partitions of fixed size**.
- **Limitations:**
    - **Internal Fragmentation**: wasted space inside a partition.
        - **External Fragmentation**: free memory exists but not in one continuous block.
        - Process size cannot exceed partition size.
        - Fixed number of partitions limits multi-programming.

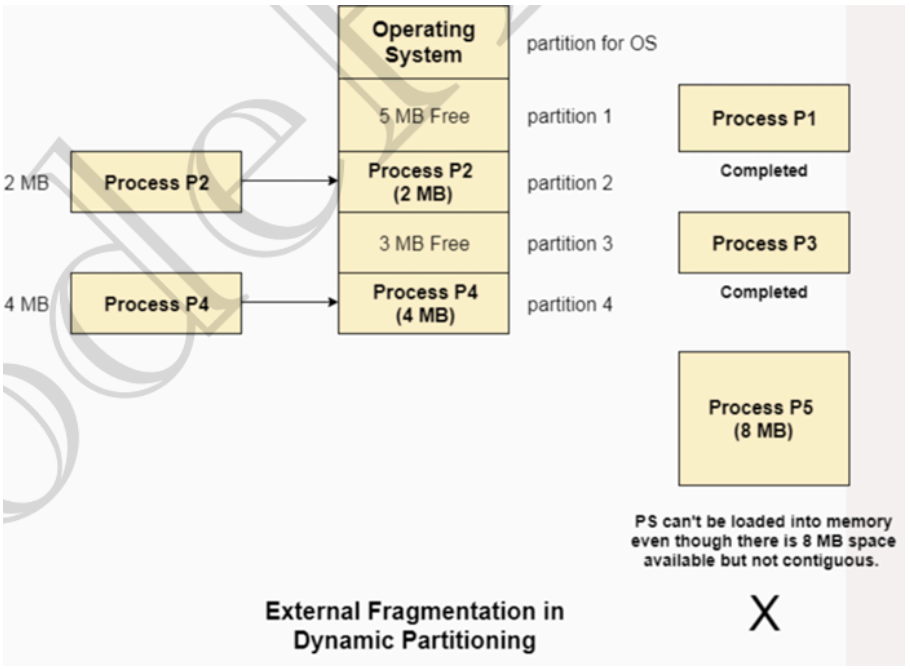


Dynamic Partitioning

- Partition size is **decided at process load time** based on its needs.



- **Advantages over Fixed:**
  - No internal fragmentation.
  - No strict limit on process size.
  - Better degree of multi-programming i.e. can support more processes in memory
  - **Limitation:**
    - External Fragmentation.

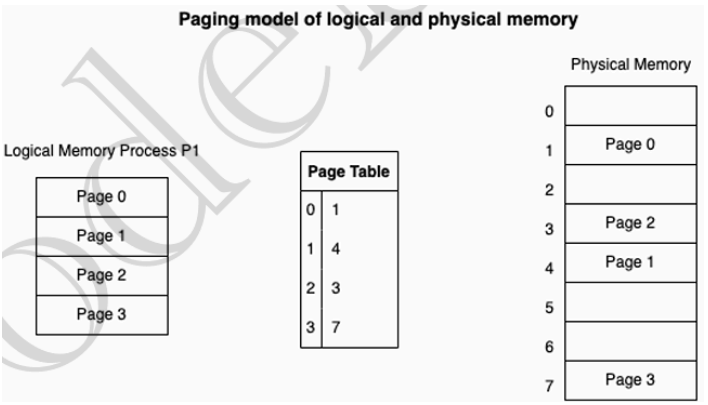


Free Space Management

- **Defragmentation/Compaction**
  - Reduces external fragmentation in dynamic partitioning.
  - Moves allocated partitions together and combines all free space into one large block.
  - Enables storing larger processes.
  - Lowers system efficiency due to extra data movement.
- **Free List Representation**
  - Free memory blocks are tracked using a **linked list**.
- **Hole Allocation Strategies:**
  - **First Fit:** Assigns the first hole large enough for the process. *(Fast)*
  - **Next Fit:** Similar to First Fit but starts searching from the **last allocated hole**.
  - **Best Fit:** Chooses the **smallest** hole that fits the process. *(Less internal fragmentation, slower, leaves many small holes)*
  - **Worst Fit:** Chooses the **largest** available hole. *(Slower, leaves fewer but larger holes)*

Paging (Non-Contiguous Memory Allocation)

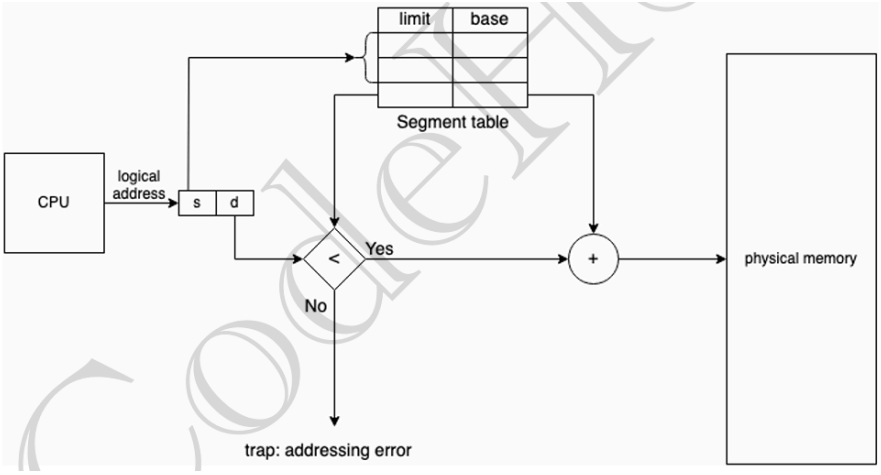
- Eliminates **external fragmentation** and avoids compaction overhead. (**Compaction overhead** refers to the **time and processing cost** the OS incurs when it rearranges processes in physical memory to make free space contiguous.)

- **Concept:**
  - Physical memory is divided into fixed-size **frames**.
  - Logical memory is divided into fixed-size **pages**.
  - **Page size = Frame size** (set by processor architecture).
- **Page Table:**
  - Stores mappings from page numbers to frame numbers.
  - Each entry holds the base address of the page in physical memory.
  - Stored in main memory; the base address of the table is kept in the PCB.
- **Logical Address Structure:**
  - **Page number (p):** Index to locate the page in the page table.
  - **Page offset (d):** Position within the page/frame.
  - 

- **Page Table Base Register (PTBR):** Points to the active page table.
- **Performance Optimization:**
  - **Translation Lookaside Buffer (TLB):** Hardware cache storing recent page mappings.
    - **TLB hit:** Mapping found in TLB.
    - **ASIDs (Address Space Identifiers):** Differentiate processes and improve protection.
- **Explanation:**
  - Process is split into **fixed-size pages** (e.g., 4 KB each).
  - Pages can be stored **anywhere** in physical memory — they don't need to be next to each other.
  - The **page table** keeps track of where each page is stored.
  - This avoids **external fragmentation**.
  - **Why non-contiguous?** → Page 0 might be in frame 5, Page 1 in frame 100, Page 2 in frame 42… all scattered around RAM.
  - The reason **paging** and **segmentation** are called **non-contiguous memory allocation** is that, even though we "split" memory, **the parts of**

a process don’t have to be stored together in one continuous chunk in physical memory.
    - **Analogy:** Paging = Cut a book into equal-sized pages, store them anywhere, use an index to find them

