📚 Operating Systems Fundamentals Overview

Brief Overview

This note covers **Operating System fundamentals** and was created from the Operating Systems Course for Beginners YouTube video. The material spans approximately 25 hours of instruction, with topics ranging from OS architecture and hardware interaction to process management, memory handling, and basic I/O; the course emphasizes fundamental theory, numerical examples, and periodic revision to reinforce learning. As you dive in, remember the course prerequisites include basic computer architecture knowledge and introductory programming experience.

Key Points

- The initial sections review key OS concepts, including OS types (batch, time-sharing, real-time), hardware components (CPU, memory, I/O), and the stored-program concept, including the critical role of main memory.
- Later sections delve into process management, including scheduling (preemptive vs. non-preemptive), the process lifecycle, and the hardware requirements for multiprogramming.
- Finally, the notes explore memory management in detail, covering logical vs physical addresses, paging, TLBs, page replacement algorithms, and the working set model.

📚 Course Overview

- **Total duration:** 25 hours
- **Materials:** 400 pages of lecture notes (available via QR-code link)
- **Core topics:**
  - Introduction & background
  - Process Management & CPU scheduling
  - Process synchronization/coordination
  - Deadlock handling
  - Memory Management
  - File Management

🛠️ Prerequisites

- Basic understanding of **computer organization and architecture** (hardware components, how a computer works)
- Introductory programming experience (e.g., **C**, **Java**, or any other language)

📖 Course Structure & Learning Phases

1. **Theory** – concepts explained with real-life examples
2. **Numericals** – problem-solving to test comprehension
3. **Revision** – periodic review to reinforce memory

> *"If you fail to revise, you fail to learn."*

🖥️ Operating System Fundamentals

❓ What Is an Operating System?

> **Operating System (OS):** An interface between the **user** and the **computer**, managing hardware resources and providing services to applications.

📋 OS Functions & Goals

- **Resource management** (CPU, memory, I/O)
- **Program execution** (loading, scheduling)
- **User interface** (CLI/GUI)
- **Protection & security**

🏗️ Types & Architecture

OS Type	Typical Use	Key Characteristic
Batch	Early mainframes	No interactive user
Time-sharing	Multi-user systems	CPU time sliced among users
Real-time	Embedded/critical systems	Deterministic response
Distributed	Networked computers	Resource sharing across nodes
Multimedia	Audio/Video processing	High I/O throughput

Von Neumann Architecture – the classic model comprising **Input**, **Output**, **Memory**, and **CPU** (Processor).

🔧 Hardware Components

📥 Input/Output Devices

- Examples: **monitor**, **keyboard**, **printer**, **hard disk** (also treated as I/O in von Neumann model)

🧠 Memory

Memory Type	Alternate Name	Volatility	Typical Speed	Examples
Primary	Main memory	**Volatile** (data lost on power-off)	Fast	RAM, ROM, cache, registers
Secondary	Auxiliary memory	**Non-volatile** (persists after power-off)	Slower	Hard disk, SSD, USB flash drives

> **Volatile:** Content disappears when power is removed.
> **Non-volatile:** Content remains after power loss.

🧱 CPU (Processor)

⚙️ Control Unit (CU)

- Generates **timing/control signals** (clock signals)
- Directs **sequential execution** of **micro-operations** (e.g., load, add, store)

➕ Arithmetic Logic Unit (ALU)

- Performs **arithmetic** (addition, subtraction) and **logical** (AND, OR, NOT) operations
- Consists of **adders**, **subtractors**, etc.

📂 Program Execution & Stored-Program Concept

1. **Source code** (e.g., test.c) → **Compiler** → **Executable file** (.exe)
2. **Operating System** loads the executable from **secondary memory** (hard disk) into **primary memory** (RAM).
3. **CPU** fetches instructions from **RAM** and executes them **sequentially** (instruction 1, then 2, ..., instruction n).

> **Stored-Program Concept:** Programs must reside in **main memory** before the CPU can execute them.

Why CPU Can't Execute Directly From Disk

- **Speed mismatch:** CPU operates at GHz speeds, while hard disks are orders of magnitude slower.
- **Latency:** Accessing disk for each instruction would stall the CPU, degrading performance.
- **Solution:** Load the entire program (or active portion) into fast **primary memory**, enabling rapid instruction fetch and execution.

Golden Rule

> *Every program that the CPU executes must first be **stored in main memory**.*

🌐 Operating System Overview

Definition 1

> *Operating system (OS) is an interface that converts high-level user language into machine language understandable by hardware, managing communication between user and hardware.*

Definition 2 – Resource Manager

> *OS acts as a manager of both hardware and software resources.*

- **Hardware resources** – CPU, monitor, input/output devices
- **Software resources** – files, semaphores, monitors (details later)

Definition 3 – Control Program

> *OS is a control program that allocates, deallocates and frees resources.*

Definition 4 – Utility Provider for Application Development

> *OS supplies utilities and an environment that let developers write programs in high-level languages while the OS handles low-level operations.*

🖥️ OS Architecture & Components

Kernel (Core) 🧩

- **Process manager**
- **Memory manager**
- **File manager**
- **Device manager**
- **Protection manager**

These modules together form the **kernel**, the core that directly interacts with hardware.

Levels of Interface

Level	Description	Example
Level 1	User-OS interface; command interpreter (shell) or GUI; mediates user commands to the kernel.	Shell, Windows GUI
Level 2	OS kernel; performs requested operations using hardware.	Kernel modules above

Hardware Components Managed by OS

- **CPU** – control unit + arithmetic logic unit (ALU)
- **Main memory**
- **I/O devices** – disks, keyboards, monitors, etc.

🔄 Representation of Languages

- **User language:** symbols **a–z** (high-level language)
- **Computer language:** symbols **0–1** (binary machine language)

🎯 Goals of an Operating System

- **Convenience** – primary goal for most general-purpose OS.
- **Efficiency** – optimal use of resources.
- **Reliability** – consistent correct operation.
- **Robustness** – ability to tolerate errors.
- **Scalability** – capacity to grow with workload.
- **Portability** – run on different hardware platforms.
  Example: Windows is more portable than macOS, which runs only on Apple devices.

Goal Priorities by OS Type

OS Type	Primary Goal(s)	Reason
General-purpose (e.g., Windows, Unix)	Convenience	Survey shows users prefer convenience.
Real-time (hard deadline)	Reliability & Efficiency	Critical systems (missile control, air-traffic) cannot tolerate delays.
Real-time (soft deadline)	Reliability	Deadline misses are tolerable but undesirable (e.g., ATM).

📋 Functions of an Operating System

1. **Processor Management** – schedule CPU time, detect errors, ensure security.
2. **Memory Management** – allocate/deallocate main memory.
3. **File Management** – organize, store, retrieve files.
4. **Device Management** – control I/O devices.
5. **Protection Management** – enforce access controls.

Detailed explanations will follow in later modules.

🏛️ OS as a Government Metaphor

- **Government = OS**
  - Ministries ↔ OS modules (process manager, file manager, etc.)
  - Provides services to citizens ↔ Provides services to applications/users.
- Not a single program; a collection of cooperating services.

📚 Types of Operating Systems

Category	Examples	Key Characteristics
Batch OS	Early mainframes	Run jobs without interactive input
Multiprogramming OS	Modern general-purpose OS	Hold several programs in memory simultaneously
Time-Sharing OS	Unix, Windows (multitasking)	Interactive use, rapid context switches
Real-Time OS	RT-Linux, VxWorks	Strict timing constraints (hard/soft)
Distributed OS	Google Borg, Hadoop	Manage resources across networked machines
Network OS	Windows Server, Linux with NFS	Provide services over a network
Multiprocessor OS	Solaris SMP, Windows Server	Run on systems with multiple CPUs

🧩 Uniprocessor vs. Multiprocessor (preview)

- **Uniprocessor:** single CPU; early systems.
- **Multiprocessor:** multiple CPUs; discussed later.

📂 Memory Layout During Boot

- **System area** – region where the OS kernel resides after boot.
- **User area** – region where user programs are loaded.

Uni-programming (single-program) Model

1. Multiple ready-to-run programs stored on disk.
2. During boot, OS loads **one** program into main memory.
3. CPU executes that program.
4. If the program performs I/O, the CPU may become idle → low utilization.

Drawbacks

- Low CPU utilization
- Reduced throughput
- Inefficient resource use

🔁 Multiprogramming (preview)

- Allows multiple programs to reside in memory concurrently.
- Improves CPU utilization by switching to another ready program when one waits for I/O.

Throughput & CPU Utilization 📈

> **Throughput** – the number of programs completed in a unit of time.

- When the CPU is **idle**, both **efficiency** and **throughput** drop.
- Goal of an operating system: **maximise CPU utilisation** to keep throughput high.

Uniprogramming Operating System (Single-Program) 🖥️

- Loads **only one** ready-to-run program from secondary storage (hard disk) to main memory.
- Example: **MS-DOS** (Microsoft Disk Operating System) – 1990s, command-based, no GUI.
- Limitation: if the sole program performs I/O, the CPU becomes idle → reduced utilisation.

Multiprogramming Operating System 🌐

> **Multiprogramming** – an OS that can load **multiple programs** from disk into main memory simultaneously.

- Holds several **ready-to-run** programs; while one waits for I/O, another uses the CPU.
- Gives the impression of **multiplexing**: the CPU switches among programs (P1 → P2 → P3 …).
- **Degree of multiprogramming** = number of programs resident in memory (e.g., 4 ⇒ degree = 4).

Schematic View

```
Secondary Storage (hard disk) → Jobs P1, P2, P3, P4
          ↓
Main Memory (multiple slots)
          ↓
CPU executes one job at a time
          ↖   ↙
   I/O devices   Ready/Waiting jobs
```

- Result: **CPU utilisation ↑**, **idle time ↓**.

Multitasking vs. Multiprogramming 🔄

- Unix terminology: **program**
- Windows terminology: **task**
  Both refer to the same concept of multiple active workloads.

Process vs. Program 📂

> A **process** is a **program in execution**.

Types of Multiprogramming

Preemptive Multiprogramming ⏱️

- The OS **forcefully deallocates** the CPU from a running job.
- Triggers:
  - Arrival of a **higher-priority** job.
  - **Time-slice** expiry to prevent long-running jobs from starving others.
- Example: Job 2 runs, Job 3 (higher priority) arrives → OS preempts Job 2 and schedules Job 3.

Feature	Preemptive
CPU allocation	Forced removal based on priority/time
Starvation handling	Reduced (higher-priority jobs get CPU)
Typical scheduling	Round-robin, priority-based

Non-preemptive Multiprogramming 🚫⏱️

- A job **releases** the CPU **voluntarily**.
- Release conditions:
  1. **Completion** of all instructions.
  2. **I/O request** – job moves to I/O devices.
  3. **System call** – explicit request to yield CPU (covered later).

Feature	Non-preemptive
CPU allocation	No forced removal; job runs to completion of a release condition
Starvation risk	Higher, because a long-running job can block others
Typical scheduling	First-Come-First-Served (FCFS)

Drawbacks of Multiprogramming ⚠️

- **Starvation** – low-priority jobs may wait indefinitely.
- **Lack of interactivity** – system may become unresponsive if a job monopolises the CPU.

Classification of Operating Systems 🗃️

Category	Examples / Characteristics
Basic OS	Batch, **Uniprogramming**, **Multiprogramming**, Time-sharing, Real-time
Advanced OS	Network, Distributed
Processor type	**Uniprocessor** – single CPU **Multiprocessor** – multiple CPUs (still one program per CPU at a time)

All concepts above stem from the discussion of throughput, CPU utilisation, and the evolution from single-program to multiprogramming operating systems.

📚 Multiprogramming & Preemption

> **Multiprogramming** – Running multiple programs in overlapping time periods by sharing the CPU.

- **Non-preemptive** multiprogramming
  - A running task keeps the CPU until it **terminates** or **requests I/O**.
  - Can cause **starvation** for lower-priority tasks (e.g., P2, P3, P4 waiting while P1 runs indefinitely).
- **Preemptive** multiprogramming (used in modern OSes: Windows 95-11, Unix, Linux, macOS)
  - The OS can **forcefully deallocate** the CPU from a running task.
  - Improves responsiveness and prevents starvation.

Feature	Non-Preemptive	Preemptive
CPU taken away	Only on termination or I/O request	On time slice expiry **or** higher-priority arrival
Starvation risk	High	Low
Typical OSes	Early systems	Modern systems (Unix, Linux, macOS, Windows)

⏱️ Time-Sharing & Priority Preemption

- **Time-sharing**: OS assigns a fixed time quantum (e.g., 10 s) to each process. When the quantum expires, the process is preempted even if not finished.
- **Priority-based preemption**: If a higher-priority process (e.g., P_y) becomes ready, the OS removes the current lower-priority process (e.g., P_x) from the CPU and schedules the higher-priority one.

> **Multitasking OS** – A preemptive multiprogramming system that supports both time-sharing and priority-based preemption.

🔄 Process Lifecycle & Queues

1. **Job Creation** – New program is introduced to the system.
2. **Ready Queue (RQ)** – The job moves here when it is ready to run (stored in memory).
3. **Scheduler** selects a process from RQ → **CPU**.
4. While on the CPU, a process can:
   - **Terminate** → leaves CPU permanently.
   - **Request I/O** → moves to an I/O wait queue (SE).
5. After I/O completes, the process returns to **Ready Queue**.
6. Cycle repeats until termination.

> **Scheduler** – Component that decides which ready-to-run process obtains the CPU next.

🖥️ Hardware Requirements for Multiprogramming

Requirement	Description
DMA-compatible secondary storage	Enables efficient data transfer between disk (or other secondary devices) and main memory without CPU intervention.
Address-translation support	Memory Management Unit (MMU) converts **logical addresses** generated by programs into **physical addresses** in main memory.
CPU dual-mode operation	Processor must support **user mode** (non-privileged) and **kernel mode** (privileged).

📁 Logical vs Physical Addresses

> **Logical Address** – Address generated by a program; an abstract reference to a location in memory.

> **Physical Address** – Actual location in main memory where data resides.

- **MMU** translates logical to physical addresses, providing:
  - **Security**: Programs cannot directly access others' memory spaces.
  - **Isolation**: Prevents buggy programs from corrupting memory of other processes.

Analogy:

- Logical address = "go 200 m straight, then turn left" (direction given to you).
- Physical address = exact coordinates of the house. Sharing only the logical directions protects the exact location.

⚙️ CPU Dual-Mode Operation

- **User Mode** (non-privileged)
  - Runs ordinary applications (e.g., PowerPoint, Chrome, games).
  - Preemptive scheduling applies.
- **Kernel Mode** (privileged)
  - Runs OS kernel routines (process management, file system, system calls).
  - Executes atomically; not preempted by user processes.
- **Mode Switching**
  - Triggered when a user application requests OS services (system call).
  - Controlled by the **Processor Status Word (PSW)**: a specific bit (mode bit) indicates current mode (0 = kernel, 1 = user).

Mode	Access Level	Typical Code
User	Restricted; cannot execute privileged instructions directly	Application programs
Kernel	Full access to hardware and memory	OS services, interrupt handlers

> **Privilege** – Ability of a program to execute certain instructions without interruption (e.g., ambulance bypassing traffic rules).

These notes capture the essential concepts of preemptive vs non-preemptive multiprogramming, process scheduling, lifecycle management, necessary hardware support, memory address translation, and CPU mode operation.

Kernel Mode 🖥️

> **Kernel Mode** – The executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Crashes in this mode are catastrophic and halt the entire system.

- Reserved for the **lowest-level, most trusted** OS functions.
- Any failure is **irreversible** for the running PC.

User Mode 👤

> **User Mode** – The executing code cannot directly access hardware or reference arbitrary memory. It can at most call library routines and must use system APIs to request privileged services.

- Crashes are **recoverable** because the OS isolates user processes.
- The majority of code on a computer runs in user mode.

System APIs 🔌

> **System APIs** – Interfaces that allow user-mode code to transition to kernel mode to access hardware or memory.

- Provide the controlled bridge between user mode and kernel mode.
- Details of specific APIs will be covered in a later lecture.

Multiprogramming 📚

Types of Operating Systems

Type	Description	Example
Uni-programming	Loads a single program from disk to main memory; CPU may idle during I/O.	"Dis" OS by Macrosoft
Multiprogramming	Loads multiple programs; while one waits for I/O, another can use the CPU.	Modern Windows, Linux, Unix

Benefits

- **Higher CPU utilization** – reduces idle time.
- **Increased throughput** – more programs completed per unit time.
- **Impression of CPU sharing** among programs (multiplexing).

Drawbacks

- Can cause **starvation** and reduced interactivity if not managed properly.

Preemptive vs Non-Preemptive Scheduling ⚡

Preemptive Multiprogramming

- **Forceful deallocation** of the CPU from a running process.
- Improves **responsiveness**; used in current OSes (Windows, Linux, Unix).
- Based on **time slices** (time sharing) or **priority**.

Non-Preemptive Multiprogramming

- Processes **voluntarily** release the CPU when:
  1. All instructions are executed.
  2. The process terminates.
  3. An I/O request is made.
  4. System code is invoked.
- May lead to **starvation** because a long-running process can block others.

> **Multitasking** – Multiprogramming with preemptive scheduling.

CPU Utilization & Throughput 📈

Metric	Uni-programming	Multiprogramming
CPU idle time	High (especially during I/O)	Low
Throughput	Low (few programs completed)	High (more programs completed per unit time)
Responsiveness	Poor	Better, especially with preemption

Architectural Requirements 🏗️

- **DMA-compatible secondary storage** – Efficient data transfer between storage and memory.
- **Address translation support** – Logical addresses must be mapped to physical addresses for security and protection.

- **Dual-mode processor** – Must support both **user mode** and **kernel mode**.

Mode Bit & Processor Registers 🔢

- **Mode bit** resides in the **Processor Status Word (PSW)** register.
  - 0 → **Kernel Mode** (privileged).
  - 1 → **User Mode** (non-privileged).
- Switching modes occurs when a user application requests OS services.

Key Quiz Review ❓

- **Q1:** OS implementation has architectural constraints – **True**.
- **Q2:** Real-time OS works on strict deadlines – **True**; primary goal for hard real-time systems is **efficiency**.
- **Q3:** During boot, OS is loaded **from disk to main memory** – **True**.
- **Q4:** System area stores the OS; user area stores programs – **True**.
- **Q5:** Throughput = total programs **completed** per unit time – **True** (not just loaded).
- **Q6:** Preemptive processes have forceful deallocation and better response time – **True**.
- **Q7:** Non-preemptive processes can lead to starvation and release CPU voluntarily – **True**.
- **Q8:** OS functions include **resource management**, **security**, and **control over system performance**; **reliability** is a goal, not a function.

User Mode vs Kernel Mode 🚀

> **User mode** – execution environment for regular applications (e.g., browsers, office software, games).
> **Kernel mode** – privileged environment where the operating system core runs.

- **Preemptive** (user mode): the OS can interrupt a running process when a higher-priority task arrives.
- **Non-preemptive** (kernel mode): kernel code runs atomically; it cannot be forced off the CPU by the OS.

Processor Status Word and Mode Bit 🧱

> The **Processor Status Word (PSW)** contains a **mode bit**:
> • 0 → kernel mode
> • 1 → user mode

The mode bit determines which instruction set (privileged vs. non-privileged) is allowed at any instant.

Mode Shifting (User ↔ Kernel) 🔄

> **Mode shifting** is the transition between user mode and kernel mode required to obtain OS services.

- When a user program needs a privileged service, it triggers a **system call** → CPU switches to kernel mode.
- After the OS routine finishes, control returns to user mode.

API and System Call Interface (SCI) 🌐

> An **API (Application Programming Interface)** or **SCI (System Call Interface)** is the contract that tells a program which OS services are available and how to request them.

Analogy: In India, an **e-Mitra** kiosk mediates between citizens and the government for services like the Aadhaar card. Similarly, an API mediates between user programs and the OS.

Function Types and Execution Context 📂

Function type	Where defined	Execution mode	Typical purpose
User-defined function	Source code of the program	**User mode**	Application logic written by the programmer
Library (predefined) function (e.g., printf)	Library file (*.a, *.so); prototype in header ()	**User mode**	Common utilities provided by the language runtime
System call (e.g., fork)	Implemented inside the OS kernel	**Kernel mode**	Access privileged OS services (process creation, I/O, etc.)

> **Header files** contain only **function declarations** (prototypes) for type checking; the actual implementation resides in library binaries.

Example: C Program Execution Flow 💻

```
#include

int main(void) {
    int a, b = 1, c = 2;
    a = b + c;              // user-defined computation
    printf("%d\n", a);      // library call (user mode)
    fork();                 // system call → kernel mode
    return 0;
}
```

1. **Compilation** – source → machine instructions.
2. **Loading** – the instruction stream is placed in main memory.
3. **Execution** – CPU fetches and runs instructions sequentially.
4. **Mode shift** occurs only at the fork() call; all other statements stay in user mode.

System Call: fork 🛠️

> fork creates a **child process** that is a duplicate of the calling (parent) process.

- Implemented as a **kernel routine**; cannot be executed directly in user mode.
- Invocation generates a **software interrupt** that transfers control to the OS (mode shift).

Compilation to Instructions: BSA vs. SVC 📐

- **User-level code** (functions, arithmetic) → compiled into **Branch and Save Address (BSA)** instructions.
  - BSA = non-privileged; runs in **user mode**.
- **Kernel-level routines** (system calls) → compiled into **Supervisory Call (SVC)** instructions.
  - SVC = privileged; executes in **kernel mode** and raises a **software interrupt**.

Interrupts: Software vs. Hardware ⚡

Type	Origin	Triggers	Typical use
Software interrupt	Generated by an instruction like SVC	System call request	Switch to kernel mode for OS services
Hardware interrupt	External hardware device (timer, I/O)	Asynchronous event	Immediate OS response to device activity

Both interrupt types inform the operating system that an action is required, but **software interrupts** are the mechanism used for mode shifting during system calls.

🚨 Hardware Interrupts

> **Hardware interrupt** – A signal generated by hardware (e.g., overheating CPU) that forces the processor to temporarily suspend the current task and invoke an interrupt service routine.

- Example: Playing *Rocket League* on a low-spec PC (2 GB RAM) causes the CPU to overheat → hardware generates an interrupt → OS requests closing programs.

💻 Software Interrupts & SVC

> **Software interrupt** – An interrupt generated by the operating system during runtime to inform the OS of a specific activity.

- **SVC (Supervisory Call)**
  - Generated at **compile time** for OS routines.
  - At **runtime** the SVC triggers a software interrupt.
- **ISR (Interrupt Service Routine)**
  - Handles any interrupt (hardware or software).
  - Two main duties:
    1. Change the **mode bit** in the PSW (Processor Status Word) from user (1) to kernel (0).
    2. Locate the required service using the **dispatch table**.

🔧 Mode Bit & Processor Status Word (PSW)

> **Mode bit** – A single bit in the PSW indicating the current privilege level: 0 = kernel mode, 1 = user mode.

Mode-shifting sequence

1. ISR sets mode bit **1 → 0** (user → kernel).
2. ISR uses the **dispatch table** to find the address of the required routine (e.g., a fork).
3. Routine executes instructions **atomically** in kernel mode.
4. After the last instruction, ISR restores mode bit **0 → 1** (kernel → user) for security.

📋 Dispatch Table

> **Dispatch table** – A kernel-resident data structure in RAM that maps service names (e.g., *fork*) to their memory addresses.

- Allows ISR to quickly locate the correct service routine.
- Enables sequential execution of the service's instructions.

🛠️ Function Types & Execution Modes

Function Type	Execution Mode	Typical Location
User-defined	User mode	Application code
Built-in (library)	User mode	Standard libraries
System call	Kernel mode	OS kernel (via ISR)

- Library functions (e.g., printf) may invoke a **system call**, causing a mode shift just like any other kernel service.

🔄 Traps & Mode Switching

> **Trap** – Another term for an interrupt generated by a software condition (e.g., a system call).

- **User → Kernel**: Generates a trap → ISR changes mode bit to 0.

- **Kernel → User**: No trap; ISR simply restores the mode bit to 1 after service completion.

✅ Privilege Instructions (Kernel-Only Operations)

Operation	Allowed in Kernel?	Reason
Enable/disable interrupts	✅	Privilege instruction
Read system time	✅	Can be done in user mode, but also allowed in kernel
Context switching	✅	Requires kernel control
Clear memory / remove process	✅	Sensitive operation

Correct answers: Enable/disable interrupts, context switching, clear memory (options a, c, d).

🔄 Mode-Bit Rules

- **Kernel → User**: Set mode bit to **1**.
- **User → Kernel**: Set mode bit to **0**.

These rules are used in the multiple-select question about switching modes.

📚 Key Concepts Recap (Blockquotes)

> **Interrupt Service Routine (ISR)** – The code that runs in response to an interrupt; it changes privilege mode, accesses the dispatch table, executes the requested service atomically, and restores the original mode.

> **Atomic execution** – Execution of a sequence of instructions without interruption, guaranteed in kernel mode for privileged services.

> **Privilege instruction** – An instruction that can only be executed in kernel mode (e.g., enabling/disabling interrupts, context switching).

🔀 Fork Behavior and Print Count

📈 How fork Duplicates Execution

- When a fork system call is executed, the **current process** is duplicated, creating a **child process**.
- Both the parent and child continue execution from the point just after the fork.
- Any subsequent statements (e.g., print) are therefore executed **by each process** that exists at that point.

🔢 Print Count Formula

> **Definition:** The number of times a statement after the last fork is executed equals 2^n, where n is the total number of fork calls that have been performed.

Number of fork calls (n)	Times the following statement prints
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$

- The pattern emerges because each fork **doubles** the number of active processes, and every active process executes the subsequent statement once.

📂 Program vs Process

📁 Program

> **Definition:** A **program** is a set of instructions and static data stored on secondary storage (e.g., hard disk) in an executable file format (e.g., .exe, a.out).

🖥️ Process

> **Definition:** A **process** is an **instance** of a program that resides in main memory and is being executed by the CPU. It includes the program's code, its **dynamic data**, and the execution context (registers, stack, etc.).

🗂️ Memory Layout

- **Program (on disk):** code + static data → *executable file*.
- **Process (in memory):**
  - **Text segment:** executable code.
  - **Data segment:** static (global) variables.
  - **Heap:** dynamically allocated memory (e.g., via malloc).
  - **Stack:** function call frames, local variables.

📊 Data Allocation Types

🗄️ Static Data

> **Definition: Static data** has a fixed size known at compile-time and is allocated **once** when the program is **loaded** into memory.

🔄 Dynamic Data

> **Definition: Dynamic data** may have a size that is not known until the program runs; it is allocated **at run-time** (e.g., using malloc).

⏱️ Timing Overview

Allocation Phase	Static Data	Dynamic Data
Compile time	No (only syntax checking)	No
Load time	Yes (memory reserved when program is loaded)	No
Run time	No	Yes (allocated during execution)

🧩 Dynamic Arrays in C and C++

❌ Invalid Declaration in C

- Syntax such as int A[n]; where *n* is read from the user **is not allowed** in standard C (C99 introduced variable-length arrays, but the transcript treats it as invalid).

✅ Valid Approach Using malloc

```
#include

int *createArray(int n) {
    /* Allocate memory for n integers at run-time */
    int *A = (int *)malloc(n * sizeof(int));
```

```
    return A;   /* Caller must check for NULL and later free(A) */
}
```

- malloc requests a **contiguous block** of memory from the heap, enabling arrays whose size is determined at run-time.

🛠️ Processor Status Word (PSW) Mod Bit

> **Fact:** The **mod bit** is **not stored** in main memory or cache; it resides in the **processor register** known as the **Processor Status Word (PSW)**, which is part of the CPU state at system boot.

📌 Dynamic Memory Allocation with Pointers

> **Definition**: A *pointer* holds the memory address of a variable. When used with malloc, it can refer to a *dynamic array* whose size is determined at run time.

- Declare a pointer, read the desired size n from the user, then allocate:
  - malloc reserves n elements of the required type.
  - The pointer receives the address of the **first element** of the allocated block.
- Static allocation (e.g., int x;) occurs at **load time**; the variable occupies a fixed 2-byte region.
- Dynamic allocation occurs at **run time**, allowing the array size to be decided while the program runs.

Allocation Type	Time of Allocation	Memory Location	Size Flexibility
Static	Load time	Fixed address in program image	Fixed
Dynamic	Run time (malloc)	Heap region, address stored in a pointer	Flexible (determined by n)

🖥️ Program vs. Process

> **Program**: An executable file (.exe, .out, etc.) stored on the hard disk; it contains code and static data but does **not** use system resources until loaded.

> **Process**: The **instance** of a program that is loaded into memory and executing; it actively consumes CPU, memory, and other resources.

- **Analogy**:
  - *Program* → a **recipe** in a cookbook (inactive).
  - *Process* → a **chef** actually preparing the dish (active).
- Key distinctions:

Aspect	Program	Process
Location	Hard disk	Main memory (RAM)
State	Passive (does not consume resources)	Active (uses CPU, memory, I/O)
Existence	Exists before execution	Exists only while execution is ongoing

---

🧩 Process as an Abstract Data Type

> **Definition**: A *process* can be viewed as an abstract data type (ADT) characterized by four components: **definition**, **representation**, **operations**, and **attributes**.

Component	Description (process)

Definition	“A program in execution; an active entity that uses computer resources.”
Representation	Memory layout comprising Text, Data, Heap, and Stack sections.
Operations	Creation (loading), execution, termination, context switching.
Attributes	Process ID, state (running, waiting, etc.), resource usage, priority.

Comparison with a linked list (another ADT):

Aspect	Linked List	Process (ADT)
Definition	Linear collection of nodes	Program in execution
Representation	Nodes with data + pointer to next node	Memory sections (Text, Data, Heap, Stack)
Operations	Insert, delete, traverse	Create, schedule, terminate, context-switch
Attributes	Head, tail, size, length	PID, state, resource allocation, priority

---

📂 Memory Layout of a Process

🧩 Text (Code) Section

> Contains the **executable instructions** of the program.

- Read-only; shared among processes running the same program.

📊 Data Section

> Holds **global and static variables**, split into:

- **Initialized data** – variables with explicit initial values (e.g., int x = 1;).
- **Uninitialized data** (BSS) – variables declared without an initial value (e.g., int x;).

Sub-section	Example	Stored in
Initialized	int x = 5;	Data segment (initialized)
Uninitialized	int y;	BSS segment (uninitialized)

📥 Heap

> Region used for **dynamic memory allocation** (malloc, calloc, realloc, free).

- Grows upward as more memory is requested.
- Managed by the runtime library and the operating system.

📥 Stack & Activation Records

> Stack stores **activation records** (also called *call frames*) for function calls.

Activation Record Contents

- Space for **local variables** of the function.
- The **return address** (the instruction to resume after the function finishes).

Example Flow

1. Execution reaches main().
    - An activation record for main is **pushed** onto the stack.
    - Stores locals k, r, l and the return address (end of program).
2. main calls another function f.
    - A new activation record for f is **pushed** above main's record.
    - Stores locals z, e, p, q, l and f's return address (the line after the call in main).
3. When f completes, its activation record is **popped**, control returns to the address saved in f's record (back to main).
4. After main finishes, its activation record is also **popped**, and the process terminates.

- The stack **grows** with each nested call and **shrinks** as calls return.

📚 Homework

- Study the **VSS (block start by symbol) extension** and understand how it modifies or interacts with program structure.

📦 Process Operations

> **Create** – allocating memory and other resources required for a process; the process becomes *created*.

> **Terminate** – de-allocating all resources that were given to the process; the process is destroyed.

> **Schedule** – selecting one of the ready processes to run on the CPU when several are waiting.

> **Dispatch** – giving the selected process the CPU so its instructions can be executed.

> **Execute / Run** – the CPU fetches and executes instructions from the process's code section.

> **Block / Wait** – the process is suspended because it issued a system call or I/O request and must wait for the operation to complete.

> **Suspend** – moving the entire process image from main memory to secondary storage (disk).

> **Resume** – bringing a previously suspended process back from disk into main memory so it can continue execution.

Operation Summary

- **Create** → resource allocation → process enters **New** state.
- **Terminate** → resource de-allocation → process enters **Terminated** state.
- **Schedule** → decision among ready processes.
- **Dispatch** → CPU granted to chosen process.
- **Execute** → CPU runs instructions.
- **Block / Wait** → process leaves CPU, waits for I/O or system service.
- **Suspend** → memory → disk.
- **Resume** → disk → memory.

🧩 Process Attributes

> Attributes are the pieces of information that describe a process; they are stored in the **Process Control Block (PCB)**.

Category	Examples of Attributes
Identification	Process ID (PID), Parent PID, Group ID
CPU-related	Program Counter (points to next instruction), Priority, Current State, CPU-burst time
Memory-related	Size of memory occupied, Memory limits, Base & limit registers

File-related	List of open files used by the process
Device-related	Devices allocated (e.g., printers, disks)
Accounting-related	Resources consumed (CPU time, I/O usage)

Key Definitions

> **Program Counter (PC)** – a register that always points to the next instruction to be executed.

> **Priority** – the importance level assigned by the OS to influence scheduling decisions.

> **State** – the current status of a process (e.g., Running, Ready, Blocked).

> **Burst time** – total CPU time a process has spent executing.

📁 Process Control Block (PCB)

> The **PCB** (also called *process descriptor*) is the OS data structure that holds all attributes of a process.

- Each process has its own PCB.
- All PCBs reside in main memory.
- The complete set of PCB fields is referred to as the **process context** or **process environment**.

🔄 Process State Transition Diagram

> A process moves through a series of states during its lifetime; the diagram shows permissible transitions.

Primary States (covered in this lecture)

1. **New** – process is being created; resources are allocated.
2. **Ready** – process is prepared to run and is waiting for CPU allocation.
3. **Running** – process currently holds the CPU and executes instructions.
4. **Blocked / Wait** – process cannot continue until an I/O operation or system call completes.
5. **Terminated** – process has finished execution; resources are reclaimed.

Additional states (mentioned for future lectures)

- **Suspend-Ready** – suspended process that is ready to be resumed.
- **Suspend-Block** – suspended while waiting for I/O.

State Transition Flow (simplified)

- **New → Ready** (after creation).
- **Ready → Running** (by scheduling & dispatch).
- **Running → Block/Wait** (when issuing I/O or system call).
- **Running → Ready** (preempted by scheduler).
- **Running → Terminated** (process completes).
- **Block/Wait → Ready** (I/O completes).
- **Any state → Suspend** (memory-to-disk).
- **Suspend → Ready / Block** (resume from disk).

⏱️ Scheduling vs. Dispatch

> **Scheduling** – the OS decision process that selects one ready process to run next.

> **Dispatch** – the act of giving the selected process control of the CPU so its instructions can be executed.

- Only **one** process can occupy the CPU at a time in a single-processor system.
- The scheduler works on the set of *ready* processes (e.g., P_1, P_2, ..., P_n).
- After scheduling, the dispatcher loads the process's context (PC, registers) and starts execution.

Conceptual Illustration

- **Process in memory** → **Scheduler picks P_k** → **Dispatcher loads P_k's PCB** → **CPU executes P_k**.

The process itself never physically moves to the CPU; the CPU merely gains the right to execute its instructions while the process remains in main memory.

📚 Additional Points

- A **program** is static code; a **process** is the dynamic execution of that code together with its allocated resources.
- The OS handles all I/O operations on behalf of a process; the process issues a system call, the OS performs the actual device interaction.

Process Placement & Swapping 💾

> **Process** – A program that has been loaded into **main memory** and is ready for execution by the CPU.

- A process resides **either** in main memory **or** on the hard disk.
- When main memory becomes crowded, the operating system may **swap** a process from memory to disk to free space.
- Conversely, a program on disk is loaded into memory to become a **process**.

Process State Diagram 🔄

States and Transitions

- **New** – Process is created; resource allocation occurs.
- **Ready** – Process is in main memory, awaiting CPU assignment.
- **Running** – CPU is executing the process's instructions.
- **Blocked / Waiting** – Process has issued an I/O or system-call request and must wait for completion.
- **Terminated** – All instructions have finished; the process leaves main memory.

> **Scheduling** – Selecting which ready process will receive the CPU.
> **Dispatching** – Giving the selected process control of the CPU.

Typical flow:

1. **New → Ready** (after allocation).
2. **Ready → Running** (scheduler selects, dispatcher assigns).
3. **Running → Blocked** (process makes an I/O/system call).
4. **Blocked → Ready** (I/O completes, OS returns results).
5. **Running → Terminated** (all instructions executed).

Multiprogramming vs. Uniprogramming 🖥️

Feature	Multiprogramming OS	Uniprogramming OS
Ready queue	Exists; holds many ready processes	Not needed; only one process
Process count in memory	Multiple processes simultaneously	Single process only
State diagram	Includes **Ready** state	Skips **Ready**; New → Running directly
CPU utilization	Higher (CPU can switch among ready processes)	Lower (CPU idle when the single process blocks)

Preemptive vs. Non-Preemptive ⚡

Aspect	Non-Preemptive	Preemptive
When does a process leave CPU?	Only after it finishes or voluntarily blocks for I/O	Can be forced off CPU by the scheduler (e.g., time-slice expiration, higher-priority arrival)
State transition	Running → Blocked or Running → Terminated	Running → Ready (forced preemption) in addition to the above
Example from diagram	Process exits CPU only when completed or I/O needed	Adding a "forceful deallocation" transition creates a preemptive behavior

Scheduling & Dispatching 📊

> **Scheduling** – The OS **selects** a process from the ready pool.
> **Dispatching** – The OS **hands over** CPU control to the selected process.

- **Scheduling** decides *what* runs; **dispatching** decides *who* runs now.

Capacity of Process States 📈

State	Theoretical maximum number of processes
Ready	Unlimited (bounded only by RAM size and OS limits)
Blocked / Waiting	Unlimited (same constraints as Ready)
Running	1 per CPU core (e.g., 1 for a single-processor system, *n* for *n* CPUs)

- The **ready** and **blocked** queues can grow until memory is exhausted.
- Only one process per CPU can be in the **running** state at any instant.

Suspension (Concept) ⏸️

> **Suspension** – Temporarily moving a process from main memory to secondary storage (disk) without terminating it, freeing memory for other processes.

- Used when the OS needs to reclaim memory while preserving the process's state for later resumption.

Key terms are bolded for quick reference; definitions are provided in blockquotes.

Process Suspension Overview 📦

> **Suspension** – moving a process from main memory to secondary storage (disk) to free memory and improve system performance.

- **Why suspend?**
  - Main memory is limited; too many active processes cause overheating (analogy: overcrowded classroom).
  - Shifting less-active processes to disk reduces contention and keeps the CPU focused on active work.
- **Typical scenario**
  - OS can handle **100** processes in memory but **150** are ready.
  - The extra **50** processes are transferred to disk (suspended) until needed.

States Eligible for Suspension 🔄

> Only processes that reside in **main memory** can be suspended.

State	In Memory?	Can Be Suspended?	Notes
New	No (just created)	❌	Not yet loaded into memory.
Ready	Yes	✅	Most desirable for suspension.
Running	Yes	✅ (possible)	Generally undesirable; shown with dotted transition lines.
Blocked	Yes	✅	Often suspended if waiting long for I/O.
Suspended Ready	No (on disk)	❌	Already suspended.
Suspended Blocked	No (on disk)	❌	Already suspended.
Terminated	No	❌	Process has finished.

Most Desirable State for Suspension 🎯

> **Ready state** – the process is not executing and not performing I/O, making it ideal for suspension.

- The process is waiting for the OS scheduler; moving it to disk does not interrupt active computation or I/O.

Suspension and Resume Operations ↔️

- **Suspend** – transfer from memory → disk.
- **Resume** – transfer from disk → memory.
- Possible suspension sources:
  1. **Ready** → *Suspended Ready*
  2. **Running** → *Suspended Ready* (possible but not preferred)
  3. **Blocked** → *Suspended Blocked*
- Resumption paths:
  - *Suspended Ready* → Ready → Running
  - *Suspended Blocked* → Suspended Ready → Ready → Running
  - *Suspended Blocked* → Blocked (if I/O still pending)

Seven-State Process Transition Diagram 📈

> The diagram describes how a process moves among the seven fundamental states.

1. **New** – process created; PCBs allocated.
2. **Ready** – loaded into memory, waiting for CPU scheduling.
3. **Running** – CPU executes the process.
4. **Blocked** – awaiting I/O or other event.
5. **Suspended Ready** – moved to disk from Ready (or Running) state.
6. **Suspended Blocked** – moved to disk from Blocked state.
7. **Terminated** – process finishes execution.

Typical Transition Paths

- New → Ready (loaded into memory)
- Ready → Running (scheduler dispatch)
- Running → Ready (preemption)
- Running → Blocked (I/O request)
- Running → Terminated (completion)
- Blocked → Ready (I/O completion)
- Ready / Running / Blocked → Suspended Ready / Suspended Blocked (suspension)
- Suspended Ready → Ready (resume)

- Suspended Blocked → Blocked (resume) or → Suspended Ready (if I/O finished)

Dotted arrows indicate transitions that are possible but generally avoided (e.g., suspending a Running process).

Resource Preemption Example ⚙️

- A process in **Ready** holds resources **R1, R2, R3**.
- OS may preempt **R1** for higher-priority needs.
- The process loses a required resource → it cannot run → moves to **Blocked**.
- Once **R1** is returned, the process returns to **Ready**.

Summary of Key Points 📚

- Suspension frees memory by moving processes to disk, enhancing overall performance.
- Only processes residing in memory (Ready, Running, Blocked) are eligible for suspension.
- The **Ready** state is the most efficient point to suspend a process.
- The seven-state model captures all possible states and transitions, including suspension and resumption paths.
- Resource preemption can force a Ready process into Blocked, after which it may be suspended if it remains idle.

📋 Process States & Transitions

> **Process State** – A condition describing a process's activity (e.g., *new*, *ready*, *running*, *blocked*, *suspended*).

- **New** → entered when a program is created.
- **Ready** → waiting in main memory for CPU time.
- **Running** → currently executing on the CPU.
- **Blocked** → waiting for an I/O operation to finish.
- **Suspended Block** → blocked *and* swapped out to disk to free main-memory resources.
- **Suspended Ready** → logically ready (I/O completed) but still resides on disk.

🔄 Transition Pathways

From → To	Condition	Where is the process stored?
Blocked → Suspended Block	OS needs memory for other processes	Swapped to **disk**
Suspended Block → Suspended Ready	I/O operation completes	Still on **disk**
Suspended Ready → Ready	OS decides to load it back	**Main memory** after *swapping/loading*
Suspended Block → Ready	OS has enough memory to load immediately	**Main memory** (rare; usually goes through Suspended Ready)

🧩 Suspension, Swapping & Loading

> **Swapping (Loading)** – The act of moving a process's image between main memory and secondary storage (disk).

- When a process is **blocked**, it may be **suspended** to free RAM.
- While in **Suspended Block**, the process:
  - Waits for I/O.
  - Occupies space on **disk**, not RAM.
- After I/O finishes, the process becomes **Suspended Ready**:
  - Logically ready but still on disk.
  - Requires a **swap-in** operation before it can enter the **Ready** state.

❓ Why Use *Suspended Ready* Before *Ready*?

- **Memory Management:** The OS may not have enough free frames to load the process immediately.
- **Resource Allocation:** Keeping the process on disk allows the OS to schedule other active processes first.

- **Control:** The OS explicitly decides when to bring the process into RAM based on system load, memory usage, and scheduling policies.

📂 Process Control Block (PCB)

> **PCB** – Data structure that holds all information about a process (e.g., PID, program counter, registers, state, pointers).

- Contains a **pointer** to the next PCB, forming a linked list used by queues.
- Attributes include:
  - Process identifier
  - Program counter
  - Register set
  - Current state
  - Resource pointers (e.g., I/O device queues)

📊 Scheduling Queues Overview

🗃️ Queues in Main Memory

Queue Type	Corresponding State	Contents (PCBs)	Implementation
Ready Queue (RQ)	Ready	Processes ready for CPU	FIFO linked list
Block (Device) Queue	Blocked	Processes waiting for specific I/O devices	One queue per device

- **Ready Queue** holds PCBs that are in main memory and eligible for CPU allocation.
- **Device Queues** (e.g., for disk, printer) hold PCBs blocked on that particular device.

💾 Queues on Disk

Queue Type	Corresponding State	Contents (PCBs)
Job Queue	New	Programs awaiting loading into memory
Suspend Queue	Suspended Block & Suspended Ready	Processes swapped out to disk

- **Job Queue** stores programs stored on disk that have not yet been admitted to memory.
- **Suspend Queue** holds PCBs of processes that have been swapped out; they may be in either the *Suspended Block* or *Suspended Ready* stage.

🔄 State Queuing Diagram (Combined View)

1. **Job Queue** → scheduler selects a program → loads into **main memory** → becomes **Ready** (placed in Ready Queue).
2. **Ready Queue** → dispatcher gives CPU → process moves to **Running**.
3. While running, the process may request I/O → moves to **Blocked** → placed in the appropriate **Device Queue**.
4. If memory pressure exists, OS may **suspend** the blocked process → it moves to **Suspended Block** (on disk, in Suspend Queue).
5. After I/O completes, the process becomes **Suspended Ready** (still on disk).
6. When sufficient memory is available, OS **loads** it back → transitions to **Ready** (returns to Ready Queue).

📌 Key Takeaways

> - **Suspended states** free RAM by moving blocked processes to disk.
> - **Suspended Ready** acts as an intermediate logical-ready stage until the OS can allocate memory.
> - **Queues** organize PCBs according to their current state and location (memory vs. disk).
> - The **PCB pointer** links PCBs into these FIFO queues, enabling efficient scheduling.

📥 Job Queue & Ready Queue

- **Job Queue (job Q)** – holds PCBs of processes that are ready to be loaded into main memory.

> *"Job Q contains the list of the PCB which are ready to be loaded in the main memory."*

- **Ready Queue (ready Q)** – contains PCBs of processes that reside in memory and are ready to be scheduled on the CPU.

⚙️ Scheduler Overview

- **Scheduler** – OS component that makes decisions about *which process* gets CPU time.

> *"Scheduler is the component of operating system that makes the decision."*

- Decision points occur in **job Q**, **ready Q**, and **suspend Q** (block queue is not the focus).

🗂️ Types of Schedulers

Scheduler Type	Activation Frequency	Primary Responsibility	Affected Queue(s)
Long-term	Infrequent (rare)	Move processes from **new** (job Q) to **ready** (ready Q)	Job Q → Ready Q
Short-term	Frequent (each CPU cycle)	Select a ready process for execution on the CPU (CPU scheduler)	Ready Q → CPU
Medium-term	Moderate (when swapping needed)	Suspend and later resume processes (swap between memory and secondary storage)	Ready Q ↔ Suspend Q (block Q)

📂 Long-term Scheduler

- Loads selected programs from disk into memory.
- Controls **degree of multiprogramming** – the number of processes simultaneously residing in main memory.

> *"Long-term scheduler decides which programs to load in memory; it changes the state from new to ready."*

🚀 Short-term Scheduler

- Chooses one process from the **ready Q** for CPU execution.
- Often called **CPU scheduler** because it determines the next CPU-bound process.

🔄 Medium-term Scheduler

- Handles **process suspension** and **resumption** (swapping).
- Works independently of the dispatcher; does **not** perform context switching.

🔁 Dispatcher & Context Switching

- **Dispatcher** – carries out the **context-switching** activity after the short-term scheduler selects a process.

> *"Dispatcher is responsible for carrying out the activity of context switching."*

- **Context Switching** – saving the PCB of the currently running process and loading the PCB of the next process.
  - If preemption or I/O request occurs, the PCB is saved to **ready Q** (preemption) or **block Q** (I/O, system call, interrupt).
  - The CPU never remains idle; the next PCB is loaded immediately.
- Alternate names: **dispatch latency**, **CPU scheduling overhead**, **context-switch time**.

🔄 Process State Transitions (State-Queue Diagram)

1. **Job Q → Ready Q** – performed by the **long-term scheduler**.
2. **Ready Q → CPU** – performed by the **short-term scheduler** and **dispatcher**.
3. From **CPU**, possible transitions:
   - **Terminate** – all instructions completed.
   - **Block Q** – needs I/O, system call, or is interrupted.
   - **Ready Q** – time slice (**SL**) expires → preemption.
4. **Block Q → Ready Q** – after I/O completion or when the block condition ends.
5. **Suspend/Resume** – managed by the **medium-term scheduler** (process may move between Ready Q and a suspended state).

📊 Degree of Multiprogramming

- Defined as the count of processes residing in main memory simultaneously.

> *"If P1, P2, P3 are in memory, the degree of multiprogramming is three."*

- **Controlled by the long-term scheduler**, which decides how many new programs to admit from disk.

👶 Fork System Call

- Invoking **fork** creates a **child process** from the calling (parent) process.

> *"Execution of fork system call creates the child process."*

- The new child process follows the same state-queue flow as any other process.

🕒 User-to-Kernel vs Process Switch Times

> **T_1** – time to switch from **user mode** to **kernel mode**.
> **T_2** – time for a **process (context) switch**.

- The process switch includes the mode-shift, so it must contain **T_1**.
- Therefore $T_2 > T_1$.

📊 CPU Scheduling Overview

> **CPU scheduling** – design and implementation of the **short-term scheduler** that decides which ready process runs next on the CPU.

🎯 Functions & Goals of the Short-Term Scheduler

- **Function:** select a process from the **ready queue (RQ)** for execution on the CPU.
- **Goals:**
  - Maximize **CPU utilization**, **throughput**, and overall **efficiency**.
  - Minimize **waiting time**, **turnaround time**, and **response time**.

⏱️ Process Timing Metrics

Metric	Definition	When Measured
Arrival time	Time a process first enters the ready queue (moves **New → Ready**).	At load from disk to main memory.
Waiting time	Total time a process spends in the ready queue awaiting CPU service.	Sum of all ready-state intervals.
Burst time	CPU time actually spent executing the process (**running state**).	Accumulated across all CPU bursts.

I/O burst time	Time a process spends blocked waiting for I/O completion.	While in **blocked** state.
Completion time	Moment the process finishes execution (last instruction retired).	Transition **Running → Terminated**.
Turnaround time	Lifetime of a process from arrival to completion.	$\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time}$

> **Turnaround time** reflects the total elapsed time a process experiences, encompassing all CPU, waiting, and I/O periods.

🧱 Relating Waiting Time to Turnaround Time

If

- T_{total} = total elapsed time for the process,
- T_{run} = total burst time,
- T_{IO} = total I/O burst time,

then

$$\text{Waiting Time} = T_{\text{total}} - (T_{\text{run}} + T_{\text{IO}})$$

📅 Scheduling Concepts

📋 Schedule & Schedule Length

- **Schedule** – ordered list of processes as they finish execution (e.g., P_1, P_2, P_3).
- **Schedule length** – total time required to complete **all** n processes according to a given schedule.

🔢 Number of Possible Schedules

Scheduling type	Number of distinct schedules for n processes
Non-preemptive	$n!$ (factorial) – each process must run to completion before the next begins.
Preemptive	Theoretically **infinite** – processes can be interrupted arbitrarily, yielding countless execution orders.

- Example with $n = 3$ (processes P_1, P_2, P_3):
  - Non-preemptive → $3! = 6$ possible orders.
  - Preemptive → unlimited permutations (e.g., P_1 runs 2 ms, preempted, P_2 runs 1 ms, etc.).