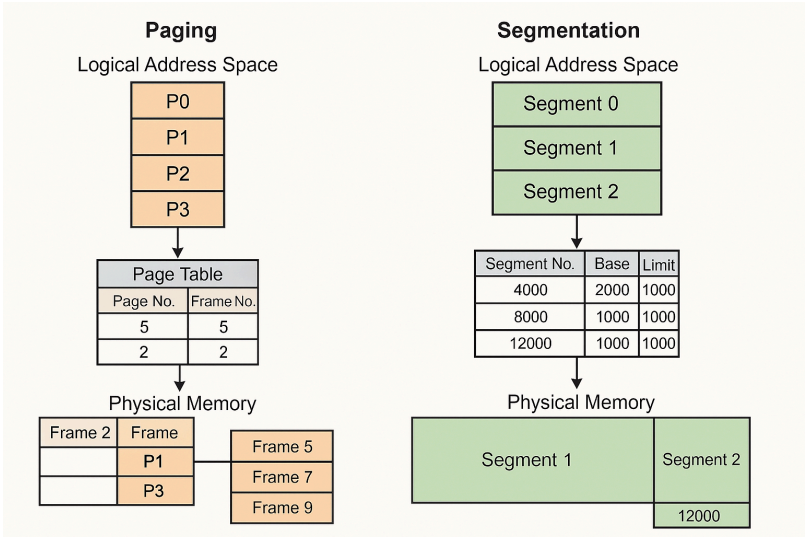
Segmentation (Non-Contiguous Memory Allocation)

- Matches the **user’s logical view** of memory.
- Logical address space is split into **segments**, each with a **segment number** and **offset (s, d)**.
- Segment size varies; segments may represent functions, arrays, or code modules.
- 

- **Advantages:**
    - No internal fragmentation.
    - Contiguous allocation within a segment allows efficient access.
    - Smaller segment table than a page table.
    - Easier recompilation: related functions remain in the same segment.
- **Disadvantages:**
    - External fragmentation.
    - Variable segment sizes complicate swapping.
- Modern systems use a hybrid approach (segmentation + paging).
- **Explanation:**
    - Process is split into **logical units** (segments) based on program structure. Ex., code, data, stack.
    - Segments can be **different sizes** and stored **anywhere** in memory.
    - The **segment table** keeps track of each segment’s location.
    - Can still suffer from **external fragmentation** because segments are variable-sized.

- **Why non-contiguous?** → Segment 0 might be in one area of RAM, Segment 1 somewhere else.
- **Analogy:** Segmentation = Store different *chapters* (code, data, stack) in different locations, sizes depend on chapter length.

Paging vs Segmentation



- **Paging:**
  - Both logical memory and physical memory are split into fixed-size blocks; **pages** (logical) and **frames** (physical).
  - Pages are mapped to frames via a **page table**.
  - The page number chooses the frame, and the page offset locates the exact byte within that frame.
- **Segmentation:**
  - Logical memory is split into **variable-sized segments** (e.g., code, data, stack).
  - Each segment is stored in a different part of physical memory.
  - The segment number selects the segment in the **segment table**, and the offset locates the exact byte within that segment.
- Key difference: **Paging uses equal-size blocks** (good for avoiding external fragmentation), while **Segmentation uses variable-size blocks** (good for reflecting program structure).

Virtual Memory

- Lets processes run even if they are not fully loaded into main memory.
- Uses secondary storage (swap space) to extend available memory.
- Gives the impression of having a very large main memory.

- **Advantages:**
  - Supports processes larger than physical memory.
  - Enables more processes to run at once, even with limited RAM.
  - Improves CPU utilization and overall throughput.
  - Allows large applications to run on systems with less physical memory.
  - Increases the degree of multiprogramming.
- **Disadvantages:**
  - Performance may slow down due to swap operations.
  - Risk of **thrashing** (excessive paging/swapping).

Demand Paging

- A **popular virtual memory management method**.
- **Least used pages** stored from secondary memory.
- **Page is copied to main memory only** when demanded (page fault occurs).
- Implements a **lazy swapper** approach - Pages are swapped in only when needed, managed by a pager that handles individual pages.
- **How it works:**
  - The pager loads only required pages, skipping unused ones.
  - Reduces swap time and saves physical memory.
  - **Valid–Invalid Bit (in page table):**
    - `1` → Page is in memory and accessible.
    - `0` → Page is either invalid or stored on disk.
  - Accessing a `0` bit causes a **page fault** (control switches to the OS).
  - **A page fault** happens **any time** the CPU requests a page that is not currently in memory, whether or not there's a free frame.
- **Page Fault Handling Procedure:**
  - Check the process control block (PCB) for the reference's validity
  - If invalid → terminate process
  - If valid but on disk → locate a free frame from the free-frame list
  - Schedule a disk operation to read the page into that free frame
  - Update the page table entry (set valid bit to `1`)
  - Resume the interrupted instruction.
- **Pure Demand Paging:** Starts with **zero pages** in memory; every first access to a page causes a fault.

- **Performance Basis:** Relies on the **principle of locality of reference** (most programs access a small set of pages repeatedly).

Page Replacement Algorithms

- Decides which memory page to replace when page fault occurs & no free frames
- Aim: minimize the number of page faults.
- **Types of Algorithms:**
  - **FIFO (First-In, First-Out):**
    - Replaces the oldest loaded page.
    - Simple to implement using a queue.
    - May remove a frequently used page, hurting performance.
    - **Belady's Anomaly:** In some cases, increasing the number of frames leads to *more* page faults (counterintuitive).
      - **Example:**
        - Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
        - With **3 frames**, FIFO → 9 page faults.
          - 1 → PF (frames: 1)
          - 2 → PF (1, 2)
          - 3 → PF (1, 2, 3)
          - 4 → PF (2, 3, 4) — 1 is replaced
          - 1 → PF (3, 4, 1) — 2 is replaced
          - 2 → PF (4, 1, 2) — 3 is replaced
          - 5 → PF (1, 2, 5) — 4 is replaced
          - 1 → hit (1, 2, 5)
          - 2 → hit (1, 2, 5)
          - 3 → PF (2, 5, 3) — 1 is replaced
          - 4 → PF (5, 3, 4) — 2 is replaced
          - 5 → hit (5, 3, 4)
          - **Total page faults = 9**
        - With **4 frames**, FIFO → 10 page faults.
          - 1 → PF (1)
          - 2 → PF (1, 2)
          - 3 → PF (1, 2, 3)
          - 4 → PF (1, 2, 3, 4)

        - 1 → hit (1, 2, 3, 4)
        - 2 → hit (1, 2, 3, 4)
        - 5 → PF (2, 3, 4, 5) — 1 is replaced
        - 1 → PF (3, 4, 5, 1) — 2 is replaced
        - 2 → PF (4, 5, 1, 2) — 3 is replaced
        - 3 → PF (5, 1, 2, 3) — 4 is replaced
        - 4 → PF (1, 2, 3, 4) — 5 is replaced
        - 5 → PF (2, 3, 4, 5) — 1 is replaced
        - **Total page faults = 10**
- **Optimal Page Replacement:**
    - Removes the page that will not be used for the longest time in the future.
    - Lowest page fault rate
    - Impossible to implement in a real system (requires future knowledge).
- **LRU (Least Recently Used):**
    - Removes the page that hasn't been used for the longest time in the past.
    - **Implementation Ways:**
        - **Counters:** Store the last access time for each page; remove the one with the smallest timestamp.
        - **Stack:** Keep pages in a stack (or doubly linked list); move accessed pages to the top, remove from the bottom.
- **Counting-Based Page Replacement:**
    - Tracks how many times each page is accessed.
    - **LFU (Least Frequently Used):** Removes the page with the lowest reference count.
    - **MFU (Most Frequently Used):** Removes the page with the highest reference count.
    - Rarely used in practice due to overhead and poor adaptability to changing access patterns.

Thrashing

- Occurs when a process does not have enough frames for its actively used pages, leading to frequent page faults.

- Needed pages are replaced too soon, causing immediate faults again.
- Results in **excessive paging activity**.
- The CPU spends more time handling page faults than executing actual instructions.
- **Techniques to handle Thrashing:**
  - **Working Set Model:**
    - Relies on the concept of **Locality of Reference**.
    - Ensure each process has enough frames to hold all pages in its current locality.
  - **Page Fault Frequency:**
    - Regulates the page-fault rate by adjusting frame allocation.
    - If the rate is too high → allocate more frames.
    - If the rate is too low → reassign some frames elsewhere.
    - Uses defined upper and lower limits for acceptable fault rates.

Interview Questions

1. **What is a process and process table?**

A process is an instance of a program in execution. For example, a Web Browser is a process, and a shell (or command prompt) is a process. The operating system is responsible for managing all the processes that are running on a computer and allocates each process a certain amount of time to use the processor. In addition, the operating system also allocates various other resources that processes will need, such as computer memory or disks. To keep track of the state of all the processes, the operating system maintains a table known as the process table. Inside this table, every process is listed along with the resources the process is using and the current state of the process.

2. **What are the different states of the process?**

Processes can be in one of three states: running, ready, or waiting. The running state means that the process has all the resources it needs for execution and it has been given permission by the OS to use the processor. Only one process can be in the running state at any given time. The remaining processes are either in a

waiting state (i.e., waiting for some external event to occur such as user input or disk access) or a ready state (i.e., waiting for permission to use the processor). In a real operating system, the waiting and ready states are implemented as queues that hold the processes in these states.

3. **What is a Thread?**

A thread is a single sequence stream within a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. Threads are a popular way to improve the application through parallelism. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads, one thread to format the text, another thread to process inputs, etc.

- **Why are threads needed? What is it giving to the end user ? Why not use multiple process ?**
  - Threads are needed because they allow **concurrent execution** within the same process. They share the same memory space, which makes communication between them much faster compared to processes.
  - For the **end user**, this means:
    - Better **responsiveness** → e.g., a UI app can keep responding to clicks while another thread downloads data.
    - Better **performance** → tasks like file I/O, networking, and computation can overlap.
    - Efficient use of **multi-core CPUs**.
  - If we use **multiple processes instead of threads**:
    - Each process has its own memory → so inter-process communication (IPC) is slower and more complex.
    - Higher memory and context-switch overhead.
    - Not ideal for tasks that need frequent data sharing.
  - So, **threads are lightweight** compared to processes and are perfect when you need concurrency with shared data. Processes are safer for isolation, but threads give speed and responsiveness to the user.

4. **What are the differences between process and thread?**

- So, the main difference is that a **process** is an independent program in execution, whereas a **thread** is a smaller unit of execution that runs inside a process.
- A process has its **own memory space and resources**, so it's more heavyweight. On the other hand, threads are lightweight because multiple threads of the same process **share the same memory and resources**.
- For example, if two processes want to talk to each other, they need some form of **inter-process communication** like pipes or sockets, but threads can just communicate directly since they share memory.
- Also, context switching between processes is more expensive compared to threads. But one big drawback of threads is, if one thread crashes, it can affect the whole process, whereas with processes, if one fails, the others are usually safe.
- To give a simple example, think of a **web browser**: the browser itself is a process, and inside it you might have multiple threads; one handling rendering, one handling network requests, and one for JavaScript execution.

5. **What are the advantages and disadvantages of multithreading ?**

Multithreading is the ability of a process to be divided into multiple threads that execute concurrently. All threads share the same memory space (code, data, heap) but have their own program counter, stack, and registers. This enables tasks to run in parallel or at least appear parallel depending on CPU cores.
Advantages of Multithreading:

- **Better CPU Utilization:** If one thread is waiting (e.g., for I/O), another can execute, ensuring the CPU stays busy.
- **Improved Throughput:** Multiple tasks can progress simultaneously, increasing the overall efficiency of a process.
- **Responsiveness:** Applications (especially GUIs) remain responsive — for example, one thread handles the UI while another does background work.
- **Faster Communication:** Since threads share memory, communication between them is faster compared to inter-process communication.
- **Efficient Resource Usage:** Threads require less overhead to create and manage than processes, saving system resources.
- **Simpler Program Structure:** Complex tasks (like server handling multiple clients) can be broken into simpler thread routines, making design and coding

easier.

Disadvantages of Multithreading:

- **Synchronization Issues:** Shared memory access can cause race conditions, deadlocks, or inconsistent results if not managed carefully.
- **Context Switching Overhead:** If there are too many threads, frequent switching may degrade performance instead of improving it.
- **Shared Memory Risks:** Since threads share the same address space, a bug in one thread (like writing to invalid memory) can crash the entire process.

Examples:

- For example, let's say we have a **web server**. The server needs to handle multiple client requests at the same time. With multithreading, each incoming request can be assigned to a separate thread — so while one thread is waiting for a slow database query, another can continue serving a different client. This improves the responsiveness and throughput of the server.
- Another simple example is a **media player**: one thread can play the audio, another handles video rendering, and another listens for user interactions (pause, play, seek). All of these run together without blocking each other, giving a smooth experience.

6. **What is Thrashing?**

Thrashing is a situation when the performance of a computer degrades or collapses. Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults is necessary in order to appreciate the benefits of virtual memory, thrashing has a negative effect on the system. As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault. Thrashing occurs when processes on the system frequently access pages, not available memory.**For example:** Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

- With **3 frames**, FIFO → 9 page faults.
  - 1 → PF (frames: 1)
  - 2 → PF (1, 2)
  - 3 → PF (1, 2, 3)
  - 4 → PF (2, 3, 4) — 1 is replaced
  - 1 → PF (3, 4, 1) — 2 is replaced

    - 2 → PF (4, 1, 2) — 3 is replaced
    - 5 → PF (1, 2, 5) — 4 is replaced
    - 1 → hit (1, 2, 5)
    - 2 → hit (1, 2, 5)
    - 3 → PF (2, 5, 3) — 1 is replaced
    - 4 → PF (5, 3, 4) — 2 is replaced
    - 5 → hit (5, 3, 4)
    - **Total page faults = 9**
- With **4 frames**, FIFO → 10 page faults.
    - 1 → PF (1)
    - 2 → PF (1, 2)
    - 3 → PF (1, 2, 3)
    - 4 → PF (1, 2, 3, 4)
    - 1 → hit (1, 2, 3, 4)
    - 2 → hit (1, 2, 3, 4)
    - 5 → PF (2, 3, 4, 5) — 1 is replaced
    - 1 → PF (3, 4, 5, 1) — 2 is replaced
    - 2 → PF (4, 5, 1, 2) — 3 is replaced
    - 3 → PF (5, 1, 2, 3) — 4 is replaced
    - 4 → PF (1, 2, 3, 4) — 5 is replaced
    - 5 → PF (2, 3, 4, 5) — 1 is replaced
    - **Total page faults = 10**

7. **What is Buffer?**

A buffer is a memory area that stores data being transferred between two devices or between a device and an application.

8. **What is virtual memory?**

Virtual memory is a memory management technique that gives each process the illusion of having a large, continuous address space starting from address zero. In reality, this address space may exceed the size of physical RAM. The operating system achieves this by dividing virtual memory into pages and mapping them to

physical memory frames. If a required page is not in RAM, it is fetched from disk (swap space). This allows:

- Execution of programs larger than available RAM.
- Isolation between processes for security and stability.
- Simplified programming, since processes don’t need to worry whether data is in RAM or on disk.

9. **Explain the main purpose of an operating system?**

- An operating system is a software that manages computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.
- An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs conveniently and efficiently.

10. **What is demand paging?**

The process of loading the page into memory on demand (whenever a page fault occurs) is known as demand paging.

11. What is a kernel?

A kernel is the central component of an operating system that manages the operations of computers and hardware. It basically manages operations of memory and CPU time. It is a core component of an operating system. Kernel acts as a bridge between applications and data processing performed at the hardware level using inter-process communication and system calls.

12. **What are the different scheduling algorithms?**

- **First-Come, First-Served (FCFS) Scheduling**
    - Processes are executed in the order they arrive (like a queue).