📅 Schedule Length & Throughput

> **Schedule Length (L)** – total time taken to complete all processes in a schedule.
> $L = \text{completion time of last process} - \text{arrival time of first process} = \max(C_i) - \min(A_i)$

> **Throughput** – number of processes completed per unit time.
> $\text{Throughput} = \frac{n}{L}$
> where n = number of processes.

Symbol	Meaning
C_i	Completion time of process *i*
A_i	Arrival time of process *i*

L	Schedule length
n	Total number of processes
Δ	Dispatch latency (context-switching overhead)

🔄 Context-Switching Overhead

> **Context-Switching Time (Δ)** – time the dispatcher spends loading a process's PCB from the ready queue (RQ) onto the CPU. *Saving* the PCB of the pre-empted process is **ignored** for simplicity.

- Δ is counted **once** each time a new process is dispatched.
- It is **not** included in the burst or I/O times of a process.

⏱️ Process Time Metrics

> **Arrival Time (AI)** – moment a process enters the ready queue.

> **Burst Time (BT)** – CPU time a process actually runs (also called *run time*).

> **I/O (Blocked) Time (IT)** – time a process spends waiting for I/O while in the blocked state.

> **Completion Time (CT)** – moment a process finishes execution.

> **Turnaround Time (TT)** – total lifetime of a process.
> $\mathrm{TT} = C_i - A_i$

> **Waiting Time (WT)** – time a process spends in the ready queue (ready state).

The relationship among the times:
$\mathrm{TT} = \mathrm{WT} + \mathrm{BT} + \mathrm{IT}$

🖥️ Scheduling Types

Type	Preemptive	Non-preemptive
Definition	Scheduler may forcibly remove a running process (e.g., time-slice expiry, higher-priority arrival).	Process runs to completion or voluntarily yields the CPU.
Typical Use	Real-time, time-sharing systems.	Batch jobs, simple systems.
Starvation Risk	Lower (if proper algorithms are used).	Higher, especially with long-burst processes.

- **Preemptive** often leads to many possible schedules: $n!$ permutations for n processes (assuming infinite preemption points).
- **Non-preemptive** schedules are limited to the order of arrivals.

⚙️ Process Characteristics

Characteristic	CPU-bound	I/O-bound
Burst Profile	Long CPU bursts, short I/O bursts.	Short CPU bursts, long I/O bursts.
System Impact	Keeps CPU busy, may idle I/O devices.	Keeps I/O busy, may leave CPU idle.

Ideal Mix	A balanced mix maximizes both CPU and I/O utilization.	

📋 First-Come-First-Serve (FCFS)

- **Selection Criterion**: Earliest **arrival time**.
- **Mode of Operation**: **Non-preemptive**.
- **Conflict Resolution**: If two processes arrive simultaneously, choose the one with the **lower process ID (PID)**.
- **Assumptions for Analysis**
  1. Time measured in **clock ticks** (no real-world units).
  2. **I/O times** and **scheduling overhead (Δ)** are taken as **zero** for simplicity (added later in the course).

⚠️ Starvation in FCFS

- A **long process** (large burst time) that arrives first can occupy the CPU for a prolonged period.
- Subsequent **short processes** wait indefinitely → **starvation**.
- This issue stems from the **non-preemptive** nature of FCFS.

🧱 Example: Computing FCFS Metrics

Given a table with **Process ID**, **Arrival Time (AT)**, and **Burst Time (BT)**, compute:

1. **Completion Time (CT)** – cumulative sum of burst times respecting arrival order.
2. **Turnaround Time (TT)** – $CT - AT$.
3. **Waiting Time (WT)** – $TT - BT$.

Process ID	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TT)	Waiting Time (WT)
P_1	...	...	...	$CT_1 - AT_1$	$TT_1 - BT_1$
P_2	...	...	...	$CT_2 - AT_2$	$TT_2 - BT_2$
P_3	...	...	...	$CT_3 - AT_3$	$TT_3 - BT_3$

Fill in the "..." with the specific numbers from the problem statement.

Steps

1. Order processes by **arrival time** (use PID as tie-breaker).
2. For the first process: $CT = AT + BT$.
3. For each subsequent process:
   - If the CPU is idle (next arrival > previous CT), start at its arrival time.
   - Otherwise, start when the previous process finishes: $CT = \text{previous } CT + BT$.
4. Compute TT and WT using the formulas above.

📚 Key Takeaways

- **Schedule length** captures the collective execution window of all processes.
- **Throughput** inversely depends on schedule length.
- **Δ (dispatch latency)** is considered only for loading the PCB; saving is ignored to keep analysis simple.
- Distinguish **CPU-bound** vs **I/O-bound** to understand workload impact.
- **FCFS** is easy but can cause **starvation**; conflict resolution defaults to lower PID.

These concepts form the foundation for analyzing and designing more sophisticated scheduling algorithms.

📊 Gantt Chart Basics

> *A Gantt chart is a visual representation of a CPU schedule, showing which process occupies the CPU over time.*

- **Starts at time 0** regardless of when processes actually arrive.
- **X-axis:** elapsed time.
- **Bars:** labeled with process IDs (e.g., *P1*, *P2*).
- Shows **schedule length** – the total time from start (0) to the completion of the last process.

🗃️ FCFS (First-Come-First-Served) Scheduling Rules

> *FCFS schedules the process that arrived earliest; if multiple processes arrive simultaneously, the one with the lower process ID runs first.*

- **Non-preemptive:** a running process runs to completion unless it voluntarily yields (no I/O in given examples).
- **Tie-breaking rule:** lower **process ID** gets priority when arrival times are equal.
- If the CPU is idle because no process has arrived yet, the chart shows an **Idle** interval.

📈 Calculating Key Times

Metric	Formula	Note
Completion Time	C_i = time when process *i* finishes execution	Directly read from the Gantt chart end of the bar.
Turnaround Time	$T_i = C_i - A_i$	A_i = arrival time. With $A_i = 0$, turnaround equals completion.
Waiting Time	$W_i = T_i - B_i$	B_i = burst time. Can also be read as the idle period before the process starts.
Schedule Length	L = last completion time	Total time from 0 to the end of the final process.

🧱 Example 1: All Processes Arrive at Time 0

Process	Burst B_i	Start	Completion C_i	Turnaround T_i	Waiting W_i
P1	4	0	4	$4 - 0 = 4$	$4 - 4 = 0$
P2	3	4	7	$7 - 0 = 7$	$7 - 3 = 4$
P3	5	7	12	$12 - 0 = 12$	$12 - 5 = 7$

Gantt chart (textual):

```
0----4----7-----------12
| P1 | P2 |    P3    |
```

- Schedule length = **12** time units.
- Completion times: $C_{P1} = 4$, $C_{P2} = 7$, $C_{P3} = 12$.
- Turnaround times equal the completion times (arrival = 0).
- Waiting times derived from $W_i = T_i - B_i$ or read from the chart.

🧱 Example 2: Varying Arrival Times & Idle Periods

Process	Arrival A_i	Burst B_i	Start	Completion C_i	Turnaround T_i	Waiting W_i
P1	3	5	3	8	$8 - 3 = 5$	$5 - 5 = 0$
P2	10	2	10	12	$12 - 10 = 2$	$2 - 2 = 0$

P3	15	4	15	19	$19 - 15 = 4$	$4 - 4 = 0$

Idle intervals:

- 0→3 : CPU idle (no process arrived).
- 8→10 : CPU idle (waiting for P2).
- 12→15 : CPU idle (waiting for P3).

Gantt chart (textual):

```
0---3---8---10---12---15---19
|Idle| P1 |Idle| P2 |Idle| P3 |
```

- Schedule length = **19** time units (including idle time).
- Each process starts exactly at its arrival time because no earlier process is pending.

📋 Quick Reference Table

Term	Meaning	How to Obtain
Arrival Time (A_i)	Moment a process enters the ready queue	Given in problem statement
Burst Time (B_i)	CPU time required by the process	Given in problem statement
Completion Time (C_i)	Time when the process finishes execution	End of its Gantt bar
Turnaround Time (T_i)	Total time in system	$C_i - A_i$
Waiting Time (W_i)	Time spent waiting in the ready queue	$T_i - B_i$
Schedule Length	Total elapsed time for all processes	End of last Gantt bar (including idle)
Tie-breaking rule	Preference when arrival times tie	Lower process ID runs first

📊 CPU Idle Time, Schedule Length & Percentage Idleness

> **Schedule Length** – The interval from the **arrival time of the first process** to the **completion time of the last process**.

> **CPU Idleness** – Periods when the CPU has no ready process to execute.

- **Total idle time** (including initial idle) = **8 units** in the first scenario.
- For **percentage CPU idleness** we consider only the idle time *within* the schedule length (initial idle is excluded).

Calculation

$$\text{Percentage Idle} = \frac{\text{Idle time within schedule}}{\text{Schedule length}} \times 100$$

$$\frac{5}{24} \times 100 \approx 20.8$$

🕒 Example 1 – FCFS with Gantt Chart & Idle Periods

Process	Arrival	Burst	Execution Window
P2	3	1	3→4
P1	5	2	5→7
P3	8	4	8→12

Idle intervals

- 0→3 (CPU waiting for first arrival)
- 4→5 (gap after P2 finishes)
- 7→8 (gap after P1 finishes)

Schedule length

$L = \text{Completion of last process} - \text{Arrival of first process} = 12 - 3 = 9 \text{ units}$

Idle time used for percentage

$\text{Idle within } L = (4-3) + (5-4) + (8-7) = 5 \text{ units}$

\text{CPU Idle %}= \frac{5}{9}\times100 \approx 55.6%

(The transcript's answer "4×23" reflects a different numeric example; the method above follows the same principle.)

⏱️ Dispatcher Latency (Δ) – Impact on Timing

> **Δ (Delta)** – The **dispatcher overhead** (load time) required to move a process from the ready queue to the CPU. It is *not* the save time unless explicitly stated.

Single-Process Example (Δ = 1)

Phase	Time Interval	Activity
Δ_1	0→1	Dispatcher loads PCB of P1
CPU_1	1→4	P1 runs on CPU (burst = 3)
I/O	4→9	P1 performs I/O (burst = 5)
Δ_2	9→10	Dispatcher reloads P1
CPU_2	10→12	P1 runs on CPU (burst = 2)
Term	12	P1 terminates

- **Total elapsed time** = 12 units (vs. 10 units without Δ).
- **Only the load latency** (Δ) is counted; the save latency is assumed negligible unless specified.

🔁 Process Life-Cycle with I/O Bursts

> **State Transition** – Ready → **Running** → (I/O) → Ready → **Running** → Terminated.
> The dispatcher is invoked **each time** a process moves from *Ready* to *Running*.

Timeline Illustration

1. **0-1** – Dispatcher loads process (Δ).
2. **1-4** – CPU execution (first CPU burst).
3. **4-9** – Process performs I/O (blocked).
4. **9-10** – Dispatcher loads process again (second Δ).
5. **10-12** – CPU execution (second CPU burst).

📋 Example 2 – Multiple Processes with Δ (FCFS)

Proc	Arrival	CPU_1	I/O	CPU_2	Δ (per load)
P1	0	3	7	–	1
P2	2	5	–	–	1
P3	5	1	4	–	1

Execution Sequence (FCFS)

1. **P1** loads (Δ = 1) → runs 3 units → I/O 7 units (4-11).
2. CPU idle **4-5** → **P2** loads (Δ) → runs 5 units **5-10**.
3. At **10**, CPU idle; **P3** arrived at 5, loads (Δ) → runs 1 unit **11-12** → I/O **12-16**.
4. After I/O, **P1** returns at **11**, loads (Δ) → runs remaining CPU (if any).

(The transcript describes a similar flow; exact burst lengths follow the given numbers.)

📐 Key Formulas & Concepts

- **Schedule Length**:

  $$L = C_{\text{last}} - A_{\text{first}}$$

- **CPU Idle Percentage** (excluding initial idle):
- **Dispatcher Overhead**:
  - Each transition **Ready → Running** adds Δ units.
  - If **save time** is ignored, only the load latency contributes to the schedule.
- **Gantt Chart Construction**:
  1. Order processes by **arrival time** (FCFS).
  2. Insert **Δ** before each CPU burst.
  3. Insert **I/O intervals** where the process is blocked.
  4. Mark **idle gaps** when no process is ready.

🗂️ Quick Reference Table

Item	Symbol	Meaning
	A_i	Arrival time of process *i*
	B_i	CPU burst time of process *i*
	I_i	I/O burst time of process *i*
	Δ	Dispatcher load latency
	L	Schedule length
	I_{total}	Total idle time *within* L
		Percentage CPU idleness

All calculations assume FCFS scheduling unless another algorithm is explicitly stated.

🕒 Scheduling Timeline & Loading Overhead

> **Loading (dispatch) time** – the time the OS spends moving a process from the ready queue to the CPU before actual execution begins.
> This interval is **not** counted as waiting time for the process.

- **P1**
  - Arrives at the ready queue at time 0.
  - First scheduled at time 0 → execution finishes at time 2.
  - Returns to the ready queue at time 11.
  - Loaded again (1 unit dispatch) → runs from **13 to 15** and terminates at 15.
- **P2**
  - Arrives at time 2.

    - First scheduled at time 4 (after 2 units waiting).
    - After its first CPU burst, it goes to I/O (10 to 12).
    - Returns to ready queue at time 12, loaded (1 unit) and runs from **15 to 19**.
- **P3**
    - Arrives at time 5.
    - First scheduled at time 11 (after 6 units waiting).
    - Loaded (1 unit) → runs from **12 to 14** and finishes at 14.

📊 Gantt Chart & Waiting Time Calculation

> **Waiting time** – total time a process spends in the ready queue **excluding** dispatch (loading) time.

Process	Arrival	CPU-ready intervals (excluding dispatch)	Waiting Time
P1	0	0 → 2, 13 → 15	1 (from 11 → 12)
P2	2	4 → 6, 15 → 19	2 (2 → 4) + 3 (12 → 15) = 5
P3	5	12 → 14	5 (5 → 10) + 3 (16 → 19) = 8* (see note)

*The second waiting interval for P3 (16 → 19) occurs after its I/O completes; the first interval (5 → 10) is **not** counted because it is the dispatch (transit) time from ready queue to CPU.

How to read the chart

1. Identify each time the process **enters** the ready queue.
2. Subtract any dispatch interval (the 1-unit loading time).
3. The remaining gap before the next CPU execution is the **waiting time**.

📈 Turnaround Time Formula Adjustments

Standard formula (without dispatch overhead):

$$\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time}$$

When dispatch overhead (Δ) and the number of times a process is scheduled (N) are considered:

$$\boxed{\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time} - N \times \Delta}$$

- For **P1**:
    - Turnaround = $15 - 0 = 15$
    - Burst = 7 (CPU) + 3 (I/O) = 10
    - $N = 2$ (scheduled twice)
    - $\Delta = 1$ (dispatch unit)

$$\text{WT}_{P1} = 15 - 10 - 2 \times 1 = 3; (\text{incorrect if dispatch is omitted}) \rightarrow \text{actual WT} = 1$$

Thus the adjusted formula yields the correct waiting time when dispatch is subtracted.

🔄 I/O Concurrency & Multiple Devices

> **I/O concurrency** – when two or more processes can perform I/O simultaneously because the system has multiple I/O devices.

- **P1** uses I/O from time 4 → 11.
- **P2** uses I/O from time 10 → 12.
- Overlap (10 → 11) is possible only if the system possesses **multiple I/O devices**.
- With a **single** I/O device, P2 would have waited until P1 released the device at 11, shifting its I/O to 11 → 13 and delaying subsequent CPU scheduling.

⚙️ CPU Idle vs. Transit Time

- The CPU is **never idle** during the entire schedule length; the apparent gaps are occupied by **dispatch (transit) time**.
- Dispatch time is counted as CPU activity, not as idle time.

📋 Summary of Key Metrics

Process	Arrival	Completion	Burst (CPU)	I/O Burst	Dispatches (N)	Waiting Time	Turnaround
P1	0	15	7	3	2	1	15
P2	2	19	4	2	2	5	17
P3	5	14	2	–	1	5*	9

*Waiting time for P3 excludes the first dispatch interval (5→6).

> **Takeaway:** Always compute waiting time directly from the Gantt chart, omitting dispatch (loading) periods, rather than relying solely on the textbook formula. This avoids the common mistake of over-counting waiting time when scheduling overhead is present.

📚 FCFS Scheduling Overview

> **First-Come-First-Served (FCFS)** – non-preemptive scheduling where the CPU selects the process that has been waiting the longest in the ready queue.

- **Dispatcher latency (Δ)** – the time the OS needs to perform a context switch before a process actually starts executing.
- A process **cannot** move directly from **Ready** to **Blocked**; it must first enter the **Running** state to issue a system call that initiates I/O.
- A process **terminates only from the Running state** after executing its final (exit) instruction.

🔄 Process State Transitions

> **Ready → Running** – occurs after the dispatcher finishes its latency.
> **Running → Blocked** – the process issues an I/O request (system call).
> **Blocked → Ready** – I/O completes; the process is placed back in the ready queue.
> **Running → Terminated** – the process executes its exit instruction.

State	Entry Condition	Exit Condition
Ready	Process created or I/O completed	Dispatcher selects it (Δ time)
Running	Dispatcher finishes latency	Completes CPU burst, requests I/O, or exits
Blocked	I/O request issued from Running state	I/O device finishes the request
Terminated	Exit instruction executed in Running state	None (process removed from system)

📆 Timeline & Gantt-Chart Details

⏳ P1 Execution Flow

1. $t = 3$ – P1 moves from Ready to Running (dispatcher Δ = 1).
2. $t = 4$ **to** 11 – CPU **burst time** of 7 units.
3. $t = 11$ **to** 12 – I/O of 1 unit (Blocked).
4. $t = 12$ – Returns to Ready.

5. Later, after other processes, $t = 48$ – P1 scheduled again, runs the **exit instruction** and **terminates at** $t = 49$.

⏳ P2 Execution Flow

1. $t = 11$ – P2 selected (lower PID than P3) after dispatcher Δ = 1.
2. $t = 12$ **to** 26 – CPU burst of 14 units.
3. $t = 26$ **to** 28 – I/O of 2 units (Blocked).
4. $t = 28$ – Returns to Ready.
5. After P1's termination, **P2 runs its single exit instruction** (no extra time) and terminates.

⏳ P3 Execution Flow

1. $t = 26$ – P3 scheduled (dispatcher Δ = 1).
2. $t = 27$ **to** 48 – CPU burst of 21 units.
3. $t = 48$ **to** 51 – I/O of 3 units (Blocked).
4. $t = 51$ – Returns to Ready, dispatcher Δ = 1, then executes its single exit instruction and terminates.

🕒 Overall CPU Utilization

- **Idle periods:** $t = 0$-2, $t = 50$-51 (no process ready).
- **Dispatch latency** adds 1 unit before each scheduled CPU burst (except when Δ is set to zero in the later example).
- **Total schedule length** (including all bursts, I/O, and idle time) ends when the last process (P3) terminates after $t = 51 + 1 = 52$.

⏱️ Dispatcher Latency Impact

- Each **context switch** incurs a fixed Δ = 1 time unit (unless specified as zero).
- **Waiting time** for a process **excludes** Δ; only the time spent in the Ready queue counts.
- Δ influences **overall schedule length** and **CPU idle intervals** but not the per-process waiting calculations.

🛠️ Tips for Solving FCFS with I/O

1. **Draw the Ready Queue (RQ) timeline** – mark arrivals, completions of I/O, and dispatcher actions.
2. **Start the Gantt chart at** $t = 0$ – ensures idle periods are visible.
3. **Follow the transition diagram** strictly (Ready → Running → Blocked → Ready).
4. **Do not add dispatcher latency to waiting time** – only to the total schedule length.
5. **Label the time when a process re-enters the Ready queue** after I/O; otherwise scheduling order can be mis-interpreted.

➖ Example: Zero Dispatcher Latency

When Δ = 0, the same three processes behave as follows:

Process	Arrival (Ready)	System-call (negligible)	I/O duration	Return to Ready	CPU burst	Post-burst I/O	Termination
P1	$t = 0$	–	2 units → $t = 2$	$t = 2$	7 units → $t = 9$	1 unit → $t = 10$	Exit at $t = 10$
P2	$t = 0$	–	4 units → $t = 4$	$t = 4$	– (not reached before end of example)	–	–
P3	$t = 0$	–	6 units → $t = 6$	$t = 6$	–	–	–

- **CPU idle** from $t = 0$ to $t = 2$ (no process ready).
- At $t = 2$, P1 begins its 7-unit CPU burst, finishes at $t = 9$, then performs a 1-unit I/O and returns at $t = 10$.
- The schedule length and idle periods are computed without adding any dispatcher overhead.

These notes capture the essential FCFS behavior, state-transition rules, and step-by-step timing for the illustrated processes, providing a complete reference for solving similar scheduling problems.

⏱️ Scheduling Timeline Example

🔷 Process Order and Time Allocation

- **Time 9** – Ready Queue (RQ) contains **P2** and **P3**.
  - **P2** arrived earlier ⇒ scheduled for **14 units** → **9 → 23**.
- **Time 23** – **P2** begins I/O (**23 → 25**) and will re-enter RQ at **25**.
  - CPU now has **P3** and **P1**; **P3** arrived earlier ⇒ scheduled for **21 units** → **23 → 44**.
- **Time 44** – **P3** starts I/O (**44 → 47**) and will return at **47**.
  - RQ holds **P1** and **P2**.
  - **P1** executes its **exit instruction** (negligible time) → **44 → 44** and terminates.
  - **P2** then executes a single instruction and terminates (also negligible).
- **Time 44 → 47** – No process in RQ ⇒ CPU idle (**2 units**).
- **Time 47** – **P3** returns from I/O, executes its exit instruction (negligible) and terminates.

📋 Timeline Summary

Interval	Running Process	Action	Duration
9 – 23	P2	CPU burst	14
23 – 25	P2	I/O	2
23 – 44	P3	CPU burst	21
44 – 47	–	CPU idle	2
44	P1	Exit (negligible)	0
44	P2	Single instruction (negligible)	0
47	P3	Exit (negligible)	0

- **Total schedule length** (completion of last process) = **47 units**.
- **Total CPU idle time** = **2 units** (explicit) + **3 units** (other idle periods mentioned) = **5 units**.

📊 Key Metrics

- **Schedule Length (makespan)**: $T_{\text{schedule}} = 47$ units (first scenario).
- **CPU Idle Time**: $T_{\text{idle}} = 5$ units.
- **Dispatcher Active Time**: $T_{\text{dispatch}} = 9$ units (as stated).

CPU Overhead

$$\text{CPU Overhead (\%)} = \frac{T_{\text{dispatch}}}{T_{\text{schedule}}}\times 100 = \frac{9}{52}\times 100 \approx 17.3\%$$

(The denominator 52 comes from the extended timeline 0 → 52 mentioned for the overhead calculation.)

CPU Efficiency

$$\text{CPU Efficiency (\%)} = 100 - \text{CPU Overhead (\%)} - \text{I/O Wait (\%)}$$

If I/O wait percentage is not explicitly given, it can be derived from total idle time:

$$\text{I/O Wait (\%)} = \frac{T_{\text{idle}}}{T_{\text{schedule}}}\times 100 = \frac{5}{52}\times 100 \approx 9.6\%$$

$$\text{CPU Efficiency (\%)} \approx 100 - 17.3 - 9.6 \approx 73.1\%$$

📚 Definitions

> **Ready Queue (RQ)** – The list of processes that are ready to use the CPU but are waiting for dispatcher allocation.

> **Dispatcher** – The OS component that performs context switch, loads the process state, and starts its execution. Its latency is the time taken to hand over the CPU to a selected process.

> **CPU Idle Time** – Periods when no process is ready to execute, causing the CPU to remain unused.

> **Schedule Length (Makespan)** – The total elapsed time from the arrival of the first process to the completion of the last process.

> **CPU Overhead** – The proportion of time the dispatcher is active relative to the total schedule length.

> **CPU Efficiency** – The proportion of time the CPU spends doing useful work (non-idle, non-overhead).

📝 Homework Example: FCFS Scheduling

Scenario Overview

- Three processes **P1**, **P2**, **P3** with given arrival times and CPU/IO bursts.
- **Dispatcher latency** = 2 units (except where noted as zero).
- All I/O operations are non-concurrent (only one process can perform I/O at a time).

Detailed Timeline

Time Interval	Event	Process	CPU/IO	Duration
0 – 2	Dispatch latency	–	CPU (dispatch)	2
2 – 5	CPU burst	P1	CPU	3
5 – 7	Dispatch latency	–	CPU (dispatch)	2
7 – 9	CPU burst	P2	CPU	2
9 – 12	Continuation	P2	CPU (remaining)	5
12 – 14	Dispatch latency	–	CPU (dispatch)	2
14 – 16	CPU burst	P1 (return from I/O)	CPU	2
16 – 20	CPU idle	–	–	4
20 – 22	Dispatch latency	–	CPU (dispatch)	2
22 – 23	CPU burst	P3	CPU	1
23 – 38	I/O burst (non-concurrent)	P3	I/O	15
38 – 40	Dispatch latency	–	CPU (dispatch)	2
40 – 46	CPU burst	P3 (post-I/O)	CPU	6
46 – …	All processes terminated	–	–	–

- **Total schedule length** = **46 units** (completion of P3).
- **CPU idle periods**: 4 units (16-20) + any idle during I/O (23-38) = **19 units**.

Calculations

- **Dispatcher active time** = 2 + 2 + 2 + 2 = **8 units**.
- **CPU Overhead** = $\frac{8}{46} \times 100 \approx 17.4$.
- **CPU Idle Percentage** = $\frac{19}{46} \times 100 \approx 41.3$.
- **CPU Efficiency** ≈ $100 - 17.4 - 41.3 \approx 41.3$.