    - Simple and fair, but can cause the *convoy effect* (long process delays shorter ones).
- **Shortest-Job-Next (SJN) / Shortest Job First (SJF)**
    - The process with the **smallest burst time** is executed first.
    - Minimizes average waiting time but requires knowing burst time in advance (not always practical).
- **Priority Scheduling**
    - Each process is assigned a **priority**, and the highest priority process runs first.
    - Can lead to **starvation** of lower-priority processes unless **aging** is used.
- **Shortest Remaining Time First (SRTF)**
    - A **preemptive version of SJF**.
    - If a new process arrives with a shorter burst time than the current one, CPU switches to it.
    - Efficient but involves more context switching.
- **Round Robin (RR) Scheduling**
    - Each process gets a **fixed time slice (quantum)** in a cyclic order.
    - Good for **time-sharing systems** and ensures fairness.
    - Performance depends on choosing the right quantum size.
- **Multilevel Queue Scheduling**
    - Processes are divided into different **queues** (e.g., system, interactive, batch).
    - Each queue may use a different scheduling algorithm.
    - Provides flexibility but adds complexity.

13. **Describe the objective of multi-programming?**

Multi-programming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The main objective of multi-programming is to keep multiple jobs in the main memory. If one job gets occupied with IO, the CPU can be assigned to other jobs.

14. **What is the time-sharing system?**

Time-sharing is a logical extension of multiprogramming. The CPU performs many tasks by switches that are so frequent that the user can interact with each program while it is running. A time-shared operating system allows multiple users to share computers simultaneously.

15. **What problem we face in computer system without OS?**

- Poor resource management
- Lack of User Interface
- No File System
- No Networking
- Error handling

16. **Briefly explain FCFS?**

FCFS stands for First Come First served. In the FCFS scheduling algorithm, the job that arrived first in the ready queue is allocated to the CPU and then the job that came second and so on. FCFS is a non-preemptive scheduling algorithm as a process holds the CPU until it either terminates or performs I/O. Thus, if a longer job has been assigned to the CPU then many shorter jobs after it will have to wait.

17. What is the RR scheduling algorithm?

A round-robin scheduling algorithm is used to schedule the process fairly for each job in a time slot or quantum and interrupting the job if it is not completed by then the job comes after the other job which is arrived in the quantum time makes these scheduling fairly.

- Round-robin is cyclic in nature, so starvation doesn’t occur
- Round-robin is a variant of first-come, first-served scheduling
- No priority or special importance is given to any process or task
- RR scheduling is also known as Time slicing scheduling

19. **Enumerate the different RAID levels?**

A redundant array of independent disks is a set of several physical disk drives that the operating system sees as a single logical unit. It plays a significant role in narrowing the gap between increasingly fast processors and slow disk drives. RAID has different levels:

- Level-0
- Level-1
- Level-2
- Level-3
- Level-4
- Level-5
- Level-6

20. **What is Banker's algorithm?**

The **Banker's Algorithm** is a **deadlock avoidance algorithm** used in operating systems. It works by simulating resource allocation before actually granting it. The idea is:

- Each process declares the **maximum number of resources** it might need.
- When a process requests resources, the algorithm checks if granting that request will still leave the system in a **safe state** (i.e., there exists at least one sequence in which all processes can finish).
- If the system would remain safe, the resources are allocated; otherwise, the process must wait.
  This is called the "banker's algorithm" because it's like how a banker only gives loans if they are sure the bank will not run out of money for future commitments.
  I'll quickly explain the logic in pseudo-code so you know my thought process:

```
Input: Allocation[n][m], Max[n][m], Available[m]

1) Need[i][j] = Max[i][j] - Allocation[i][j]   // for all i, j

2) Work = Available
   Finish[i] = false  // for all i
   SafeSeq = []
```

```
3) loop:
     found = false
     for i in 0..n-1:
        if Finish[i] == false AND (for all j: Need[i][j] <=
Work[j]):
            // pretend Pi can finish; when it does, it releases
what it holds
            for j in 0..m-1:
                Work[j] = Work[j] + Allocation[i][j]
            Finish[i] = true
            SafeSeq.push_back(i)
            found = true
     if found == false:
        break   // no progress this pass → stop

4) if (for all i: Finish[i] == true):
        print "SAFE", SafeSeq
   else:
        print "UNSAFE"
```

- `Allocation[i][j]` → This tells how many instances of resource `j` are currently allocated to process `i`.
- `Max[i][j]` → This is the maximum demand of process `i` for resource `j`. It's like the upper limit of how many resources that process may request.
- `Need[i][j] = Max[i][j] - Allocation[i][j]` → This tells the remaining resources process `i` still needs to complete its execution.
- `Available[j]` → Number of available instances of resource `j` in the system right now.
- `Work[j]` → A temporary copy of Available, which keeps track of resources as we simulate allocation during the safety check.
- `Finish[i]` → A boolean that marks whether process `i` can finish with the resources provided. Initially false for all, becomes true once we see it can finish.
- `Safe Sequence` → The order of processes that can finish without leading to deadlock. If we can find such a sequence covering all processes, the system is in a **safe state**.

21. **State the main difference between logical and physical address space?**

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	Logical address is generated by the CPU.	It is located in a memory unit.
Address Space	Logical Address Space is a set of all logical addresses generated by the CPU in reference to a program.	Physical Address is a set of all physical addresses mapped to the corresponding logical addresses.
Visibility	Users can view the logical address of a program.	Users can never view the physical address of the program.
Generation	Generated by the CPU.	Computed by MMU.
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical addresses but not directly.

22. What are overlays?

The concept of overlays is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlay concept says that whatever part you required, you load it and once the part is done, then you just unload it, which means just pull it back and get the new part you required and run it. Formally, “The process of transferring a block of program code or other data into internal memory, replacing what is already stored”.

23. What is fragmentation?

Processes are stored and removed from memory, which makes free memory space, which is too little to even consider utilizing by different processes.
Suppose, that process is not ready to dispense to memory blocks since its little size and memory hinder consistently staying unused is called fragmentation. This kind of issue occurs during a dynamic memory allotment framework when free blocks are small, so it can't satisfy any request.

24. What is the basic function of paging?

Paging is a memory management method which is used for non-contiguous memory allocation. It is a fixed-size partitioning scheme. In paging, both main memory and secondary memory are divided into equal fixed-size partitions. The partitions of the secondary memory area unit and the main memory area unit are known as pages and frames respectively.

25. How does swapping result in better memory management?

Swapping is a simple memory/process management technique used by the OS to increase the utilization of the processor by moving some blocked processes from the main memory to the secondary memory thus forming a queue of the temporarily suspended processes and the execution continues with the newly arrived process. During regular intervals that are set by the operating system, processes can be copied from the main memory to a backing store and then copied back later. Swapping allows more processes to be run that can fit into memory at one time

26. Write a name of classic synchronization problems?

- Bounded-buffer
- Readers-writers
- Dining philosophers

27. **What is the Direct Access Method?**

Direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations. The process is managed by a chip known as a DMA controller. The Direct Access method is based on a disk model of a file, such that it is viewed as a numbered sequence of blocks or records. It allows arbitrary blocks to be read or written. Direct access is advantageous when accessing large amounts of information.

28. What is the best page size when designing an operating system?

The best paging size varies from system to system, so there is no single best when it comes to page size. There are different factors to consider in order to come up with a suitable page size, such as page table, paging time, and its effect on the overall efficiency of the operating system.

29. What is multitasking?

Multitasking is a logical extension of a multiprogramming system that supports multiple programs to run concurrently. In multitasking, more than one task is executed at the same time. In this technique, the multiple tasks, also known as processes, share common processing resources such as a CPU.

30. What is caching?

The cache is a smaller and faster memory that stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data. Cache memory is used to reduce the average time to access data from the Main memory.

31. What is spooling?

Spooling refers to putting jobs in a buffer, a special area in memory, or on a disk where a device can access them when it is ready. Spooling is useful because devices access data at different rates.

32. What is the functionality of an Assembler?

The Assembler is used to translate the program written in Assembly language into machine code. The source program is an input of an assembler that contains assembly language instructions. The output generated by the assembler is the object code or machine code understandable by the computer.

33. What are interrupts?

The interrupts are a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. In I/O devices one of the bus control lines is dedicated to this purpose and is called the Interrupt Service Routine (ISR).

34. What is GUI?

GUI is short for Graphical User Interface. It provides users with an interface wherein actions can be performed by interacting with icons and graphical symbols.

35. What is preemptive multitasking?

Preemptive multitasking is a type of multitasking that allows computer programs to share operating systems (OS) and underlying hardware resources. It divides the overall operating and computing time between processes, and the switching of resources between different processes occurs through predefined criteria.

36. What is a pipe and when is it used?

A Pipe is a technique used for inter-process communication. It is a mechanism by which the output of one process is directed into the input of another process. Thus it provides a one-way flow of data between two related processes.

37. What are the advantages of semaphores?

- They are machine-independent.
- Easy to implement.
- Correctness is easy to determine.
- Can have many different critical sections with different semaphores.
- Semaphores acquire many resources simultaneously.
- No waste of resources due to busy waiting.

38. **What is a bootstrap program in the OS?**

Bootstrapping is the process of loading a set of instructions when a computer is first turned on or booted. During the startup process, diagnostic tests are performed, such as the power-on self-test (POST), which set or checks configurations for devices and implements routine testing for the connection of peripherals, hardware, and external memory devices. The bootloader or bootstrap program is then loaded to initialize the OS.

39. What is IPC?

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them.

40. What are the different IPC mechanisms?

These are the methods in IPC:

- **Pipes (Same Process)**: This allows a flow of data in one direction only. Analogous to simplex systems (Keyboard). Data from the output is usually buffered until the input process receives it which must have a common origin.
- **Message Queuing**: This allows messages to be passed between processes using either a single queue or several message queues. This is managed by the system kernel these messages are coordinated using an API.
- **Semaphores**: This is used in solving problems associated with synchronization and avoiding race conditions. These are integer values that are greater than or equal to 0.
- **Shared Memory**: This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.
- **Sockets**: This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection which is computer and OS independent

41. **What is the difference between preemptive and non-preemptive scheduling?**

- In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting for the state.
- The executing process in preemptive scheduling is interrupted in the middle of execution when a higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.
- In Preemptive Scheduling, there is the overhead of switching the process from the ready state to the running state, vice-verse, and maintaining the ready queue. Whereas the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
- In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. On the other hand, in non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with a small burst time may have to starve.

- Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive in the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.

42. **What is the zombie process?**

A process that has finished the execution but still has an entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

43. What are orphan processes?

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

44. What are starvation and aging in OS?

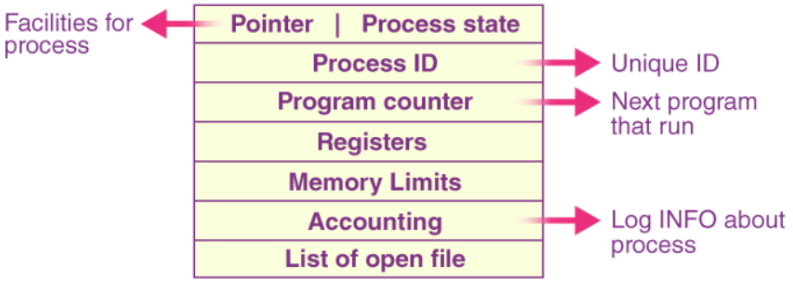
- Starvation: Starvation is a resource management problem where a process does not get the resources it needs for a long time because the resources are being allocated to other processes.
- Aging: Aging is a technique to avoid starvation in a scheduling system. It works by adding an aging factor to the priority of each request. The aging factor must increase the priority of the request as time passes and must ensure that a request will eventually be the highest priority request

45. **Write about monolithic kernel?**

A Monolithic Kernel is another classification of Kernel. It increases the size of the kernel, thus increasing the size of an operating system as well. This kernel provides CPU scheduling, memory management, file management, and other operating system functions through system calls. As both services are implemented under the same address space, this makes operating system execution faster.

46. What is Context Switching?

Switching of CPU to another process means saving the state of the old process and loading the saved state for the new process. In Context Switching the process is stored in the Process Control Block to serve the new process so that the old process can be resumed from the same part it was left.



PCB Diagram

47. What is the difference between the Operating system and kernel?

- The **kernel** is the **core part** of the operating system. It's responsible for low-level tasks like process management, memory management, device management, and system calls. Basically, it's the layer that directly interacts with the hardware.
- On the other hand, the **operating system (OS)** is a much broader concept. It includes the kernel **plus** other system programs, utilities, libraries, and user interfaces that make the system usable.

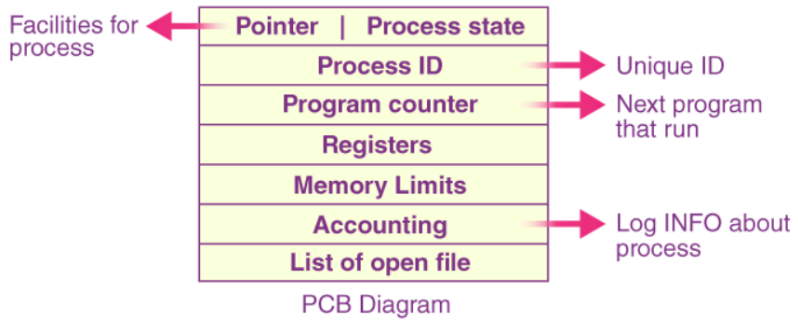
48. **What is the difference between process and thread?**

- A **process** is an independent program in execution, whereas a **thread** is a smaller unit of execution that lives inside a process.
- A process is isolated, meaning it has its own memory space, its own Process Control Block, stack, and address space. Because of that, communication between processes is slower and requires mechanisms like pipes or shared memory.
- On the other hand, threads within the same process share the same memory and resources. This makes them lightweight and communication between them much faster. Each thread only has its own stack and a Thread Control Block, but shares the parent process's address space.

- In terms of efficiency, process switching is heavier since it requires operating system involvement, whereas thread switching is faster and doesn't always require a kernel-level context switch.
- If one process is blocked then it will not affect the execution of other process. The second, thread in the same task could not run, while one server thread is blocked. **Why?**
    - If one **process** is blocked — say it's waiting for I/O — it doesn't affect the execution of other processes, because each process is independent and has its own memory space and CPU scheduling. The OS can simply schedule another process to run.
    - But with **threads**, it's different. Threads share the same address space and are part of the same process. If one thread inside a process is blocked while holding a lock or waiting for a shared resource, then other threads of the same process may also be blocked.
    - For example, imagine a multi-threaded web server: if one thread is handling a client request and gets blocked waiting for a database response while holding a mutex, other threads that need that mutex cannot proceed until it's released.
    - So the difference comes down to **independence vs. sharing**:
      - Processes are isolated → blocking doesn't affect others.
      - Threads are interdependent → blocking one may stall others in the same process.

    49. **What is PCB?**

The process control block (PCB) is a block that is used to track the process's execution status. A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCBs, that means logically contains a PCB for all of the current processes in the system.



PCB Diagram

50. When is a system in a safe state?

The set of dispatchable processes is in a safe state if there exists at least one order in which all processes can be run to completion without resulting in a deadlock.

51. What are a Trap and Trapdoor?

- A trap is a **software-generated interrupt** that occurs when a program encounters an exceptional condition. It is handled by the operating system. Unlike hardware interrupts, traps are triggered by the program itself (for example, divide-by-zero, invalid memory access, or a system call).
    - **Example:** If a program tries to divide 10 by 0, the CPU generates a trap and transfers control to the OS to handle the error.
- A trapdoor (also called a backdoor) is a **hidden or undocumented entry point** into a program or system, usually bypassing normal authentication or security checks. While it can be used by developers for debugging or maintenance, it is often considered a security vulnerability if left open.
    - **Example:** A developer might insert a secret username/password in software for testing. An attacker could exploit this trapdoor to gain unauthorized access.