⚙️ Additional Considerations: Single I/O Device

- When the system has **only one I/O device**, **I/O operations are serialized**: a second process requesting I/O must wait until the current I/O completes.
- This adds **extra waiting time** to the schedule, increasing total idle time and potentially raising the CPU overhead percentage.
- In the examples above, the I/O bursts for **P3** (15 units) caused a long CPU idle interval (23 → 38) because no other process was ready to use the CPU while the single I/O device was occupied.

CPU & IO Scheduling Scenario 📊

Key points

- Only **one I/O device** is available.
- A process that needs I/O must wait if the device is occupied.
- The CPU follows a **non-preemptive** schedule (process runs to completion of its CPU burst or I/O request).

> **Definition** – *Ready Queue (RQ)*: the set of processes that have arrived and are waiting for CPU allocation.

> **Definition** – *I/O Queue*: the set of processes waiting for the single I/O device.

Timeline of events

Time Interval	CPU Activity	I/O Device Activity	Comments
2 – 5	**P2** runs (3 units)	–	P2's first CPU burst
5 – 7	–	**P1** uses I/O (until t = 7)	P2 must wait for the I/O device
7 – 17	–	**P2** uses I/O (10 units)	P2 finally obtains the device
5 – 6	**P3** runs (1 unit)	–	P2 is in I/O, so P3 gets the CPU
6 – 7	–	–	CPU idle (no ready process)
7 – 10	**P1** runs (3 units)	–	P1 returns from I/O and executes its remaining CPU burst
10 – 17	–	–	CPU idle (no ready process)
17 – 19	**P2** runs (2 units)	–	P2 finishes its second CPU burst after I/O
19 – 20	–	–	CPU idle
20 – 25	**P3** runs (5 units)	–	P3 completes its remaining CPU work after I/O

Waiting time for a process that needs the I/O device includes the period it must stay in the I/O queue (e.g., P2 waited from t = 5 to t = 7).

Shortest Job First (SJF) Scheduling 🔧

> **Definition** – *Shortest Job First (SJF)*: a **non-preemptive** CPU-scheduling algorithm that always selects, from the **ready queue**, the process with the **smallest CPU burst time**.

- **Priority** is based solely on **burst time**; a shorter burst → higher priority.
- Processes are classified as **short processes** (short burst) and **long processes** (long burst).

- **Tie-breaking rule**: if two ready processes have equal burst times, schedule the one with the **lower process ID** (same rule used in FCFS).
- The algorithm compares burst times **only among processes that are already in the ready queue**; future arrivals are not considered until they enter the queue.

Example 1: SJF Walkthrough 📈

Process set

Process	Arrival Time	Burst Time
P1	0	2
P2	2	3
P3	3	1
P4	5	2
P5	8	3
P6	10	2

Scheduling steps

1. **t = 0-2** – Only **P1** is in the ready queue → runs for its full burst (2 units).
2. **t = 2-5** – **P2** is the sole ready process → runs for 3 units.
3. **t = 5-6** – Ready queue contains **P3** (burst = 1) and **P4** (burst = 2).
    - Choose **P3** (shorter burst) → runs 1 unit.
4. **t = 6-8** – Only **P4** remains → runs 2 units.
5. **t = 8-11** – **P5** arrives alone → runs 3 units.
6. **t = 11-13** – **P6** is the only ready process → runs 2 units.

Resulting execution order: **P1 → P2 → P3 → P4 → P5 → P6**.

Example 2: SJF with Tie-Breaking 📋

Process set

Process	Arrival Time	Burst Time
P1	10	1
P2	5	2
P3	7	4
P4	2	3
P5	5	2
P6	5	2

Scheduling steps

Time	Ready Queue (process ID : burst)	Selected Process	Reason
0-2	–	**Idle**	No arrivals yet
2-5	P4 : 3	P4	Only process present
5-7	P2 : 2, P5 : 2, P6 : 2	**P2**	Same burst → lower PID
7-9	P5 : 2, P6 : 2, P3 : 4	**P5**	Shortest burst (2) → lower PID

9-11	P6 : 2, P3 : 4	**P6**	Shortest burst (2)
11-12	P1 : 1, P3 : 4	**P1**	Shortest burst (1)
12-?	P3 : 4	**P3**	Last remaining process

Resulting order: Idle → P4 → P2 → P5 → P6 → P1 → P3.

Key observations:

- The CPU can be idle when the ready queue is empty (e.g., 0-2).
- Tie-breaking by **lower PID** ensures deterministic selection when burst times are equal.

General Rules for Non-Preemptive SJF 📝

- **Select** the ready process with the **minimum burst time**.
- **Do not preempt** a running process; it runs to completion of its CPU burst.
- **Consider only** processes **already in the ready queue**; ignore future arrivals until they appear.
- **Resolve ties** by **lower process ID** (same as FCFS).

These rules guarantee that, for a given set of arrivals and burst times, the SJF schedule is uniquely determined.

📅 Schedule Length & CPU Idleness

> **Schedule length**: the interval from the arrival time of the first process to the completion time of the last process.
> **CPU idleness**: the fraction of the schedule length during which the CPU performs no work.

- Formula: $L = C_{\text{last}} - A_{\text{first}}$
- In the example: first arrival at time 0, last completion at time 14 → $L = 14$
- CPU was busy the whole time → **CPU idleness** $= 0$

⚙️ Scheduling Algorithms Overview

🟢 Non-preemptive Shortest Job First (SJF)

> **SJF** selects the ready process with the **smallest burst time**; if several have the same burst, the one with the **lower process ID** is chosen.

- **Selection rule**: compare burst times of all processes currently in the Ready Queue (RQ).
- **Tie-break**: lower PID.
- **Mode**: non-preemptive – the chosen process runs to completion.
- **Example Gantt chart**

Time	Running Process	Reason
0-5	P1 (burst 5)	Only P1 in RQ at time 0
5-7	P2 (burst 2)	After P1 finishes, P2 has the smallest burst among ready processes
7-9	P3 (burst 2)	Remaining process with smallest burst

- **Context switches**: 2 (P1→P2, P2→P3)

🔴 Preemptive Shortest Remaining Time First (SRTF)

> **SRTF** selects the ready process with the **smallest remaining burst time**; a running process is **preempted** when a newly arrived process has a **strictly shorter** burst time.
> Tie-break: lower PID.

- **Preemption rule**: if a new arrival's burst b_{new} satisfies $b_{\text{new}} < r_{\text{current}}$ (remaining time of the current process), preempt.
- **Mode**: preemptive.
- **Example walkthrough**

1. **Time 0-2** – P1 runs (burst 5).
2. **Time 2** – P2 arrives (burst 2). Since $2 < 3$ (remaining of P1), **preempt** P1.
3. **Time 2-4** – P2 runs to completion.
4. **Time 4-7** – P1 resumes for its remaining 3 units.
5. If a new process had the **same** burst as the remaining time (e.g., both 3), **no preemption** occurs.

- **Context switches** in this scenario: 3 (P1→P2, P2→P1, P1→P3)

📊 Comparison: SJF vs. SRTF

Feature	SJF (non-preemptive)	SRTF (preemptive)
Preemptive?	No	Yes
Selection criterion	Smallest **burst time** among ready processes	Smallest **remaining burst time**
Tie-break	Lower **process ID**	Lower **process ID**
Preemption condition	N/A	New arrival with **strictly shorter** burst than current remaining time
Context switches (example)	2	3
CPU idleness (example)	0 %	0 %
Typical use	Simple batch jobs	Interactive/real-time where shorter jobs should finish quickly

---

🛠️ Roadmap for Making Scheduling Decisions

1. **Inspect the Ready Queue (RQ)** at the current time.
2. **Identify** the process(es) with the **minimum burst (or remaining) time**.
3. If multiple candidates, **choose the lower PID**.
4. For **SRTF**, when a new process arrives:
   - Compare its burst b_{new} with the **remaining** time r_{current} of the running process.
   - **Preempt** only if $b_{\text{new}} < r_{\text{current}}$.
5. **Schedule** the selected process until it completes (SJF) or until a preemptive event occurs (SRTF).

> **Key takeaway**:
>
> - **SJF** is simple and minimizes average waiting time but can cause longer jobs to wait.
> - **SRTF** further reduces waiting for short jobs by allowing preemption, at the cost of more context switches.
> - The decisive factor for preemption is the **strictly shorter** burst time of an arriving process.

🧩 SRTF (Shortest Remaining Time First) Scheduling Example

> **Definition:** *SRTF is a preemptive scheduling algorithm that always runs the process with the smallest remaining CPU burst time. If a newly arriving process has a shorter remaining time than the currently running one, a preemption occurs.*

📋 Process Arrival & Burst Data

Process	Arrival Time	Original Burst	Remarks
P2	2	– (longer than P6)	Arrives together with P6
P6	2	>7 (remaining 7 at t=3)	Shorter burst than P2, scheduled first

P1	3	6	Preempts P6 because 6 < 7
P5	4	< remaining P1	Preempts P1
P3	5	< remaining P5	Preempts P5
(new proc at 6)	6	= remaining P3	No preemption (equal)
P4	7	1	Scheduled after P3 completes

The exact original burst values for P2, P3, P5, and P6 are not provided; only relative ordering is used for scheduling decisions.

⏱️ Gantt-Chart Construction (Key Intervals)

1. **t = 0 → 2** – CPU idle (no process has arrived).
2. **t = 2 → 3** – Run **P6** (shortest burst among {P2, P6}).
3. **t = 3 → 4** – **P1** preempts P6 (burst = 6 < remaining 7).
4. **t = 4 → 5** – **P5** preempts P1 (burst < remaining P1).
5. **t = 5 → 6** – **P3** preempts P5 (burst < remaining P5).
6. **t = 6 → 7** – No preemption; **P3** continues (new arrival's burst equals remaining).
7. **t = 7 → 8** – All processes have arrived; schedule the smallest remaining burst → **P4** (burst = 1).
8. **t = 8 → …** – With no further arrivals, SRTF behaves like non-preemptive SJF:
   - Run **P5** (shortest remaining).
   - Run **P1**.
   - Run **P6**.
   - Finally run **P2** (largest burst).

📊 Completion Order & Times (derived from narrative)

Order	Process	Completion Time
1	P4	8 (1 unit)
2	P5	8 + (remaining time of P5)
3	P1	…
4	P6	…
5	P2	…

The transcript gives the final timeline: P5 finishes, then P1 runs from 11 → 16, P6 runs 16 → 23, and P2 runs 23 → 33.

📈 Performance Metric Mentioned

- The homework's answer for a related metric is **8.25** (likely the average waiting time for the above sequence).

📚 Shortest Job First (SJF) & SRTF – Theory

> **Definition:** *SJF (non-preemptive) and SRTF (preemptive) are optimal algorithms for minimizing average waiting time when burst times are known in advance.*

✅ Advantages

- **Higher throughput** – more jobs finish in a given interval.
- **Minimized average waiting time**.
- **Reduced average turnaround time**.

⚠️ Disadvantages (Starvation)

- Long-running processes may wait indefinitely if shorter jobs keep arriving.

🛠️ Implementation Issue

- Both algorithms require **prior knowledge of each process's CPU burst**.

- In practice, exact burst times are unknown, making direct implementation impossible.
- They are used as **theoretical benchmarks**; real schedulers approximate them (e.g., using burst-time prediction).

🧱 Example: Minimum Average Waiting Time with Non-Preemptive SJF

Scenario: Three processes arrive at time 0 with bursts **16 ms, 20 ms, 10 ms**. Scheduler knows burst lengths.

Steps

1. Order by increasing burst: 10 ms, 16 ms, 20 ms.
2. Compute waiting times:
   - P_1 (10 ms): **0** ms
   - P_2 (16 ms): **10** ms (waits for P_1)
   - P_3 (20 ms): **10 + 16 = 26** ms (waits for $P_1 + P_2$)
3. **Average waiting time** = ((0 + 10 + 26) / 3 = 12) ms.

> **Result:** Minimum achievable average waiting time = **12 ms**.

📐 Problem: Determining Unknown Burst (Z) from Total Waiting Time

Given:

- Four processes (P1, P2, P3, P4).
- Arrival times and bursts are known **except** for (P4) whose burst = (Z).
- Required total waiting time for all processes = **4** time units.

Outline of Reasoning (as described)

1. **Time 0 → 1:** Only (P1) is in the ready queue → (P1) runs.
2. **t = 1:** New process arrives with burst **shorter** than remaining of (P1) → preempt (P1); (P2) runs for 1 unit.
3. **t = 2:** Only (P1) remains (others arrive later) → (P1) resumes.

Waiting-time contributions

Process	Waited While	Wait Time
P1	During (P2)'s 1-unit run	**1**
P2	None (starts immediately)	**0**
P3 & P4	Both must together wait **3** units (to reach total 4)	**3** total

- (P3) arrives at **t = 3**, already waited **1** unit before being scheduled at **t = 4**.
- Therefore (P3) + (P4) need **2** additional waiting units collectively.

Scheduling Cases

- **Case A:** Schedule (P3) before (P4).
  - If (P3) runs for its known burst (7 units), (P4) would wait **≥ 3** units → exceeds total waiting budget.
  - **Impossible**.
- **Case B:** Schedule (P4) before (P3).
  - (P4) runs first for an unknown duration (Z).
  - (P3) then waits exactly the remaining needed waiting time (2 – (whatever (P4) already waited)).

> **Conclusion from transcript:** The feasible schedule is **(P4) first, then (P3)**; the exact value of (Z) must satisfy the total waiting-time constraint of 4 units. (The transcript stops before solving for (Z).)

📊 Comparison: SJF vs. SRTF

Feature	SJF (Non-preemptive)	SRTF (Preemptive)
Preemption	No	Yes, when a shorter remaining time arrives
Behavior when arrivals stop	Remains unchanged	Becomes equivalent to SJF
Optimality	Minimizes average waiting time (given exact bursts)	Same optimality, with added responsiveness
Starvation risk	High for long jobs	Also high, but can be mitigated with aging
Implementation difficulty	Requires burst prediction	Same requirement, plus overhead for context switches

---

📌 Key Takeaways

- **SRTF** continuously selects the process with the smallest *remaining* burst; it devolves to **SJF** once no newer shorter jobs appear.
- Both algorithms are **theoretically optimal** for average waiting time but **impractical without burst-time prediction**.
- **Starvation** is the main drawback; designers often add aging or minimum-burst guarantees to mitigate it.
- For a set of known bursts, **order them from shortest to longest** to achieve the minimum average waiting time (e.g., 12 ms for bursts 10, 16, 20).
- When solving waiting-time constraints with an unknown burst, enumerate possible schedules and apply the total-waiting-time condition to narrow down feasible values.

📅 Scheduling Example & Waiting Times

> **Waiting time** – the total time a process spends in the ready queue before it gets the CPU.

- **Process P4** arrives at time 4, is scheduled immediately → waiting time = 0.
- **Process P1**: initial waiting 0, then waits 1 unit before its next CPU slice → total waiting $= 1$.
- **Process P2** arrives at time 1 and is scheduled at time 1 → waiting time $= 0$.
- **Process P3** arrives at time 3, first scheduled at time 6 → waits 3 units; after its first slice it waits another 1 unit → total waiting $= 4$.

Process	Arrival Time	First Start	Total Waiting
P1	0	0	1
P2	1	1	0
P3	3	6	4
P4	4	4	0

The schedule chosen (P4 first because it has the smallest "first time") demonstrates that **SJF (Shortest Job First)** can be optimal, but its practical use is limited because actual burst times are unknown beforehand.

⚙️ CPU Burst Prediction Overview

> Predicting CPU burst lengths is essential for implementing SJF-like algorithms.

Two major categories:

1. **Static prediction** – uses information known *before* execution.
2. **Dynamic prediction** – updates estimates while the process is running.

📏 Static Prediction Methods

📐 Size-Based Estimation

- Assumes burst time correlates with program size.
- Example:
  - Old process: size $101\ \text{KB}$, CPU time $20\ \text{s}$.
  - New process: size $100\ \text{KB}$ → predicted CPU time ≈ $20\ \text{s}$.

📁 Type-Based Estimation

Process Type	Average Burst (s)
OS	5
Interactive	10
Foreground	15
Background	30

- A newly arrived **background** process would be predicted to run about $30\ \text{s}$.

🔄 Dynamic Prediction – Exponential Averaging (Aging Algorithm)

> **Exponential averaging** blends the most recent actual burst with the previous prediction.

The recurrence relation:

$$T_{n+1} = \alpha, T_n + (1 - \alpha), t_n$$

where

- T_{n+1} – predicted burst for the next CPU request.
- T_n – previous prediction.
- t_n – actual burst just completed.
- $\alpha \in [0, 1]$ – weight given to the previous prediction (larger α ⇒ smoother, slower to adapt).

🔁 Solving by Back-Substitution

1. Start from the recurrence and substitute T_n repeatedly until reaching the known **initial guess** T_1.
2. The expanded form after k substitutions:

$$T_{n+1} = \alpha^n T_1 + \sum_{i=1}^{n} \alpha^{n-i}(1 - \alpha)t_i$$

3. With T_1 (initial estimate) and α known, any future T_{n+1} can be computed.

📊 Example Calculation Using the Aging Algorithm

Given:

- $\alpha = 0.5$ (example value).
- Initial guess $T_1 = 10\,\text{ms}$.
- Actual completed bursts: $t_1 = 12, t_2 = 8, t_3 = 15, t_4 = 9$ (ms).

Goal: Predict T_5 (the 5th burst).

Step-by-step:

1. Compute T_2:

$$T_2 = \alpha T_1 + (1 - \alpha)t_1 = 0.5 \times 10 + 0.5 \times 12 = 5 + 6 = 11$$

2. Compute T_3:

   $T_3 = 0.5 \times T_2 + 0.5 \times t_2 = 0.5 \times 11 + 0.5 \times 8 = 5.5 + 4 = 9.5$

3. Compute T_4:

   $T_4 = 0.5 \times T_3 + 0.5 \times t_3 = 0.5 \times 9.5 + 0.5 \times 15 = 4.75 + 7.5 = 12.25$

4. Compute T_5:

   $T_5 = 0.5 \times T_4 + 0.5 \times t_4 = 0.5 \times 12.25 + 0.5 \times 9 = 6.125 + 4.5 = 10.625$

Result: The predicted 5th CPU burst $T_5 \approx 10.6\ \text{ms}$.

The same procedure applies for any α, T_1, and sequence of actual bursts.

🧩 Key Takeaways

- **SJF** needs burst times; without them, prediction techniques fill the gap.
- **Static methods** rely on readily available attributes (size, process class).
- **Dynamic exponential averaging** provides a continuously updated estimate using the formula $T_{n+1} = \alpha T_n + (1 - \alpha) t_n$.
- The **back-substitution** expansion shows how past actual bursts influence future predictions, weighted by α.

These concepts together enable operating systems to approximate optimal scheduling despite unknown future CPU demands.

📊 HRRN (Highest Response Ratio Next) Algorithm

📈 Response Ratio

> **Response Ratio** = $\dfrac{W + S}{S}$

- W = **Waiting Time** (time a process has waited in the ready queue)
- S = **Burst Time** (also called **Service Time**)

Interpretation

- Larger W → higher ratio → higher priority.
- Smaller S → higher ratio → shorter jobs are favored.
- The algorithm selects the **process with the highest response ratio** at each scheduling point.

⚙️ Scheduling Mode & Tie-Breaker

- **Non-preemptive**: once a process starts, it runs to completion.
- **Tie-breaker**: if two processes have identical response ratios, the **lower Process ID** is chosen.

🗓️ HRRN Example (Numerical Illustration)

Process	Arrival Time	Burst (S)	Waiting Time (W) at Decision	Response Ratio $\frac{W+S}{S}$
P1	0	3	0	$\frac{0+3}{3} = 1$
P2	2	6	– (only P1 in queue)	–
P3	4	4	5 (at time 9)	$\frac{5+4}{4} = 2.25$
P4	6	5	3 (at time 9)	$\frac{3+5}{5} = 1.6$
P5	8	2	1 (at time 9)	$\frac{1+2}{2} = 1.5$

Scheduling Steps

1. **Time 0–3**: Only P1 available → runs (non-preemptive).
2. **Time 3–9**: P2 is the sole ready process → runs.
3. **Time 9**: Ready queue contains P3, P4, P5. Compute ratios (as above).
   - Highest ratio: **P3 (2.25)** → runs 4 units (9-13).
4. **Time 13**: Queue now P4 and P5.
   - P4: $\frac{7+5}{5} = 2.4$ (waited 7 units).
   - P5: $\frac{5+2}{2} = 3.5$ (waited 5 units).
   - Highest ratio: **P5 (3.5)** → runs 2 units (13-15).
5. **Time 15–20**: Remaining P4 runs to completion.

Result – Longer-waiting processes obtain higher priority, eliminating starvation seen in pure SJF/SGF.

🔀 Comparison: SGF (Shortest-Job-First) vs. HRRN

Aspect	SGF (SJF)	HRRN
Priority Basis	Shortest **burst time** only	Highest **response ratio** (balances waiting & burst)
Starvation Risk	High for long jobs	Mitigated – long-waiting jobs eventually outrank short ones
Preemptive?	Often non-preemptive (but can be preemptive SRTF)	Non-preemptive (as described)
Tie-breaker	Lower Process ID	Lower Process ID

⏳ Longest Remaining Time First (LRTF) Algorithm

Core Idea

- Select the **process with the largest remaining burst time** among those in the ready queue.
- **Preemptive**: the running process can be interrupted when a newly arriving process has a remaining burst time **≥** the current remaining time.

Tie-breaker

- If multiple processes share the same remaining time, choose the **lowest Process ID**.

Scheduling Logic (Illustrated)

1. At **time 0**, three processes are ready. The one with the **largest burst** (P3, $S = 8$) starts.
2. The running process continues **until** a newly arriving process has a **remaining burst time** that is **equal to or greater** than the current remaining time.
   - Example: after P3 has executed 4 units, its remaining time = 4.
   - At this moment, **P2** arrives with burst $S = 4$ (equal to remaining time).
   - **Preemption occurs**: P3 is placed back in the queue, and P2 takes the CPU.

Key Differences from SRTF

Feature	SRTF (Shortest Remaining Time First)	LRTF
Selection Criterion	**Shortest** remaining time	**Longest** remaining time
Preemptive Condition	New job with **shorter** remaining time	New job with **equal or longer** remaining time
Typical Use	Favor short jobs, reduce average waiting	Favor long jobs, useful for specific real-time policies

---

📌 Important Formulas & Rules Recap

- **Response Ratio**: $RR = \frac{W + S}{S}$
- **Higher** W → higher RR → higher scheduling priority.
- **Lower** S → higher RR → short jobs still get early service.
- **HRRN Mode**: Non-preemptive; **LRTF Mode**: Preemptive.
- **Tie-breaker (both algorithms)**: Choose the **process with the lowest ID**.

📊 Preemptive Scheduling in LTF (Longest-Time-First)

> **LTF** – A preemptive scheduling rule that selects the process whose **burst time** is **greater than or equal to** the remaining burst time of the currently running process.
> When two processes have equal burst times, the one with the **lower process ID** is chosen.

- **Time 4**:
  - Burst times of *P2* and *P3* are both **4** → equal.
  - *P3* is preempted (LTF preempts on equality) and *P2* is scheduled.
- **Execution pattern** (each run = 1 time unit unless preempted earlier):

Time interval	Running process	Reason for preemption / continuation
4-5	**P2**	After 1 unit, remaining time = 3; a ready-queue process has burst **4 > 3**, so **P2** is preempted.
5-6	**P3**	Same logic: after 1 unit remaining = 3; another ready process has burst **4 > 3** → preempt.
6-7	**P2**	Again preempted after 1 unit (remaining = 2) because a ready process has burst **4 > 2**.
7-8	**P3**	Preempted after 1 unit (remaining = 2) for the same reason.
8-9	**P1** (lowest ID among equal-burst processes)	Continues until its remaining burst becomes 0.

- **Pattern repeats** while the three processes have burst times **2-2-2**: they are scheduled **alternately**, always picking the lower ID when bursts are equal.

🔄 Example: Alternating Execution of P1, P2, P3 (Burst = 2)

> When all ready processes have identical remaining burst times, **LTF** selects the process with the **smallest ID**.

Step	Process scheduled	Remaining burst after execution
1	**P1** (run 1 unit)	P1 = 1
2	**P2** (run 1 unit)	P2 = 1
3	**P3** (run 1 unit)	P3 = 1
4	**P1** (run 1 unit) → finishes	P1 = 0
5	**P2** (run 1 unit) → finishes	P2 = 0
6	**P3** (run 1 unit) → finishes	P3 = 0

The scheduler keeps rotating among the processes, always breaking ties with the **lower process ID**.

⏱️ Average Completion Time – Processes A, B, C

Execution timeline (LTF)

Time	Running process	Reason for switch
0-3	C (burst = 3)	At *t = 3* a ready process has equal burst → preempt.
3-4	B (burst = 5)	After 1 unit, *C* still has larger remaining burst (4 > 3) → preempt.
4-5	C (run 1)	Equal remaining bursts → lower ID (C) runs.
5-6	A (burst = 6)	Larger burst present, preempt C.
...	...	... (continues until all finish)

- Completion times derived from the full trace:
  - A finishes at t = 17
  - B finishes at t = 18

> Average completion time = $\frac{17 + 18}{2} = 17.5$

🖥️ CPU Utilization Calculation

- Arrival rate: 10 processes / minute → $\lambda = \frac{10}{60}\ \text{proc/sec} = \frac{1}{6}\ \text{proc/sec}$ (one every 6 s).
- Service time per process: $S = 5\ \text{s}$.