52. **Write a difference between program and process?**

Program	Process
Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
Program is a passive entity as it resides in the secondary memory.	Process is an active entity as it is created during execution and loaded into the main memory.
The program exists in a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of a task.
A program is a static entity.	The process is a dynamic entity.
Program does not have any resource requirement, it only requires memory	Process has a high resource requirement, it needs resources like

Program	Process
space for storing the instructions.	CPU, memory address, and I/O during its lifetime.
The program does not have any control block.	The process has its own control block called Process Control Block.

53. **What is a dispatcher?**

The dispatcher is the module that gives process, control over the CPU after it has been selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

54. **Define the term dispatch latency?**

Dispatch latency can be described as the amount of time it takes for a system to respond to a request for a process to begin operation.

55. **What are the goals of CPU scheduling?**

- Max CPU utilization: Keeping CPU as busy as possible and Fair allocation of CPU.
- Max throughput: Number of processes that complete their execution per time unit
- Min turnaround time: Time taken by a process to finish execution
- Min waiting time: Time a process waits in ready queue
- Min response time: Time when a process produces the first response

56. **What is a critical- section?**

A critical section is a portion of code where a process accesses **shared resources** (like variables, files, or I/O devices). Since multiple processes or threads may try to access these resources at the same time, the critical section must be executed **atomically** to avoid race conditions and ensure data consistency. In simple terms, it's the part of the program that **must not be executed by more than one process/thread at the same time**.

- **Example:** Updating a shared bank account balance — if two threads update it simultaneously without synchronization, the result may be incorrect. Hence, that update code is placed in the critical section.

57. **Tell me about different synchronization techniques?**

There are several synchronization techniques used in operating systems and concurrent programming to avoid race conditions and ensure consistency:

- **Mutex (Mutual Exclusion Object):**
  - A mutex allows only **one thread** at a time to access a shared resource.
  - If one thread has locked the mutex, others must wait until it's unlocked.
  - **Example:** Two threads trying to update a shared bank account balance — a mutex ensures only one updates at a time.
- **Semaphore:**
  - A semaphore is a signaling mechanism that can allow **multiple threads** (up to a limit) to access a resource.
  - It uses a counter: if the counter > 0, a thread can proceed and decrement it; if 0, the thread must wait.
  - **Example:** A database connection pool with 10 connections — a semaphore initialized to 10 ensures that at most 10 threads access the pool simultaneously.
- **Condition Variables:**
  - Used with a mutex to allow threads to **wait for certain conditions** to become true.
  - A thread waits on a condition variable and gets notified (signal/broadcast) when another thread changes the state.
  - **Example:** In a producer-consumer problem, the consumer thread waits until the producer signals that the buffer is not empty.
- **File Locks:**

- Special locks placed on files to control access by multiple processes.
- Can be **shared locks** (multiple readers allowed) or **exclusive locks** (only one writer allowed).
- **Example:** When one process is writing to a log file, file locking prevents other processes from writing at the same time, avoiding corruption.

58. Write a difference between a user-level thread and a kernel-level thread?

- **Implementation:**
  - User-level threads are created and managed in user space by thread libraries.
  - Kernel-level threads are managed directly by the operating system.
- **Visibility to OS:**
  - The OS is not aware of user-level threads, it only sees the process.
  - Kernel-level threads are fully recognized and scheduled by the OS.
- **Complexity:**
  - User-level threads are simpler and easier to implement.
  - Kernel-level threads are more complex to implement and manage.
- **Context Switching:**
  - User-level thread switching is fast and does not require kernel intervention.
  - Kernel-level thread switching is slower since it involves the OS.
- **Hardware Support:**
  - User-level threads do not need hardware support.
  - Kernel-level threads require hardware and OS support.
- **Blocking:**
  - If one user-level thread makes a blocking system call, the entire process gets blocked.
  - If one kernel-level thread blocks, other threads of the same process can still run.
- **Independence:**
  - User-level threads are dependent on the process they belong to.
  - Kernel-level threads are independent and scheduled individually by the OS.

59. **Difference between Multithreading and Multitasking?**

- **Definition:**
  - **Multithreading:** Multiple threads (lightweight units) of the same process execute concurrently.
  - **Multitasking:** Multiple independent processes/programs run concurrently on the system.
- **Execution Unit:**
  - **Multithreading:** CPU switches between threads of a single process.
  - **Multitasking:** CPU switches between completely different processes.
- **Weight:**
  - **Multithreading:** Threads are lightweight since they share the same address space.
  - **Multitasking:** Processes are heavyweight because each has its own memory and resources.
- **Feature of:**
  - **Multithreading:** It is a feature of a process.
  - **Multitasking:** It is a feature provided by the operating system.
- **Resource Sharing:**
  - **Multithreading:** Threads share the same memory and resources within the process.
  - **Multitasking:** Processes have separate memory and resources, managed by the OS.

60. **What are the drawbacks of semaphores?**

- **Priority Inversion:** A high-priority process may be forced to wait if a lower-priority process holds the semaphore.
- **Not Enforced by System:** Correct usage of semaphores depends entirely on the programmer's discipline; the OS doesn't enforce it.
- **Manual Tracking:** The programmer must carefully manage every `wait` (P) and `signal` (V) operation. A mismatch can lead to errors.
- **Deadlocks:** If used incorrectly, semaphores can cause processes to block indefinitely, resulting in deadlock.

61. **What is Peterson's approach?**

Peterson's algorithm is a way to let **two processes** share a resource (critical section) without stepping on each other.
It works by using two shared variables:

- `flag[2]`:
  - `flag[i] = true` → process *i* wants to enter the critical section.
  - `flag[i] = false` → process *i* is not interested.
- `turn`:
  - Decides whose "turn" it is if both want to enter at the same time.

```
// Shared
bool flag[2] = {false, false};
int turn;

// For process i
void process(int i) {
    int j = 1 - i;

    // Entry Section
    flag[i] = true;   // I want to enter
    turn = j;         // Give chance to the other
    while(flag[j] && turn == j) {
        // Wait if other also wants to enter AND it's their turn
    }

    // Critical Section
    cout << "Process " << i << " is inside\n";

    // Exit Section
    flag[i] = false;  // I'm done
}
```

How it works: Suppose we have **Process 0** and **Process 1**.

- **Process 0 wants to enter**:
  - It sets `flag[0] = true`.
  - It gives priority to the other process: `turn = 1`.
  - Now it checks: *"Is process 1 also interested (`flag[1] == true`) AND is it process 1's turn (`turn == 1`)?"*
    - If **yes**, process 0 will wait.
    - If **no**, process 0 can safely enter the critical section.

- **Process 1 wants to enter**:
  - Similarly, it sets `flag[1] = true`.
  - It sets `turn = 0`.
  - Then it checks whether process 0 also wants to enter and has priority.
- **Case when both want to enter at the same time:**
  - Let's say process 0 sets `turn = 1` and process 1 sets `turn = 0`.
  - This means **only one process wins**, because the `while(flag[j] && turn == j)` condition forces the other to wait.
  - So there's **no deadlock** and **no two processes enter together**.
- **After leaving critical section:**
  - Each process sets its `flag[i] = false`, so the other can now enter.

62. **Define the term Bounded waiting?**

In operating systems, bounded waiting is a condition that guarantees a process will not be indefinitely blocked from entering its critical section while other processes are allowed to enter repeatedly. It ensures that every process gets a fair chance to access shared resources and prevents starvation.

63. **What are the solutions to the critical section problem?**

The **critical section problem** arises when multiple processes or threads try to access a shared resource simultaneously. To ensure **mutual exclusion**, **progress**, and **bounded waiting**, we need proper solutions. Broadly, there are three categories:

- **Software Solutions**
  - These are purely algorithmic approaches that use shared variables and program logic to control access.
  - Example: **Peterson's Algorithm** (for two processes).
  - They don't require special hardware but are mostly educational, since they can be complex and not always practical on modern architectures.
- **Hardware Solutions**
  - Here, special hardware instructions are used to achieve mutual exclusion.

    - Example: **Test-and-Set**, **Compare-and-Swap**, or disabling interrupts temporarily.
    - These are faster and simpler but can lead to busy-waiting (spinning) and sometimes fairness issues.
- **Semaphores (Synchronization Tools)**
    - Semaphores are high-level constructs provided by the OS.
    - They use `wait()` and `signal()` operations to control process access.
    - Unlike hardware busy-waiting, semaphores can block processes until the resource is free, which is more efficient.

64. **What is concurrency?**

A state in which a process exists simultaneously with another process than those it is said to be concurrent.

65. **Write a drawback of concurrency?**

- It is required to protect multiple applications from one another.
- It is required to coordinate multiple applications through additional mechanisms.
- Additional performance overheads and complexities in operating systems are required for switching among applications.
- Sometimes running too many applications concurrently leads to severely degraded performance.

66. **What are the necessary conditions which can lead to a deadlock in a system?**

- **Mutual Exclusion**: There is a resource that cannot be shared.
- **Hold and Wait**: A process is holding at least one resource and waiting for another resource, which is with some other process.
- **No Preemption**: The operating system is not allowed to take a resource back from a process until the process gives it back.
- **Circular Wait:** A set of processes waiting for each other in circular form.

67. What are the issues related to concurrency?

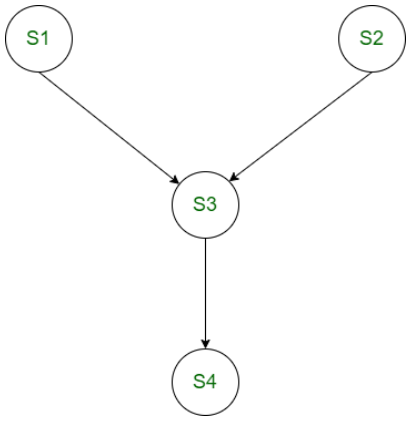
When multiple processes or threads run concurrently, several issues can arise:

- **Non-atomic operations** → If an operation is not atomic, it can be interrupted midway by another process. This may leave shared data in an inconsistent state.
- **Race conditions** → When the final outcome depends on the order of execution between processes, leading to unpredictable results.
- **Blocking** → A process might be forced to wait for a long time if it needs a resource or input. This can reduce responsiveness, especially if that process was supposed to update critical data.
- **Starvation** → Some processes may be continuously overlooked by the scheduler and never get CPU or resources, preventing them from making progress.
- **Deadlock** → Two or more processes are stuck waiting for each other's resources, meaning none of them can proceed.

68. Why do we use precedence graphs?

A precedence graph is a directed acyclic graph that is used to show the execution level of several processes in the operating system. It has the following properties also:

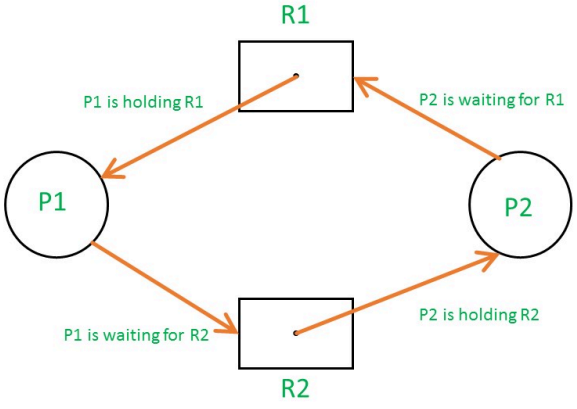
- Nodes of graphs correspond to individual statements of program code.
- An edge between two nodes represents the execution order.
- A directed edge from node A to node B shows that statement A executes first and then Statement B executes



69. Explain the resource allocation graph?

A **Resource Allocation Graph (RAG)** is a graphical way to represent how processes and resources interact in a system. It's mainly used to understand and detect deadlocks.

- It's a **directed graph** with two types of nodes:
  - **Processes (P1, P2, …)** represented as circles.
  - **Resources (R1, R2, …)** represented as rectangles, with small dots inside them showing individual instances.
- There are two types of edges:
  1. **Request Edge (P → R):** A directed edge from a process to a resource means the process has requested that resource but has not been allocated it yet.
  2. **Assignment Edge (R → P):** A directed edge from a resource instance to a process means that instance has been allocated to that process.
- **How it helps:**
  - If the graph has **no cycle**, the system is safe (no deadlock).
  - If there **is a cycle**:
  - With **one instance per resource type**, the cycle definitely means a deadlock.
  - With **multiple instances**, a cycle may or may not lead to deadlock.



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

70. What is a deadlock?

Deadlock is a situation when two or more processes wait for each other to finish and none of them ever finish. Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s).

71. **What is the goal and functionality of memory management?**

The goal and functionality of memory management are as follows;

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

72. **Write a difference between physical address and logical address?**

- **Definition**
  - **Logical Address** → The virtual address generated by the CPU during program execution.
  - **Physical Address** → The actual location in the main memory (RAM).
- **Address Space**
  - **Logical Address Space** → The set of all logical addresses a program can generate.
  - **Physical Address Space** → The set of all physical addresses corresponding to logical addresses.
- **Visibility**
  - **Logical Address** → Visible to the user/programmer.
  - **Physical Address** → Not visible to the user, only the OS/hardware knows it.
- **Access**
  - **Logical Address** → Used by the program to access memory.

    - **Physical Address** → Accessed indirectly through mapping done by the **MMU (Memory Management Unit)**.
- **Generation**
    - **Logical Address** → Generated by the CPU.
    - **Physical Address** → Computed by the MMU when mapping logical to physical.

73. **Explain address binding? Write different types of address binding?**

Address binding is the process of mapping logical addresses to physical addresses, and it can happen at three different stages:

- **Compile-time Binding** → If the memory location is known at compile time, absolute addresses are generated directly. But if the program is moved later, it has to be recompiled.
- **Load-time Binding** → If the program's location is not known at compile time, the compiler generates **relocatable code**. The final binding to physical addresses happens when the program is loaded into memory.
- **Execution-time Binding** → If the process can be moved during execution (like in paging/segmentation), the binding is done dynamically by the **MMU** at runtime.

74. **Write an advantage of dynamic allocation algorithms?**

- When we do not know how much amount of memory would be needed for the program beforehand.
- When we want data structures without any upper limit of memory space.
- When you want to use your memory space more efficiently.
- Dynamically created lists insertions and deletions can be done very easily just by the manipulation of addresses whereas in the case of statically allocated memory insertions and deletions lead to more movements and wastage of memory.
- When you want to use the concept of structures and linked lists in programming, dynamic memory allocation is a must

75. **Write a difference between internal fragmentation and external fragmentation?**

- **Definition**
  - **Internal Fragmentation** → Wastage of memory *inside* a fixed-sized block because the allocated memory is larger than what the process actually needs.
  - **External Fragmentation** → Wastage of memory due to *scattered free spaces* between allocated blocks that are too small to be used.
- **Cause**
  - **Internal** → Occurs when memory is divided into **fixed-sized partitions**.
  - **External** → Occurs when memory is divided into **variable-sized partitions** allocated dynamically.
- **Example**
  - **Internal** → If a process needs 18KB and is given a 20KB block, 2KB is wasted internally.
  - **External** → If free memory exists as small non-contiguous holes (e.g., 10KB + 5KB + 7KB scattered), a 15KB process cannot be allocated.
- **Solution**
  - **Internal** → Use best-fit or dynamic partitioning to reduce unused space.
  - **External** → Use **compaction, paging, or segmentation** to make memory contiguous.