Utilization U (useful time ÷ total time):

$U = \frac{\text{Number of arrivals} \times S}{\text{Observation interval}} = \frac{10 \times 5\ \text{s}}{60\ \text{s}} = \frac{50}{60} = \frac{5}{6} \approx 83.33$

📋 Shortest-Job-First (SJF) – Non-Preemptive Scheduling with Unknown X

Given burst times

Process	Burst time
P5	1
P4	2
P3	5
P6	X (unknown, $5 < X < 7$)
P2	7
P1	9

> SJF selects the shortest available burst; ties are broken by lower process ID. The algorithm is non-preemptive (process runs to completion once started).

Scheduling order & completion times

Order	Process	Start	Completion
1	P5	0	1

2	P4	1	$1+2=3$
3	P3	3	$3+5=8$
4	P6	8	$8+X$
5	P2	$8+X$	$8+X+7=15+X$
6	P1	$15+X$	$15+X+9=24+X$

Solving for X using the given average completion time

$\text{Average} = \frac{1+3+8+(8+X)+(15+X)+(24+X)}{6} = 13$

$\frac{59+3X}{6} = 13; \Longrightarrow; 59+3X = 78; \Longrightarrow; 3X = 19; \Longrightarrow; X = \frac{19}{3} \approx 6.33$

- The solution satisfies the condition $5 < X < 7$.

📚 Key Takeaways

- **LTF** preempts when a ready process has a **burst time ≥** the remaining burst of the current process; ties are resolved by **lower ID**.
- With equal burst times, the scheduler **rotates** among processes, again using the **lowest ID** rule.
- **Average completion time** is the arithmetic mean of individual completion times.
- **CPU utilization** = (total service time) / (observation interval).
- In **non-preemptive SJF**, compute completion times sequentially, then solve any algebraic constraints (e.g., given average) to find unknown burst values.

📊 Priority Based Scheduling Overview

> **Priority based scheduling** selects the next process to run according to a numeric priority assigned to each process.

- Works like **SJF** or **SRTF**: can be **preemptive** or **non-preemptive**.
- The scheduler always prefers the **highest priority** process that is ready.

🔢 Priority Definition

> A **priority** is a number representing the importance of a process.

- Higher numeric value → higher priority (default convention).
- Some problems may define the opposite (e.g., “1 = highest”), so follow the given convention.

🔀 Tie-Breaker Rule

> If two ready processes share the same priority, the one with the **lower process ID** is chosen.

⚙️ Scheduling Rules & Conventions

Aspect	Detail
Priority range	Integer value assigned based on type, size, resources, etc.
Default convention	Larger number = higher priority (unless problem states otherwise).
Preemptive mode	Running process is interrupted when a newly arriving process has a higher priority.

Non-preemptive mode	Running process continues to completion, even if higher-priority processes arrive.
Mode switch	When no new processes arrive, a preemptive scheduler may behave like a non-preemptive one until the current process finishes.

⏱️ Non-Preemptive Example

Processes

Process	Arrival	Burst	Priority
P1	0	4	–
P2	4	3	–
P3	4	5	–

Schedule (Gantt chart)

1. **P1** runs from **0 → 4** (first arrival, no preemption).
2. At time 4, **P2** and **P3** are ready; **P3** has higher priority → runs **4 → 9**.
3. **P2** runs last from **9 → 12**.

🔄 Preemptive Example

Processes (same set as above)

Process	Arrival	Burst	Priority
P1	0	4	–
P2	1	1	Higher than P1
P3	2	5	Highest

Timeline

Time Interval	Running Process	Reason
0 → 1	**P1**	Only process available.
1 → 2	**P2**	Arrives with higher priority → preempts P1.
2 → 5	**P3**	Arrives with highest priority → preempts P2, runs to completion.
5 → 6	**P2**	Resumes (still highest among remaining).
6 → 9	**P1**	Last remaining process finishes.

- No new arrivals after time 3, so the scheduler stops preempting and lets each process finish.

📈 Larger Preemptive Scenario (6 Processes)

Process data

Proc	Arrival	Burst	Priority
P1	4	–	Low
P2	3-5?	–	Highest (until time 8)

P3	8	6	8 (max)
P4	2	–	3
P5	5	–	2nd highest
P6	3	5	5

Key events

1. **0-2** – CPU idle (no process).
2. **2-3** – **P4** (priority 3) runs.
3. **3** – **P6** (priority 5 > 3) arrives → preempts **P4**.
4. **4** – **P1** arrives (priority lower than **P6**) → ignored.
5. **5** – **P5** (priority > P6) arrives → preempts **P6**.
6. **5-7** – **P5** runs (until a higher-priority process arrives).
7. **7** – **P2** (still highest) runs until **P3** arrives at **8**.
8. **8-14** – **P3** (priority 8) runs to completion (burst 6).
9. After **P3** finishes, **P6** (priority 5) runs **14-19** (burst 5).
10. No further arrivals → scheduler switches to **non-preemptive**; remaining processes finish without interruption.
11. Final completion at **23** → **schedule length** = **23 – 2 = 21** time units (from first arrival to last finish).

🧩 Priority Types & Starvation

> **Static priority** – fixed for the lifetime of a process.
> **Dynamic priority** – can change while the process is in the system.

- **Starvation** occurs when low-priority processes never get CPU time because higher-priority processes keep arriving.
- **Aging** (dynamic priority adjustment) gradually **increases** the priority of waiting processes, preventing starvation.

📚 Example with Infinite Instances & Aging Concept

Process set

Process	Period (ms)	Priority (inverse of period)
P1	3	1 / 3
P2	7	1 / 7
P3	20	1 / 20

- Each process generates an infinite sequence of instances.
- First instances arrive at **1 ms**; subsequent instances follow their periods (e.g., P1 at 4 ms, 7 ms, …).

Partial schedule

Time	Running Process	Reason
0-1	**Idle**	No process yet.
1-2	**P1** (highest priority)	Runs to completion (burst 1).
2-4	**P2** (next highest)	Runs 2 units.
4-5	**P1** (new instance)	Higher priority than P3.
5-7	**P3** runs until **P1** arrives at 7 (preempted).	
7-8	**P1** (new instance)	Completes.
8-10	**P2** (new instance)	Completes.
…	…	Continues with same rule (higher priority preempts lower).

- When no newer higher-priority instance appears, the scheduler behaves non-preemptively until the current process finishes.

📅 Process Instance Scheduling Example

- **Instances**
  - 1st instance of **P1** runs, then **P2**, then **P1** again.
  - The **4th instance of P1** arrives at time **10** and is scheduled for **1 unit** of time.
  - **P3** has not run yet after the first three instances.
- **Arrival times of next instances**

Process	Next instance arrival
P1	13
P2	15
P3	11 (first time it can use CPU)

- At time **11**, only **P3** is in the CPU; it has already executed **2 units**.
- Remaining burst for P3 = **2 units** → it runs from **11 to 13**.

> **Completion time of the first instance of P3: 13 ms**

⏳ Aging Algorithm & Scheduling Disciplines

> **Aging algorithm:** a preemptive priority-based scheduler where priorities change dynamically.

- **Initial priority** of a newly arriving process = **0**.
- **Ready-queue (RQ) priority increase rate** = α (alpha).
- **Running process priority increase rate** = β (beta).

Priority Comparison

Condition	Effect on Scheduling
$\beta > \alpha$	Running process stays ahead → **FCFS** behavior (no preemption).
$\alpha > \beta$	Ready-queue processes outrank the running one → **preemptive**; newly arrived process can preempt.
New arrival priority > both RQ and running	The newest process runs first → **LIFO** (last-come-first-serve).

Example at Time 1

- Ready-queue priority = α (since it started at 0).
- Running process priority = β .

Compare α and β to decide preemption.

🔄 Round Robin Scheduling

> **Round Robin (RR):** a preemptive time-sharing algorithm that assigns each process a fixed **time quantum**.

- If a process finishes before the quantum expires → it leaves the CPU.
- If it does not finish → it is **preempted** and placed at the **end of the ready queue**.

Key Characteristics

- Improves **interactiveness** and **responsiveness**.
- **Preemptive** by design.
- Requires **maintaining the ready-queue status** and removing completed processes continuously.

📊 Round Robin Example (Quantum = 2)

Time	Running Process	Action	Ready Queue (order)
0-2	P1 (burst 4)	Preempt after 2 units → remaining 2	P2, P3, P1
2-4	P2 (burst 5)	Preempt after 2 units → remaining 3	P3, P1, P2
4-6	P3 (burst 3)	Preempt after 2 units → remaining 1	P1, P2, P3
6-8	P1 (remaining 2)	Completes → removed	P2, P3
8-10	P2 (remaining 3)	Runs 2 units → remaining 1	P3, P2
10-11	P3 (remaining 1)	Completes → removed	P2
11-12	P2 (remaining 1)	Completes → removed	–

- **Gantt chart** can be drawn from the table above.
- **Tip:** mark each completed process with a cross (✗) in the ready queue to avoid re-adding it.

📌 Additional Round Robin Scenario (Quantum = 2)

- **Five processes** with distinct arrival times and burst times.
- CPU remains **idle** from time 0 to time 3 because no process has arrived.
- At time 3, **P4** arrives (earliest among the five) and is scheduled.

Step	Time Interval	Process	Quantum Used	Remaining Burst
1	3-5	P4	2	(burst – 2)
…	…	…	…	…

- Continue rotating through the ready queue, applying the same preempt-and-re-enqueue rule.

📌 Practical Tips for Round Robin Problems

- **Always update the ready queue** after each quantum expiration.
- **Remove** any process whose remaining burst becomes **0**; do not re-insert it.
- Use a **cross mark (✗)** or similar indicator to track completed processes.
- When the CPU is idle, note the **gap** until the next arrival.

🔄 Round Robin Scheduling Overview

> *Round Robin (RR) is a preemptive CPU-scheduling algorithm that assigns each ready process a fixed time slice (time quantum). When the quantum expires, the running process is preempted and placed at the **end of the ready queue (RQ)**, allowing the next process to run.*

- **Fixed time quantum** determines how long a process may execute before being preempted.
- Preempted processes **join the back of the RQ after any newly arriving processes**.
- The CPU may be **idle** if no process is in the RQ.

📅 Example Execution Timeline

Time Interval	Running Process	Event / Arrival	Remaining Burst (after interval)
0 – 3	– (CPU idle)	P4 arrives at 3	P4 = 3 units (total)
3 – 5	P4	–	P4 remaining = 1 unit
4	–	P1 arrives	–
5	–	P5 arrives	–
5 – 7	P1	–	P1 remaining = 2 units
7 – 9	P5	–	P5 remaining = 2 units
9 – 10	P4 (last unit)	–	P4 finishes
10 – 12	P1	–	P1 remaining = 0 (finishes)
12 – 14	P3 (if present)	–	–
15	P2 arrives	–	P2 = ? (burst not specified)
…	…	…	…

*(The timeline reflects the pattern: new arrival → placed at front of RQ, then the **preempted** process is appended.*

📦 Ready Queue Management Rules

1. **Arrival Handling** – When one or more processes arrive at the same instant, they are **enqueued first** in the order of arrival.
2. **Preempted Process Placement** – After the time quantum expires, the preempted process is **added to the tail of the RQ** *after* any newly arrived processes.
3. **CPU Idle** – If the RQ is empty, the CPU remains idle until the next arrival.

Example:

- At time 5, both **P1** (arrived at 4) and **P5** (arrived at 5) are ready.
- **P1** is placed in the RQ first, then **P5**, followed by the preempted **P4** at the back.

📊 Performance Considerations of Round Robin

> *The performance of RR hinges on the chosen time quantum.*

- **Very Small Quantum**
  - Dispatcher overhead dominates → **efficiency approaches zero**.
  - Frequent context switches reduce useful CPU time.
- **Very Large Quantum**
  - Behaves like a **FCFS** (First-Come-First-Served) scheduler, increasing response time for short jobs.
- **Efficiency Formula**

  $$\text{Efficiency} = \frac{\text{Useful CPU Time}}{\text{Total Time (Useful + Dispatcher)}}$$
- **Goal** – Select a quantum that balances **low waiting/turnaround/response times** with **acceptable dispatcher overhead**.

⏱️ Response Time Concept

> *Response time is the interval from a process's arrival in the ready queue to the first time it gets the CPU.*

- It is a key metric alongside **turnaround time** and **waiting time**.
- In RR, response time is affected by the position of a newly arriving process in the RQ and the length of the current quantum.

📌 Key Terms

- **Time Quantum** – Fixed execution slice for each RR round.
- **Preempted Process** – A process whose quantum expired before completing its burst.
- **Ready Queue (RQ)** – FIFO queue holding processes ready to run.
- **Dispatcher** – OS component that performs context switches.
- **Efficiency** – Ratio of useful CPU time to total time (including dispatcher overhead).

🌀 Round Robin Scheduling Overview

> **Round Robin (RR)** – A pre-emptive CPU-scheduling algorithm where each ready process receives a fixed time slice called the **time quantum (Q)**. After using its quantum, the process is placed at the end of the ready queue.

- **Selection criteria** for RR:
  - **Arrival time** – when the process enters the ready queue.
  - **Burst time** – assumed **substantially larger** than Q in the presented scenario (to force pre-emption).
  - **Time quantum (Q)** – the fixed slice each process may run before being pre-empted.

⏱️ Effects of the Time Quantum (Q)

Quantum Size	Impact on Overhead	Responsiveness / Interactiveness	Behavioral Comparison
Small Q (≈ dispatch latency)	↑ many context switches → higher **dispatch overhead (s)**	↑ responsiveness (process gets CPU quickly) but ↓ interactiveness (CPU spends more time switching)	Approaches **pure time-sharing**
Large Q (≫ burst time)	↓ context switches → lower overhead	↓ responsiveness (process may wait long) but ↑ interactiveness for each burst	Behaves like **FCFS** when Q > max burst time

- **Overhead (dispatch latency)** – the time the CPU spends **switching** from one process to another; denoted by **s** seconds.
- When **Q > maximum burst time**, no pre-emption occurs; RR degenerates to **First-Come-First-Served (FCFS)**, making the system **least interactive**.

📊 Deriving Q for a Fixed Inter-Turn Interval T

Problem statement

- n processes arrive at time 0.
- Each has a **very large burst time** (≫ Q).
- CPU-scheduling overhead per switch = **s**.
- Find **Q** such that **each process gets the CPU exactly T seconds after its previous turn** (i.e., the interval between successive executions of the same process is **T**).

Gantt-Chart Reasoning

1. **First round** (each process runs once):
   - Start of P_1 = **s** (dispatch) → runs **Q** → finishes at **s + Q**.
   - Start of P_2 = **s + Q + s** = **2s + Q** → finishes at **2s + 2Q**.
   - …
   - Start of P_i = **i·s + (i-1)·Q** → finishes at **i·s + i·Q**.
   - P_n finishes its first turn at **n·s + n·Q**.
2. **Second round** (processes re-enter the queue in the same order):
   - After P_n finishes, the dispatcher incurs another **s** before P_1 can run again.
   - Start of P_1's second turn = **(n+1)·s + Q**.
3. **Interval between two successive turns of the same process**

$$\text{Interval} = \big[(n+1)s + Q\big] - \big[s + Q\big] = n, s$$

Solving for Q

To guarantee the interval equals the required T: $n, s = T;; \Longrightarrow;; s = \frac{T}{n}$

Since the interval is **independent of Q**, any Q that satisfies
$Q < (\text{minimum burst time})$ will meet the requirement, provided the overhead s is set to (T/n).

Inferences

- If Q is **exactly** the value derived above, each process gets the CPU **exactly after T seconds**.
- **Smaller Q** → each process runs for less time per turn; the interval **remains T**, but processes may finish earlier, giving the same process a **potentially earlier** next turn.
- **Larger Q** (still < burst) → each turn is longer, but the interval **still equals T** because it is governed by the fixed overhead **s**.

⚡ Overhead and Dispatch Latency

> **Dispatch latency (overhead, s)** – The fixed amount of time the CPU spends **switching** from one process to the next. It is incurred **before** each quantum execution.

📈 Responsiveness vs. Interactiveness

- **Responsiveness** – Time from a request's arrival to the **first** moment the process runs. Improves with **smaller Q**.
- **Interactiveness** – The degree to which the system appears to handle many tasks **simultaneously**. Improves with **larger Q** (fewer switches) but may hurt responsiveness.

⏳ Turnaround Time vs. Response Time

> **Turnaround time** – Total elapsed time from **arrival** to **completion** of a process.
> $\text{Turnaround time} = \text{completion time} - \text{arrival time}$

> **Response time** – Time from **arrival** (or request admission) to the **first** produced result (first CPU execution).
> $\text{Response time} = \text{first-response time} - \text{arrival time}$

Example

- Process P_1 arrives at time **4**.
- Waits **6** units → first scheduled at time **10**.
- Runs for **5** units (time 10 → 15), then continues later.
- **Response time** = **first-response time (10) – arrival time (4) = 6** units.
- **Turnaround time** (if it completes at time **30**) = **30 – 4 = 26** units.

These definitions help differentiate **how quickly a request is first serviced** (response) from **how long it takes to finish entirely** (turnaround).

🔪 Response Time Definition

> **Response time** – the interval from the moment a request (or process) is admitted to the system until the moment its first result is produced.

🔄 Round-Robin Scheduling Example

Scenario Overview

- **Processes:** 10 (all arrive at time 0)
- **Requests per process:** 20 identical requests
- **CPU burst per request:** 20 ms

- **I/O time per request:** 10 ms (after the CPU burst)
- **Time quantum (Q):** 20 ms
- **CPU-scheduling overhead:** 2 ms per dispatch

Calculating Response Times

Request	Admission Time	Completion Time	Response Time (Completion − Admission)
First request of Process 1 (P_{11})	0 ms	22 ms	**22 ms**
First request of Process 10 (P_{101})	0 ms	220 ms	**220 ms**
Second request of Process 1 (P_{12})	32 ms	242 ms	**210 ms**

Explanation of key steps

1. **Dispatch overhead** adds 2 ms before each CPU burst.
2. Since the quantum equals the CPU burst (20 ms), each request runs uninterrupted from **2 ms to 22 ms** for P_{11}.
3. After the CPU burst, the request spends 10 ms in I/O, returning at **32 ms** for its next request (P_{12}).
4. The last process (P_{10}) must wait for the first nine processes to complete all their 20 requests, leading to a completion at **220 ms**.

📚 Multi-Level Queue (MLQ) Scheduling

Why a single ready queue is insufficient

- **Mixed process types** (OS, user, interactive, background) cause **high searching & filtering time** when all are in one queue.
- Each process is forced to follow **only one scheduling algorithm**, limiting flexibility (e.g., OS processes cannot use FCFS while user processes use Round-Robin).

MLQ Structure

- **Multiple ready queues**, each dedicated to a class of processes (e.g., system, interactive, batch).
- Queues are **ordered by priority**: higher-priority queues are serviced before lower-priority ones.

Queue Level	Process Type	Typical Scheduling Algorithm
Q_0 (highest)	System / OS	FCFS or Priority
Q_1	Interactive / Foreground	Round-Robin
Q_2	Batch / Background	FCFS
Q_3 (lowest)	Low-priority jobs	FIFO

Scheduling Rules

1. **Serve the highest-priority non-empty queue**.
2. A process in a lower-priority queue runs **only when all higher queues are empty**.
3. No process can jump to a higher-priority queue; it stays within its assigned queue.

🔁 Multi-Level Feedback Queue (MLFQ) Scheduling

Core Idea

- **Feedback**: processes that consume more CPU time are **demoted** to lower-priority queues, receiving larger time quanta.
- Allows **dynamic adjustment** of priority based on actual CPU usage.

Example Problem

- **Process:** CPU-bound, total burst = 80 s (no I/O).
- **Initial quantum (Q_1):** 4 s.
- **Quantum increment per level:** +10 s (Q_2 = 14 s, Q_3 = 24 s, ...).

Execution Trace

Level	Quantum (s)	Remaining Burst before execution (s)	Executed (s)	Remaining after execution (s)
Q_1	4	80	4	76
Q_2	14	76	14	62
Q_3	24	62	24	38
Q_4	34	38	34	4
Q_5	44	4	4	0 (completed)

Results

- **Number of preemptions:** 4 (the process is preempted at the end of Q_1, Q_2, Q_3, and Q_4).
- **Termination queue:** Q_5 (the fifth level, where the remaining 4 s fit within the 44 s quantum).

Key Takeaways

- **Higher priority → smaller quantum**, promoting quick response for short jobs.
- **Long-running jobs** gradually receive larger quanta, reducing the overhead of frequent context switches.
- The **feedback mechanism** ensures fairness while still favoring interactive or short-duration processes.

📡 Interprocess Communication (IPC)

> **IPC** – *Interprocess Communication* is the mechanism that allows two or more processes to exchange data and synchronize their actions.

- **Shared media** is required for communication
  - Can be **hardware** (wires, cables, wireless) or **software** (protocols).
- Common IPC mechanisms

Mechanism	Description
Pipe	A file-like buffer in memory that connects a producer and a consumer.
Global variable	A memory location accessible to all processes in the same address space.
Message passing	Sending explicit messages through a defined protocol.

Intra-process communication (within a single process) also uses shared media, e.g., **function parameters** or **global variables**.

🧩 Types of Processes

> **Independent processes** – *Do not communicate or share resources.*
> **Coordinating processes** – *Communicate and cooperate by sharing resources.*

Process Type	Communication Needed?	Typical Use
Independent	No	Batch jobs, isolated tasks
Coordinating	Yes	Client-server, producer-consumer

🔄 Synchronization Overview

> **Synchronization** – *Agreement or protocol that ensures concurrent processes perform actions in a coordinated, conflict-free manner.*

Key ideas:

- Guarantees **orderly execution** of shared-resource accesses.
- Prevents unintended interference between concurrent activities.

⚠️ Problems Caused by Lack of Synchronization

1. **Inconsistency (Race Condition)** – Multiple processes update a common variable simultaneously, leading to incorrect results.
2. **Data loss** – Overwrites or missed updates erase information.
3. **Deadlock** – Processes wait indefinitely for resources held by each other.

Deadlock illustration: Four traffic flows converge on a ring; each side waits for the others to move, so none can proceed → infinite blocking.

📚 Illustrative Examples

🌳 Lawn-Sharing Example

- Two neighbors (H1 with a cat, H2 with a dog) share a lawn.
- **Agreement**: cat uses the lawn in the morning, dog in the evening.
- This schedule is a **synchronization protocol** that avoids conflict.

🥛 Milk-Buying Note Example

- Three hostel guests (P1, P2, P3) check a fridge for milk.
- Without coordination, all notice “no milk” and go out simultaneously, buying duplicate milk.
- **Solution**: The first guest leaves a note “I’m buying milk”; others wait, preventing redundant trips.

🚂 Train-Track Example

- Two trains (Process 1, Process 2) share a track segment.
- A protocol (e.g., a signal) decides which train may enter, avoiding collision.

🖨️ Printer Example

- Two processes request the printer at the same time.
- The OS grants the printer to one process and places the other in a **waiting** state until the printer becomes free.

🔀 Types of Synchronization

Type	Description	Typical Scenario
Competitive	Processes *compete* for exclusive access to a shared resource.	Amazon flash sale: multiple buyers vie for a single iPhone.
Cooperative	Processes *depend* on each other; execution of one influences the other.	Producer-Consumer problem: producer generates data, consumer processes it.

Competitive Example Detail

- Shared variable C.
- **P1**: C = C + 1 (increment).
- **P2**: C = C - 1 (decrement).

- Expected final value: original C.
- Without synchronization, possible outcomes are **C+1** or **C-1** due to race conditions.

Cooperative Example Mention

- The classic **Producer-Consumer** problem demonstrates coordinated use of a bounded buffer, requiring mechanisms such as semaphores or monitors (details to be covered later).

🛠️ OS Role in Enforcing Synchronization

1. **Resource request** → OS checks if the resource is free.
2. **Grant** → Process proceeds; resource marked as occupied.
3. **Deny & wait** → Process is placed in a waiting queue until the resource becomes available.

This control ensures **mutual exclusion**, prevents **deadlock**, and maintains **data integrity** across concurrent processes.

🤝 Competitive vs Cooperative Synchronization

⚔️ Competitive Synchronization

> **Competitive synchronization** occurs when multiple processes or users contend for the *same* limited resource, and the actions of one can overwrite or block the actions of another.

- **Typical examples**
  - Four people trying to book a single train seat. 🚂
  - Three people attempting to purchase the single iPhone on an Amazon sale. 📱
  - Two processes attempting to update the same shared variable simultaneously.
- **Consequences of missing synchronization**
  - **Inconsistency** – final data does not reflect the intended logical result.
  - **Data loss** – some updates may be overwritten and never recorded.

🤝 Cooperative Synchronization

> **Cooperative synchronization** involves processes whose actions *depend* on each other, requiring coordination to achieve a correct overall outcome.

- **Classic example:** *Producer-Consumer* problem.
  - **Producer** creates a data item and tries to place it into a bounded buffer.
  - **Consumer** removes a data item from the same buffer.
  - The buffer's capacity limits both actions:
    - If the buffer is **full**, the producer must wait.
    - If the buffer is **empty**, the consumer must wait.
- **Effect of lacking synchronization**
  - **Data lock** (deadlock) can occur when each process waits indefinitely for the other.
- **Note:** An IPC (Inter-Process Communication) application may involve **competition**, **cooperation**, or **both** simultaneously.

📦 Producer-Consumer Problem

- **Roles**
  - **Producer** → generates an item *X*, *Y*, … and places it in the buffer.
  - **Consumer** → removes an item from the buffer for use.
- **Bounded buffer mechanics**
  - A variable (e.g., nextEmptySlot) holds the index of the next free slot.
  - After each successful production, nextEmptySlot is incremented (e.g., from 2 to 3).