76. **Define the Compaction?**

Compaction is a memory management technique used to solve the problem of **external fragmentation**. It works by **shifting processes and data in memory** so that all the free memory gets collected into one large **contiguous block**, instead of being scattered in small pieces. For example:

- If free memory is scattered as 5KB, 10KB, and 8KB in different locations, a 20KB process cannot be allocated. After compaction, these small holes are merged into a single 23KB free block, making allocation possible.

77. **Write about the advantages and disadvantages of a hashed-page table?**

Advantages

- The main advantage is synchronization.
- In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

  Disadvantages
- Hash collisions are practically unavoidable. when hashing a random subset of a large set of possible keys.
- Hash tables become quite inefficient when there are many collisions.
- Hash table does not allow null values, like a hash map.

78. **Write a difference between paging and segmentation?**

Paging	Segmentation
In paging, program is divided into fixed or mounted-size pages.	In segmentation, the program is divided into variable-size sections.
For the paging operating system is accountable.	For segmentation compiler is accountable.
Page size is determined by hardware.	Here, the section size is given by the user.
It is faster in comparison of segmentation.	Segmentation is slow.
Paging could result in internal fragmentation.	Segmentation could result in external fragmentation.
In paging, logical address is split into that page number and page offset.	Here, logical address is split into section number and section offset.
Paging comprises a page table which encloses the base address of every page.	While segmentation also comprises the segment table which encloses the segment number and segment offset.
A page table is employed to keep up the page data.	Section Table maintains the section data.

Paging	Segmentation
In paging, operating system must maintain a free frame list.	In segmentation, the operating system maintains a list of holes in the main memory.
Paging is invisible to the user.	Segmentation is visible to the user.
In paging, processor needs page number, offset to calculate the absolute address.	In segmentation, the processor uses segment number, and offset to calculate the full address.

79. **Write a definition of Associative Memory and Cache Memory?**

- **Definition**
  - *Associative Memory*: A memory unit where data is accessed by **content** (also called content-addressable memory).
  - *Cache Memory*: A **fast, small memory** placed between CPU and main memory to speed up access.
- **Access Method**
  - *Associative Memory*: Data is searched and accessed **using its content**.
  - *Cache Memory*: Data is accessed **using its address**.
- **Purpose**
  - *Associative Memory*: Reduces **search time** when looking for a matching item.
  - *Cache Memory*: Reduces the **average memory access time** by storing frequently used data.
- **Usage**
  - *Associative Memory*: Used where **quick lookups** are needed, like in translation lookaside buffers (TLB).
  - *Cache Memory*: Used when **data is reused repeatedly**, e.g., CPU instructions and operands.
- **Example:**
  - **Associative Memory Example**
    - Think of a **phone contact search**.
      - You don’t remember the phone number, but you type the **name**, and the phone finds the number by matching the content.

        - That’s exactly what **associative memory (content-addressable memory)** does — you give the content, and it finds the data.
    - **In Computer Systems**: Associative memory is used in the **TLB (Translation Lookaside Buffer)** to quickly find the frame number for a given page number.
- **Cache Memory Example**
    - Imagine you are studying and you keep a **notepad** for quick formulas.
        - Instead of opening the textbook every time (main memory), you just check your notepad (cache).
        - The notepad stores the most **frequently used** data for quick access.
    - **In Computer Systems**: Cache stores frequently used instructions and data close to the CPU to avoid fetching them again from slower main memory.

80. **What is "Locality of reference"?**

Locality of reference is the tendency of a program to access a small portion of memory repeatedly over a short time, rather than randomly accessing the whole memory.

81. **Write down the advantages of virtual memory?**

- A higher degree of multiprogramming.
- Allocating memory is easy and cheap
- Eliminates external fragmentation
- Data (page frames) can be scattered all over the PM
- Pages are mapped appropriately anyway
- Large programs can be written, as the virtual space available is huge compared to physical memory.
- Less I/O required leads to faster and easy swapping of processes.
- More physical memory is available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

- More efficient swapping

82. **Write down the basic concept of the file system?**

A file is a collection of related information that is recorded on secondary storage.

83. **Write the names of different operations on file?**

Operation on file: Create | Open | Read | Write | Rename | Delete | Append | Truncate | Close

84. **Define the term Bit-Vector?**

A Bitmap or Bit Vector is a series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: 0 indicates that the block is allocated and 1 indicates a free block.

85. **What is a File allocation table?**

A file allocation table (FAT) is a table that an operating system maintains on a hard disk that provides a map of the cluster (the basic units of logical storage on a hard disk) that a file has been stored in.

86. **What is rotational latency?**

Rotational latency is the delay caused while the disk platter rotates and brings the **desired sector** under the read/write head.

87. **What is seek time?**

Seek time is the time taken for the **disk arm (read/write head)** to move to the correct **track** on the disk where the required data is stored.

88. **What is Belady's Anomaly?**

Belady's Anomaly is a phenomenon in which **increasing the number of page frames** in memory **leads to more page faults** instead of reducing them. It occurs in some page replacement algorithms, most notably **FIFO (First-In-First-Out)**. Ideally, adding more frames should reduce page faults (since more pages can be stored), but under Belady's anomaly, the opposite happens.
For example: Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

- With **3 frames**, FIFO → 9 page faults.
  - 1 → PF (frames: 1)
  - 2 → PF (1, 2)
  - 3 → PF (1, 2, 3)
  - 4 → PF (2, 3, 4) — 1 is replaced
  - 1 → PF (3, 4, 1) — 2 is replaced
  - 2 → PF (4, 1, 2) — 3 is replaced
  - 5 → PF (1, 2, 5) — 4 is replaced
  - 1 → hit (1, 2, 5)
  - 2 → hit (1, 2, 5)
  - 3 → PF (2, 5, 3) — 1 is replaced
  - 4 → PF (5, 3, 4) — 2 is replaced
  - 5 → hit (5, 3, 4)
  - **Total page faults = 9**
- With **4 frames**, FIFO → 10 page faults.
  - 1 → PF (1)
  - 2 → PF (1, 2)
  - 3 → PF (1, 2, 3)
  - 4 → PF (1, 2, 3, 4)
  - 1 → hit (1, 2, 3, 4)
  - 2 → hit (1, 2, 3, 4)
  - 5 → PF (2, 3, 4, 5) — 1 is replaced
  - 1 → PF (3, 4, 5, 1) — 2 is replaced
  - 2 → PF (4, 5, 1, 2) — 3 is replaced
  - 3 → PF (5, 1, 2, 3) — 4 is replaced
  - 4 → PF (1, 2, 3, 4) — 5 is replaced

- 5 → PF (2, 3, 4, 5) — 1 is replaced
- **Total page faults = 10**

89. **What happens if a non-recursive mutex is locked more than once?**

If a thread that has already locked a **non-recursive mutex** tries to lock it again, it will **block itself** and enter the waiting list for that mutex. Since no other thread can unlock it (the thread itself is the owner and is waiting), this leads to a **deadlock**.

- A **non-recursive mutex** does not keep track of how many times the same thread has locked it.
- Only a **recursive mutex** allows the same thread to lock it multiple times (but requires the same number of unlocks).

90. **What are the advantages of a multiprocessor system?**

There are some main advantages of a multiprocessor system:

- Enhanced performance.
- Multiple applications.
- Multi-tasking inside an application.
- High throughput and responsiveness.
- Hardware sharing among CPUs.

91. **What are real-time systems?**

A real-time system means that the system is subjected to real-time, i.e., the response should be guaranteed within a specified timing constraint or the system should meet the specified deadline.

92. **How to recover from a deadlock?**

We can recover from a deadlock by following methods:

- Process termination

    - Abort all the deadlock processes
    - Abort one process at a time until the deadlock is eliminated
- Resource preemption
    - Rollback
    - Selecting a victim