- **Synchronization conditions**

Buffer State	Producer Action	Consumer Action
Full (nextEmptySlot == capacity)	*Cannot produce* → must wait	–
Empty (nextEmptySlot == 0)	–	*Cannot consume* → must wait

- **Cooperation**: The producer's ability to place items depends on the consumer having removed previous items, and vice-versa.

🧩 Process Types & Synchronization Needs

> **Independent processes** do not communicate and therefore do **not** require synchronization.
> **Coordinating processes** communicate and **must** synchronize to avoid erroneous behavior.

Process Type	Communication	Synchronization Required?
Independent	None	No
Coordinating	Yes (shared resources, messages)	Yes

- **Lack of synchronization can cause**
  - **Inconsistency** – wrong results (e.g., race conditions).
  - **Data loss** – updates overwritten.
  - **Deadlock** – a situation where a process waits forever for an event that will never occur.

> **Deadlock:** *A state in which a process (or set of processes) is waiting for an event that never happens, typically because each is holding a resource the other needs.*

⚡ Race Condition

> **Race condition:** *A scenario where the system's behavior depends on the relative timing of independently executing processes, leading to nondeterministic results.*

Example Setup

- Shared integer variable C initially holds 5.
- **Process P_1**: increments C (C = C + 1).
- **Process P_2**: decrements C (C = C – 1).

Low-level instruction sequence (per process)

1. **Load** value of C into a register (R_1 or R_2).
2. **Modify** the register (+1 or -1).
3. **Store** the register back into C.

Interleaving & Outcomes

Execution Order	Steps Executed (by each process)	Final Stored Value of C	Expected Logical Result
P_1 first, preempted after step 2 → P_2 completes → P_1 resumes	P_1: Load (5) → Increment (6) *preempted* P_2: Load (5) → Decrement (4) → Store (4) P_1: Store (6)	**6**	**5** (increment then decrement should cancel)
P_2 first, preempted after step 2 → P_1 completes → P_2	P_2: Load (5) → Decrement (4) *preempted*	**4**	**5**

resumes	P_1: Load (5) → Increment (6) → Store (6) P_2: Store (4)		
Both preempt after step 2 → Both resume, store order decides	Both registers hold 6 and 4 respectively; whichever store executes last determines final value.	**6** *or* **4**	**5**

- **Key observation:** The *last* process to execute its store instruction **wins**.
- **Important rule:** When a preempted process returns to the CPU, it **resumes** where it left off; it does **not** restart from the beginning.

📚 Summary of Key Concepts

Concept	Definition (blockquote)	Typical Effect of Missing Synchronization
Competitive synchronization	*Multiple entities compete for a single resource; unsynchronized actions may overwrite each other.*	Inconsistency, data loss
Cooperative synchronization	*Entities cooperate, each action depending on the other; requires coordination.*	Data lock (deadlock)
Deadlock	*A waiting state where a process awaits an event that will never occur.*	System stall
Race condition	*Outcome depends on unpredictable timing of concurrent operations.*	Nondeterministic results (e.g., 4 or 6 instead of 5)

These notes capture the essential mechanisms, examples, and consequences of process synchronization, the producer-consumer paradigm, process classifications, and race conditions.

🤝 Correct Result Without Preemption

> **Definition:** *Preemption* is the interruption of a process by the scheduler, allowing another process to run before the first one finishes its current instruction sequence.

- When **no preemption** occurs, each process runs to completion of its three instructions before the other process starts.
- Example with shared variable **C** (initially 5):

Process	Steps (in order)	Effect on C
P1	1. R1 ← C (load 5) 2. R1 ← R1 + 1 (increment → 6) 3. C ← R1 (store 6)	**C = 6**
P2	1. R2 ← C (load 6) 2. R2 ← R2 − 1 (decrement → 5) 3. C ← R2 (store 5)	**C = 5**

- Whether **P1** runs first then **P2**, or **P2** runs first then **P1**, the final value of **C** is **5** – the *expected* correct result.

⚠️ Inconsistency With Preemption

- If a process is preempted **mid-sequence**, the intermediate value of **C** may be read or written by the other process, producing an incorrect final value.
- End users (e.g., a 5-year-old playing a game) care only about receiving the **correct result every time**, regardless of whether preemption occurs.

📦 Producer–Consumer Problem

> **Definition:** The *producer–consumer* problem involves two concurrent processes that share a **bounded buffer** of fixed size *n*. The **producer** inserts items, the **consumer** removes items.

🗃️ Buffer Structure

Symbol	Meaning
n	Size of the bounded buffer (e.g., 100)
buffer[0…n-1]	Array holding the items
in	Index of the *next empty slot* (where the producer will place the next item)
out	Index of the *first filled slot* (where the consumer will take the next item)
count	Number of items currently in the buffer

- **Circular indexing**: after index *n-1*, the next index wraps to 0 (index = (index + 1) mod n).

▶️ Producer actions

1. **Produce** an item → store in temporary variable itemP.
2. **Busy-wait** while count == n (buffer full).
3. Place the item: buffer[in] ← itemP.
4. Update in ← (in + 1) mod n.
5. Increment count ← count + 1.

> *Busy waiting* repeatedly tests the condition (count == n) until the buffer has space.

⏬ Consumer actions

1. **Busy-wait** while count == 0 (buffer empty).
2. Retrieve the item: itemC ← buffer[out].
3. Update out ← (out + 1) mod n.
4. Decrement count ← count − 1.

📊 Comparison of Producer vs. Consumer

Aspect	Producer	Consumer
When it proceeds	Only if count < n (buffer not full)	Only if count > 0 (buffer not empty)
Primary variable changed	buffer[in], in, count	buffer[out], out, count
Busy-wait condition	while (count == n)	while (count == 0)
Effect on count	count++	count--

🔄 Synchronization Nature

- The problem exhibits **cooperative synchronization** because the producer and consumer must *coordinate* their accesses to the shared buffer.
- It also involves **competitive synchronization**: each process *competes* for the limited slots (in vs. out) and must respect mutual exclusion on shared variables (buffer, count).

🛡️ Guaranteeing Correct Results

- To satisfy end-user expectations (correct result regardless of preemption), the system must enforce **atomicity** of the critical sections (the sequences that read-modify-write shared data).
- Common mechanisms (not detailed in the transcript) include **mutexes**, **semaphores**, or **lock-free algorithms** that ensure only one process can execute the critical section at a time, eliminating the inconsistency caused by preemption.

🚀 Producer-Consumer Race Condition

🔄 Example Scenario

- Shared variable **count** is accessed by a **producer** and a **consumer**.
- Both processes run in **user mode** and can be pre-empted at any instruction.
- Execution order:
    1. Producer executes four instructions, then is pre-empted.
    2. Consumer executes three instructions, then is pre-empted **just before updating count**.
- At this point both processes are **waiting to update count**, creating a race.

📈 Expected vs. Actual Results

Initial count	Producer action	Consumer action	Expected final count	Possible outcomes
5	increase → 6	decrease → 5	6 (producer) and 5 (consumer)	6 or 4 (depending on which update occurs last)

> **Definition**: A **race condition** occurs when the final result depends on the unpredictable order in which concurrent processes execute their critical-section code.

🧬 Necessary Conditions for a Synchronization Problem

📌 Critical Section

> **Definition**: The part of a program where **shared resources** (e.g., count, buffers) are accessed. Only one process should execute its critical section at a time.

⚔️ Race Condition

> **Definition**: A situation where two or more processes **compete** to update a shared variable, and the final value depends on the order of execution.

🦟 Analogy with Malaria

Malaria Spread Condition	Synchronization Analogy
Stagnant water (mosquito breeding ground)	**Critical section** (shared resource)
Unhygienic environment	**Race condition** (processes competing)
Mosquito (vector) – the *main culprit*	**Premature pre-emption** – the *main culprit*

⚡ Premature Preemption

> **Definition**: A process is **pre-empted** before it completes all instructions in its critical section, allowing another process to enter the critical section and potentially corrupt the shared data.

- Occurs when the scheduler interrupts a process **mid-critical-section**.
- Enables the competing process to perform its update first, leading to inconsistent results.

🛡️ Synchronization Mechanism: Security Guard Analogy

🚪 Entry Section

> **Definition**: Controls **entry** to the critical section, permitting **only one** process to enter at a time.

🚪 Exit Section

> **Definition**: Executes when a process leaves the critical section; it **notifies** other waiting processes that the critical section is now free.

🔄 Overall Flow

1. **Non-critical section** – process works with local variables.
2. **Entry section** – security guard checks if the critical section is free.
3. **Critical section** – process updates shared resource (count).
4. **Exit section** – security guard signals that the critical section is empty.
5. **Non-critical section** – process resumes its independent work.

📋 Assumption for Solving Synchronization Problems

> **Assumption**: Processes running in **user mode** can be pre-empted **anywhere**, after any instruction, and may be pre-empted **multiple times**.

📏 Requirements of the Critical Section Problem

Requirement	Meaning
1️⃣ **Mutual Exclusion**	**Only one** process may be in the critical section at any given time.
2️⃣ **Progress**	If no process is in the critical section and some processes wish to enter, **one** of the waiting processes must eventually be allowed to enter.

> **Mutual Exclusion**: No two processes are allowed to be inside the critical section simultaneously.
> **Progress**: The system must guarantee that waiting processes are not indefinitely blocked when the critical section becomes free.

🤝 Mutual Exclusion

> **Definition:** No two processes may be in the *critical section* at the same time.

- Guarantees data consistency.
- Violations lead to **inconsistency** (e.g., wrong count) and **loss of data**.

Example of Inconsistency

Process	Action	Initial Count	Result
Producer	$+1$	5	6
Consumer	-1	5	4
If both enter concurrently, the final count may be 4 *or* 6 *instead of the expected* 5 *.*			

🚀 Progress

> **Definition:** An *uninterested* process must never block the entry of an *interested* process.

- If a process does not need the critical section (e.g., P_3), it cannot prevent P_1 or P_2 from entering.
- Violation causes **indefinite postponement** (unfairness).

⏱️ Bounded Waiting

> **Definition:** There must be a finite bound on the number of times other processes can enter the critical section after a request before the requesting process is granted access.

- Prevents **starvation** (e.g., P_1 repeatedly entering while P_2 and P_3 wait forever).
- Guarantees that waiting time is limited.

📋 Synchronization Mechanism Requirements

Requirement	What It Ensures
Mutual Exclusion	No simultaneous critical-section entry
Progress	Uninterested processes do not block interested ones
Bounded Waiting	No process waits indefinitely

A mechanism satisfying **all three** is considered a **solution** (e.g., Peterson's algorithm, Dekker's algorithm). Mechanisms that satisfy only some are **not** full solutions.

🛠️ Types of Synchronization Mechanisms

1. Busy-Waiting (Spin-Lock)

- Processes repeatedly check a condition.
- Implemented in **software** (e.g., lock variable).

2. Non-Busy-Waiting (Blocking)

- Processes sleep until they are allowed to proceed.

Both categories can be realized in three layers:

Layer	Example	Nature
Software	Lock variable	User-mode
Hardware	Test-and-Set (TSL), Swap instruction	Special CPU instructions
OS-Based	Semaphores, Monitors (sleep/wake)	Kernel mode

📐 Lock Variable (Busy-Waiting)

- **Variable:** lock $\in \{0, 1\}$
  - 0 → critical section **free**
  - 1 → critical section **in use**

Entry Section

1. Check lock.

2. If lock == 0, set lock = 1 and enter critical section.
3. If lock == 1, keep waiting (spin).

Exit Section

- Set lock = 0 to release the critical section.

Analogy: A bathroom door that is locked from inside (1) or unlocked (0).

Limitations:

- Guarantees **progress** (interested process can eventually see 0).
- **Fails** to guarantee **mutual exclusion** and **bounded waiting** under certain interleavings.

⚖️ Comparison: Lock Variable vs. Strict Alternation

Property	Lock Variable	Strict Alternation
Mutual Exclusion	❌ (may fail)	✅
Progress	✅	❌ (can block)
Bounded Waiting	❌	✅
Type	Busy-waiting (software)	Busy-waiting (software)

Observation: Each mechanism supplies some desirable properties but not all. Combining their strengths leads to algorithms like **Peterson's** and **Dekker's**, which satisfy every requirement.

📌 Key Assumptions for Correct Synchronization

1. **Finite Critical-Section Time** – A process will leave the critical section after a bounded period; otherwise, others may starve.
2. **Entry Implies Interest** – A process reaches the entry section **only** if it intends to enter the critical section.
3. **Exit Completes Before Leaving** – A process is considered to have left the critical section only after executing its exit section.
4. **Preemption Anywhere** – The CPU may preempt a process at any point (entry, critical, or exit section) and any number of times.

These assumptions underpin the design and correctness proofs of synchronization algorithms.

🔒 Lock Variable & Critical Section

📖 Conceptual Analogy

> **Lock variable** – a flag that indicates whether the *critical section* (e.g., a bathroom or a room) is **free** (0) or **in use** (1).
> When the lock is 0, anyone may "enter" the room. When a process enters, it sets the lock to 1 (locking the room from inside). Upon leaving, it resets the lock to $\mathbb{0}$, unlocking the room for the next process.

🛠️ High-Level Implementation

- **Initial state**: lock = 0 (critical section free).
- **Entry section**:
  1. **Check** lock.
  2. If lock == 1, **busy-wait** until it becomes 0.
  3. When lock == 0, **set** lock = 1 and **enter** the critical section.
- **Exit section**: set lock = 0 to unlock.

```
/* Entry section */
while (lock == 1) {
    // busy-wait
```

```
}
lock = 1;   // acquire lock

/* Critical section */
// ... critical operations ...

/* Exit section */
lock = 0;   // release lock
```

⚙️ Low-Level (Assembly-like) Translation

Step	Action (pseudo-assembly)	Meaning
1	LOAD R, lock	Load current lock value into register R.
2	CMP R, 0	Compare R with 0.
3	JNZ B	If R != 0, jump to label **B** (busy-wait loop).
4	STORE lock, 1	Store 1 into lock → acquire the lock.
...	...	Critical-section code executes.
N	STORE lock, 0	Store 0 into lock → release the lock.

🔄 Execution Example with Two Processes (P1 & P2)

Time	Process	Action	Lock Value	Remarks
1	P1	Checks lock (0) → sets lock = 1	1	Enters critical section.
2	P2 (preempted)	Checks lock (1) → **busy-wait**	1	Stays in entry loop.
3	P1	Completes critical section, executes exit → lock = 0	0	Leaves critical section.
4	P2	Detects lock = 0, sets lock = 1	1	Enters critical section.

❓ Does the Simple Lock Guarantee Mutual Exclusion?

- **No.** If a **preemption** occurs **between** the two statements of the entry section (the check and the assignment), both processes can believe the lock is free and set it to 1 simultaneously.
- **Violation scenario**:
  1. P1 loads lock (0) into R1.
  2. **Preempt** before storing 1.
  3. P2 runs, loads lock (0) into R2, stores 1.
  4. **Resume** P1; it now also stores 1.
  5. Both P1 and P2 are inside the critical section → **mutual exclusion broken**.

📊 Summary of Lock Mechanics

Component	Symbol	Meaning	Typical Value
Lock variable	lock	Indicator of critical-section status	0 (free) / 1 (occupied)
Entry check	while (lock == 1)	Busy-wait loop	–
Acquire	lock = 1	Locks the section	–

Release	lock = 0	Unlocks the section	–
Mutual exclusion guarantee	–	**Fails** if preempted between check and assign	–

🧩 Key Takeaways

- The lock variable provides a **simple** way to enforce exclusive access, but **atomicity** of the check-and-set operation is essential.
- Without atomic hardware support (e.g., test-and-set), a naïve lock can be **preempted** and cause **simultaneous entry** to the critical section.
- Understanding the **busy-wait** behavior and the **preemption window** is crucial for designing correct synchronization primitives.

🔒 Lock Variable (Busy-Waiting) Solution

> **Definition**: A *lock* variable is a shared flag used to indicate whether the critical section is occupied (0 = free, 1 = occupied). Processes repeatedly read and write this flag to gain access.

- **Entry section**
  1. Set lock = 1.
  2. If preempted before the next statement, another process may also set lock = 1.
- **Critical section** – the code that must not be executed concurrently.
- **Exit section** – reset lock = 0.

❗ Violations Observed

- When **P1** is preempted after setting lock = 1, **P2** can also set lock = 1 and both enter the critical section simultaneously → **mutual exclusion is violated**.
- The scenario shows that the lock variable **does not guarantee mutual exclusion** under preemption.

📈 Progress & Bounded Waiting Conditions

> **Progress**: If no process is in the critical section and some processes wish to enter, one of the interested processes must eventually be allowed to enter.

- A *non-interested* process never enters the entry section, so it cannot block an interested one.
- In the lock-variable scheme, progress is **guaranteed** because an uninterested process never changes lock.

> **Bounded Waiting**: Once a process requests entry, there must be a limit on how many times other processes can enter before it does.

- The lock solution **fails bounded waiting**: a process (e.g., **P1**) can repeatedly re-enter the critical section while another (e.g., **P2**) waits indefinitely.

Example of Unbounded Waiting

1. **P1** finishes its critical section, sets lock = 0, and immediately re-enters the entry section.
2. **P2** remains waiting because lock becomes 1 again before it can acquire the lock.

Result: **P2** may starve → bounded waiting not satisfied.

⏱️ Busy-Waiting and CPU Waste

- The lock variable forces each process to **continuously poll** the flag (while (lock == 1) {}), consuming CPU cycles.

- **Alternative idea**: block the process when the critical section is occupied and wake it up when the lock is released (sleeping queue). This would eliminate the wasteful polling.

📂 Device-Sharing Example (Busy Variable)

> **Definition**: A *busy* variable works identically to the lock variable, using false (0) for free and true (1) for occupied.

```
// Pseudocode for a process sharing a device
repeat
    while (busy == true)   // busy-waiting
        ;                  // spin
    busy = true;           // claim the device
    // critical section: use the device
    busy = false;          // release the device
until (termination_condition);
```

True Statements About This Approach

Statement	Correct?	Reason
Provides a reasonable solution for mutual exclusion	**No**	Same violation as lock variable.
Consumes substantial CPU time (busy-waiting)	**Yes**	Processes repeatedly check busy.
Fails to guarantee mutual exclusion	**Yes**	Preemption can cause simultaneous entry.

Thus, options 2 and 3 are correct.

🔁 Strict Alternation (Two-Process Solution)

> **Definition**: *Strict alternation* forces two processes to take turns entering the critical section, using a shared variable turn that can be 0 or 1.

```
int turn = random(0,1);   // initial turn value

/* Process 0 */
while (true) {
    while (turn != 0) ;      // busy-wait
    // critical section
    turn = 1;                // give turn to Process 1
}

/* Process 1 */
while (true) {
    while (turn != 1) ;      // busy-wait
    // critical section
    turn = 0;                // give turn to Process 0
}
```

How It Works

1. turn = 0 → **Process 0** may enter; **Process 1** busy-waits.
2. After finishing, **Process 0** sets turn = 1.
3. **Process 1** now proceeds, while **Process 0** busy-waits.
4. The cycle repeats, guaranteeing **mutual exclusion** because only the process whose turn matches turn can enter.

Properties

Property	Satisfied?	Explanation
Mutual Exclusion	**Yes**	Only one process can satisfy turn == its_id.
Progress	**Yes**	The process whose turn it is will eventually enter.
Bounded Waiting	**Yes**	Each process gets the turn after at most one entry by the other.
CPU Efficiency	**No**	Still uses busy-waiting loops.

📊 Comparison: Lock Variable vs. Strict Alternation

Feature	Lock Variable	Strict Alternation
Process count	Multi-process	Exactly two
Mutual exclusion	*Not guaranteed* (preemption)	**Guaranteed**
Progress	Guaranteed	Guaranteed
Bounded waiting	*Not guaranteed* (starvation possible)	**Guaranteed**
CPU usage	Busy-waiting (high waste)	Busy-waiting (high waste)
Complexity	Simple flag	Simple turn variable, but limited to two processes

🛠️ Key Takeaways

- The **lock** (or **busy**) variable is a naïve busy-waiting mechanism that fails to ensure mutual exclusion and bounded waiting, though it does guarantee progress.
- **Strict alternation** improves correctness for two processes by enforcing a turn-taking scheme, yet it remains a busy-waiting solution and cannot scale beyond two processes.
- Both approaches suffer from **CPU cycle wastage**; replacing busy-waiting with **blocking** (sleeping queues) can mitigate this inefficiency.

🛡️ Mutual Exclusion & Turn Variable

> **Mutual Exclusion** – At most one process may be inside its critical section at any moment.

- The **turn** variable determines which process may enter the critical section.
- When turn = I the process *I* may execute its critical section; when turn = J the process *J* may do so.
- After a process finishes its critical section it **exits** by setting turn to the other process, enabling progress.

How the Turn Variable Operates (Scenario)

1. Two processes, **P_0** and **P_1**, are in the ready queue (RQ).
2. turn is initially set randomly to 0 or 1.
3. **P_0** is scheduled first, enters the critical section, then is pre-empted.
4. **P_1** runs while turn = 0 (still P_0's turn).
5. **P_1** repeatedly checks turn; because it is not its turn, it busy-waits.
6. When **P_0** later resumes, completes its critical section, and executes the **exit** step, it sets turn = 1.
7. **P_1** now sees turn = 1 and proceeds into the critical section without further waiting.

⏱️ Progress & Bounded Waiting

> **Progress** – If no process is in its critical section and some processes wish to enter, at least one of the interested processes will eventually be allowed to enter.

> **Bounded Waiting** – There exists a finite bound on the number of times other processes can enter their critical sections after a process has indicated its desire to enter.

> **Unbounded Waiting** – A process may be forced to wait indefinitely while others repeatedly enter the critical section.

- In the turn-based scheme, **bounded waiting** is achieved because after a process sets turn to the other, the other process must complete its critical section before the first can re-enter.
- **Unbounded waiting** would occur if a process kept re-entering the critical section (e.g., turn never changes), starving the other process.

🚦 Lock Variable vs. Turn Variable

Feature	Lock Variable	Turn Variable
Guarantees **Mutual Exclusion**	❌ (not guaranteed)	✅
Guarantees **Progress**	✅	❌ (progress can be blocked by uninterested process)
Guarantees **Bounded Waiting**	❌	✅ (strict alternation)
Number of Processes Supported	Multiprocess (any number)	Two-process only
Busy-waiting Impact	Consumes CPU cycles	Consumes CPU cycles (but bounded)

- An **uninterested** process (e.g., P_0 never enters its critical section) can block progress of the other process under the turn scheme because turn is never changed.

🔄 Strict Alternation

> **Strict Alternation** – Processes take turns in a fixed order (e.g., $P_0 \rightarrow P_1 \rightarrow P_0$ ...).

- Provides **mutual exclusion** and **bounded waiting** because each process must wait for its turn.
- **Fails to guarantee progress** when a process is uninterested; the other process cannot proceed because the turn never switches.

🧩 Peterson's Solution (Two-Process Algorithm)

> **Peterson's Solution** – A software protocol that combines a **flag** array and a **turn** variable to satisfy mutual exclusion, progress, and bounded waiting for two processes.

Key Variables

- flag[i] (i = 0 or 1): true if process *i* wants to enter the critical section, false otherwise.
- turn: indicates whose turn it is (0 or 1).

Algorithm Steps (for process *i*)

1. flag[i] ← true // announce interest
2. turn ← j // give priority to the other process *j*
3. **While** flag[j] = true **and** turn = j **do** // busy-wait

---## 🛡️ Mutual Exclusion & Turn Variable

> **Mutual Exclusion** – At most one process can be inside its critical section at any time.

- The **turn** variable indicates whose turn it is to enter the critical section.
  - $turn = I$ → process *I* may enter.
  - $turn = J$ → process *J* may enter.

- After leaving the critical section, a process sets turn to the other process, allowing progress.

Execution Scenario

1. Processes P_0 and P_1 are in the ready queue.
2. $turn$ is randomly initialized to 0 or 1.
3. P_0 is scheduled, enters the critical section, and is then pre-empted.
4. P_1 runs while $turn = 0$ (still P_0's turn). It repeatedly checks $turn$ (busy-wait).
5. When P_0 later resumes, finishes its critical section, and executes the **exit** step, it sets $turn = 1$.
6. P_1 now sees $turn = 1$ and proceeds into the critical section without further waiting.

⏱️ Progress & Bounded Waiting

> **Progress** – If no process is in its critical section and some processes wish to enter, at least one of the interested processes will eventually be allowed to enter.

> **Bounded Waiting** – There exists a fixed upper bound on the number of times other processes may enter their critical sections after a process has indicated its desire to enter.

> **Unbounded Waiting** – A process may be forced to wait indefinitely while others repeatedly enter the critical section.

- In the turn-based scheme **bounded waiting** is guaranteed because the process that finishes always changes turn, forcing the other process to run before it can re-enter.
- **Unbounded waiting** would occur if a process could re-enter repeatedly without changing turn, starving the other process.

🔀 Lock Variable vs. Turn Variable

Property	Lock Variable	Turn Variable
Guarantees **Mutual Exclusion**	❌ (not guaranteed)	✅
Guarantees **Progress**	✅ (interested process eventually proceeds)	❌ (an uninterested process can block progress)
Guarantees **Bounded Waiting**	❌ (no bound on repetitions)	✅ (strict alternation gives a bound)
Supported Process Count	Multiprocess (any number)	Two-process only
Busy-waiting effect	Consumes CPU cycles	Consumes CPU cycles (but bounded)

- An **uninterested** process (e.g., P_0 never enters its critical section) can prevent the other process from progressing because turn never changes.

🔄 Strict Alternation

> **Strict Alternation** – Processes take turns in a fixed order ($P_0 \rightarrow P_1 \rightarrow P_0$ ...).

- Provides **mutual exclusion** and **bounded waiting** due to the enforced order.
- **Fails to guarantee progress** when a process is uninterested; the other process cannot enter because the turn never switches.

🧩 Peterson's Solution (Two-Process Algorithm)

> **Peterson's Solution** – A software protocol that combines a **flag** array and a **turn** variable to satisfy mutual exclusion, progress, and bounded waiting for two processes.

Core Variables

Variable	Meaning
flag[i] (i = 0,1)	true → process *i* wants to enter; false → not interested
turn	Indicates which process has priority (0 or 1).

Algorithm (process *i*)

1. flag[i] ← true // announce interest
2. turn ← j // give priority to the other process *j*
3. **while** flag[j] = true **and** turn = j **do** // busy-wait
4. *Critical section*
5. flag[i] ← false // exit, relinquish interest

Guarantees

- **Mutual Exclusion** – Only one process can be in the critical section at a time.
- **Progress** – If a process is not interested, it does not block the other.
- **Bounded Waiting** – The turn variable enforces a strict alternation, limiting the number of entries of the other process.
- **Busy-waiting** – The algorithm uses busy-waiting but remains a purely software solution executable in user mode.
- **Scope** – Works for exactly two processes (e.g., P_0 and P_1 or any pair *I*, *J*).

🔢 Generalizing Turn to Arbitrary Processes *I* and *J*

- turn can hold any two identifiers, not just 0 and 1.
- Initialization: turn ← random(I, J) selects either *I* or *J*.
- If turn = I, process *I* may enter; otherwise process *J* may enter.
- The same busy-waiting check applies: a process waits while turn points to the other process.

❌ Incorrect Synchronization Solutions

Solution	Fails Which Property?
Lock only	Mutual exclusion and bounded waiting
Strict alternation	Progress (blocked by uninterested process)
Combination without proper handling	Any missing property (mutual exclusion, progress, or bounded waiting)

All three requirements—mutual exclusion, progress, and bounded waiting—must be satisfied simultaneously; otherwise the solution is considered incorrect.

📚 Homework Prompt

> Determine, for a system where turn is set to a random choice between processes *I* and *J*, whether the following hold:
>
> 1. **Mutual Exclusion** – Is it guaranteed?
> 2. **Progress** – Is it guaranteed?
> 3. **Bounded Waiting** – Is it guaranteed?

Use the concepts above (lock variable, turn variable, strict alternation, Peterson's solution) to justify each answer.

🛡️ Peterson's Solution Overview

📌 Key Concepts

- **Flag array** – *boolean array flag[i] where true means process i is interested in entering the critical section*

> The flag is set to **true** when a process wants to enter and reset to **false** after leaving.

- **turn variable** – *indicates which process has the right to enter next*

> The process that writes to turn last must wait, allowing the other process to proceed.

🔧 Algorithm Steps

1. **Entry Section** (process *i*):
   - Set flag[i] = true (express interest).
   - Set turn = i (claim turn).
   - **Wait** while flag[j] == true && turn == i (where *j* is the other process).
2. **Critical Section** – execute the protected code.
3. **Exit Section** (process *i*):
   - Set flag[i] = false (no longer interested).

📊 Variables and Initialization

Variable	Description	Initial Value
n	Number of processes	–
flag[0…n-1]	Interest flags for each process	false for all
turn	Holds the ID of the process that last set the turn	*undefined* (not preset)

🗂️ Pseudocode

```
void process(int i) {
    int j = 1 - i;              // the other process
    while (true) {
        // entry section
        flag[i] = true;
        turn = i;
        while (flag[j] && turn == i) { /* busy-wait */ }

        // critical section
        // ... protected code ...

        // exit section
        flag[i] = false;

        // remainder section
        // ... non-critical work ...
    }
}
```

⏱️ Execution Example (Processes P0 and P1)

Time	Action	flag[0]	flag[1]	turn	Who is waiting?
T_1	P0 enters entry section	true	false	0	–
T_2	P1 enters entry section	true	true	1	P1 sees flag[0]=true & turn=1 → **waits**
T_3	P0 resumes, checks condition: flag[1]=true but	true	true	1	–

	turn=1 (≠0) → **proceeds**				
…	P0 executes critical section, then exit	false	true	1	–
T_4	P1 re-evaluates while condition: flag[0]=false → **proceeds**	false	true	1	–

- The **last** process to write turn (P1) must wait, allowing the earlier writer (P0) to enter the critical section first.

✅ Guarantees Provided by Peterson's Solution

Property	How Peterson's algorithm ensures it
Mutual Exclusion	Only one process can have turn equal to its own ID while the other's flag is true; the waiting condition blocks the later writer.
Progress	If a process is not interested (flag = false), the other process proceeds without waiting.
Bounded Waiting	A process can be forced to wait at most **one** turn of the other process, because the turn variable alternates after each entry.

📈 Reasoning Behind the Guarantees

- **Mutual Exclusion** – The while-loop while (flag[j] && turn == i) blocks a process *i* only when the other process *j* is also interested **and** *i* was the last to set turn. Thus both cannot be inside the critical section simultaneously.
- **Progress** – When flag[j] is **false**, the condition fails regardless of turn, so the interested process proceeds immediately.
- **Bounded Waiting** – The turn variable can change at most once per entry attempt; therefore a process can be delayed only for the duration of the other process's critical section, guaranteeing a finite bound.

🔁 Re-entering the Critical Section

- After exiting, a process resets its flag to **false**.
- If it attempts to re-enter immediately, it must set flag[i] = true and turn = i again.
- Since the other process (if still interested) will have turn equal to its own ID, the re-entering process will wait, preventing consecutive entries by the same process.

These notes capture the essential mechanics, variable roles, execution flow, and correctness arguments of Peterson's solution for two-process mutual exclusion.

Peterson's Algorithm Overview 🛡️

Variables and Concepts

- **flag**i – Boolean array; true indicates process *i* wants to enter the critical section.
- **turn** – Indicates which process has priority to enter the critical section next.

> **Definition (Mutual Exclusion)** – If a process is executing in its critical section, no other process can be in its critical section simultaneously.

> **Definition (Progress)** – When no process is in the critical section and at least one process wishes to enter, one of the waiting processes must eventually be allowed to enter; an uninterested process must not block the others indefinitely.

> **Definition (Bounded Waiting)** – After a process requests entry to the critical section, there exists a limit on how many times other processes may enter before its request is granted.

Execution Steps (for process *i*)

1. flag[i] ← true – declare interest.

2. turn ← j – give priority to the other process *j*.
3. while flag[j] && turn = j : wait – busy-wait while the other process is interested **and** it has the turn.
4. **Critical Section** – exclusive access guaranteed.
5. flag[i] ← false – exit section, allowing the other process to proceed.

How Peterson Guarantees Mutual Exclusion 🔐

- When both processes set their flags to true, the last assignment to **turn** decides who must wait.
- Example with processes P_0 and P_1:
  - P_0 sets flag[0]=true, turn=1.
  - P_1 subsequently sets flag[1]=true, turn=0.
  - The while condition for P_0 becomes while flag[1] && turn=1, which is **false** because turn=0; P_0 proceeds.
  - P_1 sees while flag[0] && turn=0 true, so it waits.
- Thus only one process can be inside the critical section at a time.

Progress Property in Peterson's Algorithm 🚀

- If only one process sets its flag to true, the while condition evaluates to false (the other flag is false).
- The interested process enters the critical section without being blocked by an uninterested one.

> **Key Point:** An uninterested process never sets its flag to true; therefore it cannot hinder progress.

Bounded Waiting in Peterson's Algorithm ⏳

- The **turn** variable ensures that a process cannot repeatedly re-enter the critical section before the other waiting process gets a chance.
- After a process exits, turn holds the identifier of the other process, forcing the next entry attempt to give the other process priority.
- Consequently, the number of times a process may be bypassed is bounded by one.

Analysis of the Given “turn” Code Snippet 🔄

```
/* Randomly assign turn to either I or J */
turn = random(I, J);

while (turn != I) {
    // busy-wait
}
```

Mutual Exclusion Check

- The code only checks the value of **turn**; there is **no flag** indicating interest.
- If both processes set turn to the same value, they could both pass the while test simultaneously, violating mutual exclusion.

Progress Check

- If the non-interested process never changes **turn**, the interested process may loop forever (while (turn != I) never becomes false).
- Hence **progress is not guaranteed**.

Bounded Waiting Check

- The algorithm does not limit how many times a process can re-enter the critical section; a process may repeatedly set turn to its own identifier.
- Therefore **bounded waiting is not satisfied**.

Summary Table

Property	Peterson's Algorithm	Given “turn” Code

Mutual Exclusion	✅ Guaranteed	❌ Not guaranteed
Progress	✅ Guaranteed	❌ Not guaranteed
Bounded Waiting	✅ Guaranteed	❌ Not guaranteed

Dijkstra's (D's) Algorithm – Optimized Peterson 📚

- Replaces the two-process flag/turn scheme with a more scalable approach for *n* processes.
- Retains the same three correctness properties: **mutual exclusion**, **progress**, and **bounded waiting**.

Core Idea

1. Each process announces its intent with a flag (similar to Peterson).
2. A shared "turn" variable (or a hierarchy of turns) determines which process gets priority.
3. The algorithm ensures that only one process proceeds to the critical section while others wait, and that waiting is bounded.

> **Result:** Dijkstra's algorithm is a correct solution for mutual exclusion, providing both progress and bounded waiting for any number of processes.

Dekker's Algorithm 🛡️

Overview

- Each process has a **flag** (true = interested, false = not interested).
- A shared variable **turn** indicates which process may enter the critical section when both are interested.

Execution Steps

1. **Non-critical section** – process finishes its work.
2. Set its own flag to **true** (interested).
3. If the other process's flag is also **true**, the process checks turn.
    - If turn equals the other process, it **waits** (busy-wait) until turn changes.
    - When turn equals its own ID, it proceeds to the critical section.
4. **Critical section** – performs exclusive work.
5. On exit, set its own flag to **false** and set turn to the other process's ID, allowing the partner to enter.

Correctness Properties

Property	How Dekker's algorithm ensures it
Mutual Exclusion	Two Boolean flags indicate intent; turn resolves conflicts when both are true, guaranteeing that only one process can be in the critical section at a time.
Progress	If a process is not interested (flag = false), it never blocks the other; the interested process eventually sees turn in its favor and enters.
Bounded Waiting	After a process leaves, it sets turn to the other process, ensuring the partner gets the next turn and preventing a process from entering repeatedly.

Synchronization Problems Example 🤔

Two processes **P1** and **P2** use Boolean variables want1 and want2 (initially false).

Mutex (Mutual Exclusion)

- Both processes set their want flag to **true** and then test the other's flag.
- If the other flag is also **true**, each enters a busy-wait loop, so **both are blocked** → **deadlock**.

- Since neither can enter the critical section simultaneously, **mutual exclusion is satisfied**.

Progress

> **Progress**: *If a process wants to enter the critical section and the other does not, the interested process must eventually be allowed to enter.*

- When P2 is uninterested (want2 = false), P1 can set want1 = true and enter because the while-condition (while want2 == true) is false.
- Thus, an uninterested process never blocks the other → **progress is guaranteed**.

Bounded Waiting

Situation	Observation
After P1 enters and exits, want2 is set to false.	P2 can immediately enter on its next attempt.
If a process repeatedly re-sets its own want to **true** while the other's flag stays **false**, it may re-enter the critical section many times in a row.	**Bounded waiting is *not* guaranteed** for this algorithm (unlike strict alternation).

Summary Table

Property	Satisfied?	Reason
Mutual Exclusion	✅	Flags prevent simultaneous entry.
Progress	✅	Uninterested process never blocks the other.
Deadlock	✅ (present)	Both may wait forever when both are interested.
Bounded Waiting	❌	A process can re-enter repeatedly while the other waits.

Hardware Synchronization 🔧

Atomic Instructions

> **Atomic**: *An operation that cannot be preempted until it completes; it appears indivisible to other processors.*

- Provided by the processor, usable from user mode.
- Ensure correct behavior for concurrent updates without additional locking.

Test-and-Set Lock (TSL) 🔪

> **Test-and-Set (TSL)**: *An atomic instruction that returns the current value of a lock variable and simultaneously sets that lock to true.*

Function prototype (illustrative C-style)

```
bool test_and_set(bool *lock);
```

Behavior

1. Read the current value of *lock.
2. Store that value as the return result.
3. Set *lock to true.
- If the lock was false, the caller obtains the lock (return false).

- If the lock was already true, the caller learns it is taken (return true) and must retry (busy-wait).

Using TSL for Critical Sections

- A process repeatedly executes while (test_and_set(&lock)) ; – this loop spins until the lock is acquired.
- After finishing the critical section, the process sets lock = false to release it.

Key points

- TSL is a **multiprocess** solution (works for any number of processes).
- It relies on the **atomicity** of the hardware instruction, eliminating the need for higher-level software flags.

Comparison of Software vs. Hardware Solutions

Aspect	Software (e.g., Dekker, Peterson)	Hardware (e.g., TSL)
Process count	Typically 2-process algorithms	Works for many processes
Synchronization primitive	Boolean flags, turn variable	Atomic test-and-set instruction
Busy-waiting	Yes (spin loops)	Yes (spin loops) but at hardware level
Guarantee of properties	Mutual exclusion, progress, bounded waiting (depends on algorithm)	Mutual exclusion and progress; bounded waiting may need additional protocol
Implementation complexity	Higher (logic in code)	Lower (single atomic instruction)

Key Takeaways

> **Mutual Exclusion** ensures that *only one* process is in the critical section at any time.
> **Progress** guarantees that a process wanting to enter will eventually be allowed if the other is not interested.
> **Bounded Waiting** limits the number of times other processes can enter before a waiting process gets its turn.

Hardware atomic instructions like **TSL** provide a simple, scalable foundation for building correct synchronization mechanisms across multiple processes.

🔐 Test-and-Set Lock (TSL)

> **Definition:** *Test-and-Set Lock* is an atomic instruction that returns the current value of a lock variable and simultaneously sets that lock to **true**.

📋 How TSL Operates

- The instruction executes three sub-operations **atomically** (no preemption):
  1. **Read** the current value of log.
  2. **Store** that value in a temporary return variable.
  3. **Set** log to **true**.
- The returned value reflects the *previous* state of log.

🔁 Execution Scenario with Two Processes (P1 & P2)

Step	Process	Action	Result
1	P1 (non-critical)	Executes TSL(log) while log = false.	Returns **false**, sets log = true.
2	P1	false == true → **false**, so P1 enters **critical section**.	

3	P1	Preempted (still inside critical section).	
4	P2 (non-critical)	Executes TSL(log) while log = true.	Returns **true**, keeps log = true.
5	P2	true == true → **true**, stays in the **entry loop** (spins).	
6	P1	Completes critical section, executes exit section: log = false.	
7	P2	Executes TSL(log) again, now log = false.	Returns **false**, sets log = true.
8	P2	false == true → **false**, enters **critical section**.	

✅ Properties of TSL

- **Mutual Exclusion** ✅

> *Only one process can be in the critical section at a time.*
> The atomic nature of TSL ensures that when one process sets log to **true**, any other process sees log = true and cannot pass the entry test.

- **Progress** ✅

> *If no process is in the critical section and some processes wish to enter, one of them will eventually succeed.*
> When a process is not interested (its non-critical section never touches log), log stays **false**, allowing an interested process to pass the entry test immediately.

- **Bounded Waiting** ❌ (not guaranteed)

> *There must be a limit on the number of times other processes can enter the critical section after a process has requested entry.*
> In the basic TSL scheme, a process (e.g., P1) can repeatedly reacquire the lock after each exit, starving the other process (P2) indefinitely. The Galvin textbook adds extra logic (e.g., a turn variable) to achieve bounded waiting, but that logic is **absent** here.

📊 Property Summary

Property	Satisfied?	Reason
Mutual Exclusion	✅	Atomic test-and-set prevents simultaneous entry.
Progress	✅	Uninterested processes never block the lock.
Bounded Waiting	❌	A process may repeatedly reacquire the lock before others get a chance.

🔄 Swap Mechanism (Lock-Key)

> **Definition:** The *swap mechanism* atomically exchanges the values of two variables (here, lock and key) using a temporary storage, ensuring the three underlying operations occur without interruption.

🛠️ Core Idea

- **Atomic swap** of lock and key is performed in a single, indivisible step.
- Implemented with a **temporary variable** to avoid losing a value during the exchange.

📈 Solving the Critical Section with Swap

1. **Initialize** global lock = false. Each process has a private key = true.
2. **Entry Section (per process)**
   - While key == true:
     - Perform atomic swap swap(lock, key).
     - After swap, if key == false, the process proceeds to the critical section; otherwise it repeats the loop.
3. **Critical Section** – the process executes its exclusive code.
4. **Exit Section** – set lock = false (or perform another swap) so that the next waiting process can acquire the lock.

📋 Example Walkthrough

Phase	Process	lock	key	Action	Outcome
Start	P1	false	true	swap(lock, key) → lock = true, key = false	P1 exits the while loop, enters critical section.
Preempt	P2	true	true	swap(lock, key) → lock = true, key = true (no change)	P2 continues looping (spins).
Exit P1	P1	true → false (exit)	—	lock set to false.	Allows next swap to succeed.
Resume P2	P2	false	true	swap(lock, key) → lock = true, key = false	P2 now proceeds to critical section.

✅ Guarantees Provided by Swap

- **Mutual Exclusion** – only the process that obtains key = false after the swap can enter the critical section.
- **Progress** – if a process is not interested, it never alters lock, so an interested process can acquire the lock.
- **Bounded Waiting** – the basic swap alone does **not** enforce a strict bound; additional turn-taking logic (as in Galvin's extension) would be required.

🔐 Mutual Exclusion with the Swap (Test-and-Set) Mechanism

> **Swap (Test-and-Set)** – an atomic instruction that reads a lock variable, returns its old value, and simultaneously writes a new value (typically **true** to indicate "locked").

- **Execution flow**
  1. **Process P1** finishes its non-critical section, sets key = true, then executes *swap* on lock.
  2. lock becomes **true**, key becomes **false** → P1 enters the critical section.
  3. **P2** reaches its non-critical section, sees key = true, attempts *swap* on lock.
  4. Because lock is already **true**, P2 keeps swapping (busy-waiting) until P1 releases the lock.
- When P1 exits, it flips lock back to **false**.
- P2 then observes key = false and can acquire the lock, entering its critical section.

✅ Guarantees

Property	Satisfied?	Reason
Mutual Exclusion	✓	Only one process holds lock = true at any time.
Progress	✓	No uninterested process changes lock; waiting processes merely spin.
Bounded Waiting	✗	A process may be starved indefinitely if another repeatedly re-enters the critical section.

> **Bounded Waiting** – a guarantee that every requesting process will enter the critical section after a bounded number of other entries.

⚠️ Starvation Risk

- If a process repeatedly finishes its critical section and immediately re-acquires the lock, other processes can wait **unboundedly**.
- The swap mechanism alone does **not** enforce bounded waiting.

❓ Test-and-Set Correctness Question

> **Scenario:** X is initialized to 0 (critical section free). test_and_set(X) returns the old value of X and sets it to 1 (critical section busy).

Option	Statement	Verdict
1	The solution is **deadlock-free**.	✓ (test-and-set cannot deadlock)
2	The solution is **starvation-free**.	✗ (processes can be starved)
3	The solution restricts entry to a **single folder**.	✗ (no such restriction)
4	**More than one** process can be in the critical section simultaneously.	✗ (mutual exclusion holds)

Correct answer: Option 1 only.

🔄 Priority Inversion & Deadlock Example

> **Priority Inversion** – a low-priority process holds a lock while a high-priority process waits for it, potentially causing deadlock when combined with preemptive priority scheduling.

Narrative

Process	Priority	Situation
P_L (Low)	Low	Holds the critical section, then is pre-empted by the CPU scheduler when a high-priority process arrives.
P_H (High)	High	Wants the critical section, but cannot enter because lock is held by P_L.
Deadlock	–	P_H waits for the lock; P_L cannot finish because it needs the CPU (currently held by P_H). Both wait forever.

- This exact scenario occurred in **NASA's Mars Pathfinder** mission.

Conflict of Principles

1. **Priority rule:** Low-priority processes should yield to higher-priority ones.
2. **Mutual exclusion rule:** Only one process may be inside the critical section.

When both rules are enforced strictly, each process "stubbornly" (ad-*ment*) refuses to give way → deadlock.

🛠️ Solution: Priority Inheritance

> **Priority Inheritance** – the low-priority process temporarily **inherits** the higher priority of the waiting process.

- **Steps**
  1. P_L (holding the lock) inherits P_H's high priority.
  2. Scheduler gives CPU to P_L, allowing it to complete the critical section and release the lock.
  3. After releasing, P_L reverts to its original low priority.
  4. P_H now acquires the lock and proceeds.
- This breaks the deadlock cycle and restores progress while respecting mutual exclusion.

📊 Busy-Wait Lock with Fetch-and-Add

> **Fetch-and-Add (F&A)** – an atomic *read-modify-write* instruction that:
>
> 1. Reads the current value of a memory location **X**.
> 2. Adds an increment **I** to **X**.
> 3. Returns the *old* value of **X**.

Lock Variables

- L – integer lock variable, initially **0** (available).
- Non-zero value → lock is **busy**.

Acquire Procedure (pseudo-logic)

1. while (fetch_and_add(L, 1) != 0) // spin while lock is already taken
2. After the loop, set L = 1 explicitly (optional reinforcement).

Release Procedure

- L = 0 // make the lock available again

How It Works

- The first call to fetch_and_add(L,1) when L = 0 returns **0** and sets L to **1**, allowing the caller to enter the critical section.
- Subsequent callers see a non-zero return value, continue spinning until the lock is released.

> **Busy-Waiting** – processes repeatedly test the lock variable, consuming CPU cycles while waiting.

🧩 Problem Overview

- **Goal:** Analyze a synchronization algorithm that uses a *fetch-and-add* primitive on a variable log (the lock) and an assignment L = 1 in the entry section.
- **Key Variables:**
  - log – lock counter modified atomically by *fetch-and-add*.
  - L – flag set to 1 in the entry section and cleared to 0 in the exit section.

🔐 Synchronization Mechanism Properties

Mechanism	Mutual Exclusion	Bounded Waiting	Progress
Lock variable	✖ (does not guarantee)	✖	✅
Test-and-Set (TAS)	✅	✖	✅
Swap	✅	✖	✅
TAS + Swap (with extra logic)	✅	✅	✅
Fetch-and-Add (used here)	✅ (when $L = 1$ is present)	*Not guaranteed*	✅

> **Definition – Mutual Exclusion:** *Only one process may be inside the critical section at any given time.*
> **Definition – Bounded Waiting:** *There exists an upper bound on the number of times other processes can enter their critical sections after a request has been made.*
> **Definition – Progress:** *If no process is in the critical section and some processes wish to enter, one of those processes will eventually be allowed to enter.*

🧱 Fetch-and-Add Primitive

> **Definition:** *fetch_and_add(x) returns the current value of x and then atomically increments x by 1.*

In the algorithm:

1. temp = fetch_and_add(log) – temp receives the old value of log.
2. log becomes $log + 1$ atomically.

The returned temp is used to decide whether the process proceeds to the critical section.

❓ Questions to Solve

1. **Why is $L = 1$ written before the while loop?**
2. **What overflow would occur if $L were not set to 1?**
3. **Does the code guarantee mutual exclusion?**
4. **Can the lock be 1 while the critical section is free (possible starvation)?**
5. **Is it possible that no process can ever enter the critical section?**

✅ Answer 1 – Role of $L = 1$ (Preventing Overflow)

- Without $L = 1$, $logcanincreasewithoutbound: eachentryincrementslog and never resets it.
- Example trace (no $L = 1$):
  1. $log = 0 → returns 0, becomes 1.
  2. Next process reads 1, returns 1, becomes 2.
  3. Subsequent processes see 2, 3, 4, …
- As more processes repeatedly enter, log eventually **overflows** (wrap-around to 0), breaking the lock logic.
- Setting $L = 1$ forces the algorithm to reset log to 1 after each exit, causing log to **oscillate between 1 and 2**, thus preventing unbounded growth and overflow.

✅ Answer 2 – Mutual Exclusion

- **Scenario with three processes (P1, P2, P3):**

1. **P1** executes fetch_and_add(log) when $log = 0 \rightarrow gets\,0, sets$log = 1, enters critical section, then is pre-empted.
2. **P2** runs fetch_and_add(log) → gets 1, sets $log = 2. Because$L was set to 1, the entry loop forces $log back to 1 before the next check, causing P2 to **busy-wait**.
3. When **P1** finishes, it executes the exit section (L = 0), resetting $log to 0.

- After the reset, the next waiting process (e.g., P2) sees $log = 0, obtains the lock, and enters.
- At any moment, **only one process** can be in the critical section, satisfying **mutual exclusion**.

✅ Answer 3 – Can $log = 1 While Critical Section Is Free? (Starvation)

- Yes. If a process is pre-empted **after** fetch_and_add(log) returns 1 **but before** it executes the statement that resets $log to 1, the lock remains at 2 while the critical section is empty.
- Example timeline:
  1. P1 enters, sets $log = 1, pre-empted inside the critical section.
  2. P2 executes fetch_and_add(log) → obtains 1, increments $log to 2, then is pre-empted **before** executing L = 1.
  3. P1 finishes, sets $log = 0.
  4. P2 resumes, sees $log = 2 (greater than 1) and **busy-waits** in the entry loop because the condition $log != 1 holds.
- During this busy-waiting period, **no process is in the critical section** even though $log is non-zero, demonstrating a **starvation-like situation**.

✅ Answer 4 – Possibility of Permanent Blockage

- The algorithm can enter a state where processes continuously loop in the entry section without ever resetting $log to 0 (e.g., if a process is repeatedly pre-empted right after incrementing $log).
- However, because the exit section explicitly sets $log = 0, **eventual progress** is guaranteed as long as each process eventually reaches its exit code.
- Thus, **permanent blockage** is not possible under the assumption of *fair scheduling* (every process eventually runs).

📊 Summary Table of Key Behaviors

Condition	$log Value	Critical Section Status	Outcome
Normal entry (with $L = 1$)	0 → 1	Free	Process enters
Second process attempts entry	1 → 2 (then reset to 1)	Occupied by first process	Busy-wait
First process exits (L = 0)	0	Free	Next waiting process proceeds
No $L = 1$ (absent)	Increases without bound	May become free but risk overflow	Overflow → lock failure
Pre-empt after increment, before reset	2	Free	Busy-wait → temporary starvation

---

These notes capture the essential reasoning, definitions, and outcomes required to answer the five posed questions.

Critical Section and Lock Issues 🔒

> **Critical Section:** A portion of code that must not be executed by more than one process at the same time.

- **Lock variable** (log in the transcript) indicates the state of the critical section
  - 0 → critical section is **free**.
  - 1 → critical section is **locked** (occupied).
- **Scenario analysis**

1. P1 exits the critical section and sets log = 0.
2. P2 arrives, erroneously sets log = 1 without actually entering the critical section.
3. The lock now incorrectly signals "occupied" while the section is empty → deadlock-like condition.
4. If P3 later attempts entry, it finds log ≠ 0 and stays blocked, even though no process is inside.

- Overflow risk
  - Each preempted process may increment the lock before failing to set it to 1.
  - Repeated increments can cause the lock value to exceed the maximum representable integer.
  - Example on a 16-bit signed integer: maximum value 32767.
  - If the number of processes > 32767, the lock overflows and the mechanism breaks.

Starvation in Scheduling Algorithms ⚡

> **Starvation:** A situation where a process waits indefinitely because the scheduling policy never selects it.

Algorithm	Starvation?	Reason
SGF (Shortest-Job-First)	Yes	Shorter jobs keep being chosen, longer jobs may never run.
Priority scheduling	Yes	Higher-priority processes can continuously preempt lower-priority ones.
FCFS (First-Come-First-Served)	Generally No	Processes are served in arrival order; only exceptional cases (e.g., infinite loops) cause starvation.

Key note: When answering "which algorithm suffers from starvation," consider general behavior; FCFS is excluded unless an explicit exceptional case is mentioned.

Busy Waiting vs. Non-Busy Waiting ⏳

> **Busy waiting:** Repeatedly testing a condition in a loop, consuming CPU cycles.

- Implemented with a while loop that checks the lock continuously.
- Wastes CPU time even when the critical section is free.

> **Non-busy waiting:** Tests the condition once; if the critical section is busy, the process blocks and yields the CPU.

- Implemented with an if-else construct.
- The blocked process is placed in a sleep queue, freeing the CPU for the process that holds the lock.

Comparison Table

Aspect	Busy Waiting (while)	Non-Busy Waiting (if-else)
CPU usage	High (continuous polling)	Low (process blocked)
Responsiveness	Immediate detection when condition changes	Depends on wake-up mechanism
Suitability	Simple hardware without blocking primitives	Systems with sleep/wake primitives

Blocking Mechanisms: Sleep & Wakeup 😴🔔

> **Sleep:** A system call that atomically places a process into a *sleep queue* and releases the CPU.
> **Wakeup:** Removes a sleeping process from the sleep queue and places it back into the *ready queue*.

Example Flow

1. P1 attempts entry, discovers P2 is in the critical section.
2. P1 executes sleep(), moving to the Sleep Q and releasing the CPU.
3. P2 finishes its critical section, executes the exit section, and calls wakeup(P1).
4. P1 returns to the Ready Q, rechecks the entry condition, finds log = 0, and enters the critical section.

This mechanism eliminates wasted CPU cycles caused by busy waiting.

Non-Busy Solution to the Producer-Consumer Problem 📦

- Consumer behavior
  - If the buffer is empty → sleep() (blocks).
  - When the producer places an item, it calls wakeup(consumer).
- Producer behavior
  - If the buffer is full → sleep() (blocks).
  - When the consumer removes an item, it calls wakeup(producer).

The two processes cooperate: each wakes the other only when the condition it was waiting for becomes true, avoiding continual condition checks.

📦 Bounded Buffer Basics

🧭 Pointers in and out

> in – index of the next empty slot where the producer will place an item.
> out – index of the first filled slot from which the consumer will take an item.

Both pointers wrap around the buffer using modulo arithmetic (mod N) to achieve a circular buffer.

📊 Buffer State Variables

- count – current number of items in the buffer.
  - $count = 0$ → buffer empty.
  - $count = N$ → buffer full (where N is the buffer capacity).
- itemP – temporary variable holding the item produced.
- itemC – temporary variable holding the item consumed.

🚦 Producer Logic

🛏️ Sleep When Buffer Full

- If $count = N$ (buffer full) → producer executes sleep; it does not produce until woken.

📩 Produce an Item

1. Place the produced item into itemP.
2. Insert itemP into buffer[in].
3. Increment in (in = (in + 1) \bmod N).
4. Increment count (count = count + 1).
5. If $count = 1$ (first item placed) → wake the consumer.

🛒 Consumer Logic

🛏️ Sleep When Buffer Empty

- If $count = 0$ (buffer empty) → consumer executes sleep; it does not consume until woken.

📤 Consume an Item

1. Retrieve itemC = buffer[out].
2. Increment out (out = (out + 1) \bmod N).
3. Decrement count (count = count - 1).
4. If $count = N - 1$ (buffer was full and now has one free slot) → **wake** the **producer**.

⚠️ Deadlock & Race Conditions

🕸️ Deadlock Scenario

1. **Consumer** checks count = 0 and is preempted **before** sleeping.
2. **Producer** runs, fills the buffer, reaches $count = N$, and goes to **sleep**, assuming the consumer will wake it.
3. **Consumer** resumes, also goes to **sleep**, assuming the producer will wake it.
4. Both processes are asleep → **deadlock**.

🔀 Race Condition (Inconsistency)

- When **producer** and **consumer** attempt to modify count simultaneously, the value can become incorrect (e.g., expected 5 becomes 6 or 4).
- This unsynchronized access leads to **inconsistency**, a classic **race condition**.

🔐 Semaphore Solution (Sema4)

📖 Definition

> **Semaphore** – an abstract data type that holds an integer value and provides two atomic operations: **down** (P) and **up** (V). It is used to coordinate access to shared resources and solve deadlock and race-condition problems.

⚙️ Operations

- **down (P)** – atomically decrement the semaphore value; if the result would be negative, the calling process blocks.
- **up (V)** – atomically increment the semaphore value; if there are blocked processes, one is awakened.

🗂️ Types of Semaphores

Type	Value Range	Alternate Name
Binary Semaphore	0 or 1	**Mutex (M_X)**
Counting Semaphore	$-\infty$ to $+\infty$ (any integer)	**General semaphore**

🏗️ Semaphore Structure (Counting Semaphore)

- **value** – current integer count.
- **Q** – queue (list) of process control blocks (PCBs) blocked on an unsuccessful **down** operation.

> The **down** operation is unsuccessful when it would cause the semaphore's value to become negative; the requesting process is placed in **Q** and blocked until a corresponding **up** operation occurs.

📘 Counting Semaphore Overview

🔷 Definition

> **Counting semaphore** – a synchronization primitive that holds an integer value representing the number of available resources. Its operations are **atomic** and are performed in **kernel mode**.

🔷 Core Operations

- **Down (P) operation** – decrements the semaphore value by 1.
- **Up (V) operation** – increments the semaphore value by 1.

Both operations switch to **kernel mode** to ensure atomicity.

✅ Down Operation: Success vs. Failure

> The outcome of a down operation depends on the semaphore's current value:

Current value of S	Result of down	Next action
$S \geq 0$	**Successful** – process continues to the next statement.	Continue execution
$S < 0$	**Unsuccessful** – process is placed in the **sleep queue** Q_{sleep} and blocked.	Blocked (sleeping)

- When the result is successful, the process proceeds immediately.
- When the result is unsuccessful, the process's PCB is enqueued in Q_{sleep} (referred to as **QL**).

📊 Example: Initial Value $S = 1$

1. **Down** is performed → S becomes 0 (still non-negative) → **success**.
2. Another **down** → S becomes -1 → **failure** → process blocked in Q_{sleep} .

📈 Example: Initial Value $S = 5$

Down operation #	S after operation	Success?	# of blocked processes
1	4	Yes	0
2	3	Yes	0
3	2	Yes	0
4	1	Yes	0
5	0	Yes	0
6	-1	No	1
7	-2	No	2
8	-3	No	3

- The first five downs are **successful** because S remains non-negative.
- Starting with the sixth down, S turns negative; each negative unit corresponds to **one blocked process**.

🔪 Inference

- **Positive (or zero) value** of a counting semaphore indicates the exact number of **down operations that can succeed**.
- **Magnitude of a negative value** $|S|$ equals the **number of processes currently blocked** in Q_{sleep} .

🔒 Mutual Exclusion via Counting Semaphore

Consider three processes P_1, P_2, P_3 sharing a critical section protected by a semaphore S initialized to 1 .

Step	Process	Action	S value	Result
1	P_1	**Down** (entry)	0	Enters critical section
2	P_2	**Down** (entry)	-1	Blocked in Q_{sleep}
3	P_3	**Down** (entry)	-2	Blocked in Q_{sleep}

4	P_1	**Up** (exit)	-1	Wakes one blocked process (e.g., P_2)
5	P_2	Resumes → **critical section**	$-1 \to 0$ after its **up**	Completes, wakes P_3
6	P_3	Resumes → **critical section**	$0 \to 1$ after its **up**	Completes, semaphore returns to 1

- While a process is inside the critical section, $S = 0$; any other process attempting **down** gets a negative result and is blocked.
- An **up** operation on a negative S increments the value and **awakens one sleeping process**, preserving mutual exclusion.

🔁 Up Operation Details

> **Up (V) operation** – increments S by 1. If the new value is ≤ 0, a process is removed from Q_{sleep} and awakened; otherwise, the operation simply returns.

- When S is negative, the up operation **reduces the magnitude of the negative value**, thereby decreasing the count of blocked processes and potentially waking one.
- When S becomes non-negative, no process is awakened.

Key Takeaways

- Counting semaphores manage concurrent access by tracking available "permits".
- The sign of the semaphore value distinguishes **success** (≥ 0) from **blocking** (< 0).
- The absolute value of a negative semaphore equals the **number of blocked processes**.
- Proper sequencing of **down** and **up** operations guarantees **mutual exclusion** for critical sections.

Semaphore Operations 🔁

Up (V) operation

> **Definition:** Increments the semaphore value by 1. If the value after increment is **≤ 0**, a process from the waiting queue is awakened; otherwise no process is woken.

- **Effect on value:** $s \leftarrow s + 1$
- **Wake-up condition:** $s \leq 0$ → select one PCB from the sleep queue and move it to the ready state.

Down (P) operation

> **Definition:** Decrements the semaphore value by 1. If the resulting value is **negative**, the calling process is placed in the sleep queue; otherwise it proceeds to the critical section.

- **Effect on value:** $s \leftarrow s - 1$
- **Blocking condition:** $s < 0$ → enqueue the process PCB in the sleep queue.

Mutual Exclusion & Semaphore Values 🔐

Semaphore value	Interpretation	Action
$s > 0$	At least one slot available	No process is awakened on an up; down succeeds without blocking.
$s = 0$	No slots available	Up does **not** wake a process (nothing in sleep queue). Down blocks the caller.
$s < 0$	One or more processes are waiting	Up wakes exactly one waiting process (the most recent blocked).

- If the initial value of a counting semaphore is **greater than 1**, multiple processes can enter the critical section simultaneously, **violating mutual exclusion**.
- Example: with $s = 2$, two down operations by P1 and P2 both succeed, allowing both processes to be in the critical section at the same time.

Counting Semaphore Examples 📊

Example 1 – Final value calculation

Operations (down = -1, up = +1):

```
8  - 10 + 1 - 5 + 2 - 6 + 3
```

- **Computation:** $8 - 10 + 1 - 5 + 2 - 6 + 3 = -7$
- **Interpretation:** The semaphore value becomes **-7**, meaning **7 processes are blocked** in the sleep queue.

Example 2 – Finding an initial value for a target final value

Wanted final value: **-6** after the same sequence of up/down operations.

- Let initial value be x.
- Equation: $x - 10 + 1 - 5 + 2 - 6 + 3 = -6 \rightarrow x - 15 = -6 \rightarrow$x = 9$.
- Thus, starting with **9** yields a final value of **-6** (i.e., 6 processes blocked).

Example 3 – Largest initial value leaving at least one process blocked

- Operations: 20 down (P) and 12 up (V).
- Net change: $-20 + 12 = -8$.
- To have at least one blocked process, final value must be ≤ -1.
- Condition: $x - 8 \leq -1 \rightarrow$x ≤ 7$.
- **Largest permissible initial value: 7**.

Blocking Conditions & Sleep Queue 📥

- A process **remains blocked** when, after its down operation, the semaphore value is **negative**.
- The **sleep queue** holds the PCBs of all processes that performed an unsuccessful down.
- An up operation **wakes only one** waiting process (if any) when the post-increment value is ≤ 0.

Client-Server Analogy 🌐

- **Scenario:** Server can handle **100** concurrent requests; **5,000** clients send requests.
- **Solution:** Use a counting semaphore initialized to **100** (or 1000 in the transcript's example).

Action	Semaphore effect
Request arrives → **down**	Decrements value; if value becomes negative, the request waits.
Request completed → **up**	Increments value; possibly wakes a waiting request.

- The semaphore limits the number of active requests to the server's capacity, analogous to limiting processes in a critical section.

Binary Semaphore (Binary Sem4) ⚙️

> **Definition:** A semaphore whose value can only be **0** or **1**; used to enforce strict mutual exclusion.

- **Declaration (conceptual):** typedef struct { enum {0,1} value; queue sleep_q; } binary_sem4;
- **Initialisation:** Set to **0** (no process may enter) or **1** (one process may enter).

Down operation on binary semaphore

> **Definition:** Attempts to acquire the lock.

- If $s = 1$: set $s \leftarrow 0$ and **return** (acquisition successful).
- If $s = 0$: place the calling PCB in the **sleep queue** and block it.

Up operation on binary semaphore

> **Definition:** Releases the lock.

- Increment $s \leftarrow s + 1$.
- If the new value is **≤0** (i.e., $s = 0$ after increment because it was -1 before), wake one process from the sleep queue; otherwise, simply return.
- Because the value never goes below **0**, the semaphore never records a count larger than one, guaranteeing that **exactly one** process can be in the critical section at any time.

🟦 Binary Semaphore Overview

> **Binary semaphore** – a synchronization primitive that can hold only two values: 0 (unavailable) or 1 (available). It is used to enforce *mutual exclusion* for a critical section.

- When the value is 1, the critical section is free and no process is waiting.
- When the value is 0, either a process is inside the critical section **or** one or more processes are blocked in the sleeping queue.

⬇️ Down Operation (P)

> The **down** (or **P**) operation attempts to acquire the semaphore.

Current value of s	Action	Result
1	Decrease s to 0	Process enters the critical section (success).
0	Block the requesting process	Process is placed in the **sleeping queue** (blocked).

- The down operation can **succeed** (when $s = 1$) or **fail** (when $s = 0$, leading to blocking).
- Unlike counting semaphores, a binary semaphore never takes negative values, so the exact number of blocked processes cannot be deduced from s alone.

⬆️ Up Operation (V)

> The **up** (or **V**) operation releases the semaphore.

1. **If the sleeping queue is empty**
   - Set s to 1 .
   - The operation is always successful.
2. **If the sleeping queue is not empty**
   - Wake one process from the queue and hand it the CPU directly to the critical section.
   - s remains 0 (because a process will immediately occupy the critical section).

- The up operation is **always successful**; it never blocks a process.

- Performing an up on $s = 1$ leaves s at 1 (binary semaphores cannot exceed $\mathbb{1}$).

💤 Sleeping (Blocking) Queue

- The queue is **dynamic**: it grows as processes are blocked and shrinks when they are awakened.
- It stores only the *identities* of blocked processes; the semaphore value s does **not** reflect the queue length.
- Example: after several down operations on $s = 0$, the queue may contain many processes even though s stays 0 .

🔄 Comparison: Binary vs. Counting Semaphore

Feature	Binary Semaphore	Counting Semaphore
Value range	0 or 1	$0, 1, 2, \ldots$ (any non-negative integer)
Negative values	Never	Can become negative (e.g., -17 indicates 17 blocked processes)
Ability to infer queue length	**No** (value 0 gives no count)	**Yes** (negative value equals number of blocked processes)
Up operation effect when queue non-empty	Wake one process, s stays 0	Increment value; blocked processes may be awakened depending on implementation

📚 Example Scenario: Three Processes Competing for a Critical Section

1. **Initial state**: $s = 1$, sleeping queue empty.
2. **P1** executes down → s becomes 0, enters critical section.
3. **P2** attempts down on $s = 0$ → blocked, added to queue.
4. **P3** also attempts down on $s = 0$ → blocked, added to queue.

- At this point, $s = 0$ and the sleeping queue holds **P2** and **P3**.
- When **P1** exits, it performs up:
  - Since the queue is **not empty**, a blocked process (e.g., P2) is awakened and moves directly into the critical section; s remains 0 .

📈 Example Calculation: Sequence of Operations

> **Initial condition**: $s = 1$, sleeping queue empty.

Operation sequence:

1. **10 down**
2. **2 up**
3. **18 down**
4. **3 up**
5. **4 down**
6. **5 up**

Step-by-step outcome

Step	Operation	Processes entering CS	Processes blocked	Queue size after step	s value
0	Start	–	–	0	1
1	10 down	P1 enters CS; P2–P10 blocked	9	9	0
2	2 up	P2, P3 awakened → enter CS	P4–P10 remain blocked	7	0

3	18 down	P4–P10 (7) already blocked; 11 new processes (P11–P21) blocked	18	18	0
4	3 up	P4, P5, P6 awakened → enter CS	P7–P21 remain blocked	15	0
5	4 down	4 new processes (P22–P25) blocked	P7–P21 + P22–P25	19	0
6	5 up	P7, P8, P9, P10, P11 awakened → enter CS	Remaining blocked: P12–P25	**21**	0

- **Final semaphore value**: $s = 0$ (critical section occupied or queue non-empty).
- **Final sleeping queue size**: **21** processes.

✅ Key Takeaways

- $s = 1$ → critical section free, queue empty.
- $s = 0$ → either a process is in the critical section **or** one or more processes are blocked.
- **Down** may block; **up** never blocks.
- Binary semaphores provide mutual exclusion but do **not** reveal how many processes are waiting.
- Proper handling of the sleeping queue is essential for correct synchronization.

🟢 Binary Semaphore Basics

Binary semaphore – a semaphore whose value can be only **0** or **1**.

> **Down (P) operation** – decrements the semaphore.
> • If the value becomes **-1**, the invoking process is **blocked** and placed in the waiting queue Q.
> • If the value is **0**, the process is blocked while the value stays **0**.

> **Up (V) operation** – increments the semaphore.
> • If Q **is empty**, the semaphore value becomes **1**.
> • If Q **is not empty**, the value remains **0** and the **first process** in Q is woken up and removed from the queue.

📊 Counting Semaphore Overview

A **counting semaphore** may hold any non-negative integer value, allowing more than one process to enter the critical section simultaneously.

Operation	Effect when Q empty	Effect when Q not empty
Down	Decrease value; block if result < 0	Decrease value; block and enqueue
Up	Increase value	Keep value = 0, wake one waiting process

🔐 Mutual Exclusion with Semaphores

- When a process enters the **entry section**, it performs **down(s)**.
- If $s = 1$, the down operation changes it to 0, allowing the process to enter the critical section.
- Any other process that later performs **down(s)** finds $s = 0$, so it **blocks**.

> **Mutual exclusion** is guaranteed because the semaphore value 0 prevents more than one process from being in the critical section at the same time.

🚀 Progress Property

- A process that is **not interested** in the critical section never executes **down(s)**.
- Consequently it never enters the entry section and cannot block interested processes.

> **Progress** is satisfied because only interested processes compete for the semaphore; idle processes do not hinder the execution of others.

⏳ Bounded Waiting

- After a process finishes its critical section, it executes **up(s)**, which wakes **one** waiting process from Q.
- The awakened process gets the next chance to enter, while any others remain blocked.

> **Bounded waiting** holds because a process cannot be postponed indefinitely; each up operation releases at most one waiting process, guaranteeing a finite bound on the number of times other processes may enter before it.

📚 Example Scenario with Multiple Down/Up Operations

1. **Initial state**: $s = 1$, $Q = \emptyset$.
2. **10 down operations** → s becomes **0**; processes **P2–P10** are placed in Q (blocked).
3. **2 up operations** → Q not empty, so s stays **0** and **P2, P3** are removed from Q and resumed.
4. **3 up operations** later → **P4, P5, P6** wake; remaining processes **P7–P28** stay in Q .
5. **4 down operations** → four new processes join Q (now **P32** added).
6. **5 up operations** → **P7–P12** wake; the rest stay blocked.

The semaphore value remains **0** throughout; the size of Q changes as shown.

⚔️ Deadlock with Two Semaphores (s and t)

Two processes **I** and **J** use semaphores s and t, each initialized to **1**.

Entry sections

- **Process I**: down(s) ; down(t)
- **Process J**: down(t) ; down(s)

Possible execution leading to deadlock

1. **I** executes down(s) → $s = 0$.
2. **Preempt** → **J** executes down(t) → $t = 0$.
3. **I** resumes, attempts down(t) → $t = 0$, so **I blocks** and is placed in Q_t .
4. **J** resumes, attempts down(s) → $s = 0$, so **J blocks** and is placed in Q_s .

Both processes are waiting for a semaphore that the other holds; no other process can perform an **up** to release them.

> **Result: Deadlock** occurs, and **mutual exclusion** is still maintained (only one process could have entered the critical section, but both are now stuck).

Avoiding deadlock: consistent ordering

If both processes acquire the semaphores in the **same order** (e.g., down(s) ; down(t) for both), the scenario above cannot arise because the second down will always block the later process, leaving the first process able to complete its critical section and perform the necessary **up** operations.

Condition	Outcome
Different acquisition order	**Deadlock** possible
Same acquisition order	**Deadlock avoided**, mutual exclusion preserved

📐 Regular Expression of Concurrent Execution (S = 1, T = 0)

With $s = 1, t = 0$, processes I and J compete as follows:

- I must perform down(s) (possible) then down(t) (blocked because $t = 0$).
- J can only start with down(t) (blocked immediately).

Thus the only observable actions are the successful down(s) of I, after which the system is stuck until I performs an up(s) (which it cannot reach without acquiring t).

The generated regular expression reflects the single enabled operation:

```
down(s)_I
```

(Any further progress requires a change of t by an external event, which is not described.)

🤝 Alternating Execution of Processes I and J

> **Definition:** *Down (P) operation* decrements a semaphore; if the value becomes negative, the process blocks.
> **Definition:** *Up (V) operation* increments a semaphore and may wake a blocked process.

- Initial values: s = 1, t = 0
- **Process I** performs **down S** → s becomes 0, prints 0.
- **Process J** performs **down T** (blocked because t = 0) → cannot proceed until t is incremented.
- After **Process I** finishes its critical section, it performs **up T** → t becomes 1.
- **Process J** now succeeds, performs **down T** → t becomes 0, prints 1, then **up S** → s becomes 1.

The resulting output pattern repeats as:

```
0 0 1 1 0 0 1 1 …
```

This pattern corresponds to the regular expression **(00 11)***.

📌 Why Alternation Occurs

- I: down S, up T
- J: down T, up S
- Each process releases the semaphore needed by the other, forcing strict alternation of prints 0 and 1.

✅ Properties of the I–J Protocol

Property	Satisfied?	Reason
Mutual Exclusion	✓	A process cannot enter the critical section until it has performed a down on its own semaphore, which is 0 while the other is inside.
Progress	✗	If one process (e.g., J) is uninterested (never executes its down on T), the other (I) cannot proceed because t never becomes 1.
Bounded Waiting	✓	The alternation guarantees that a waiting process will be allowed entry after at most one turn of the other process.

📚 Binary Semaphore Array Example (M_0-M_4) with Processes P_0-P_4

🔧 Process Code Sketch

Each process P_i (indices modulo 5) executes:

1. P(m_i) // down on its own semaphore
2. P(m_{i+1 mod 5}) // down on the next semaphore
3. *critical section*
4. V(m_i) // up its own semaphore
5. V(m_{i+1 mod 5}) // up the next semaphore

All semaphores are **binary** and initially 1.

📈 Maximum Simultaneous Critical Sections

- **Scenario:** P_0 acquires m_0 and m_1; P_2 acquires m_2 and m_3.
- P_1, P_3, and P_4 are blocked because at least one of their required semaphores is already 0.

Result: At most **2** processes can be in their critical sections simultaneously (e.g., P_0 and P_2). Mutual exclusion is **not** guaranteed for the whole system.

⚠️ Deadlock Possibility

If each process executes only its first P operation and then is pre-empted:

Process	First P on	Semaphore after first P
P_0	m_0	0
P_1	m_1	0
P_2	m_2	0
P_3	m_3	0
P_4	m_4	0

All semaphores become 0. Each process now waits for the second P (on the next semaphore), which is held by another blocked process. No process can reach its V operations → **deadlock**.

🔄 Binary Semaphores S, T, Z Example

- Initial values: S = 1, T = 0, Z = 0
- Processes: P_i, P_j, P_k (only P_i has a while loop)

Execution Flow

1. P_i starts (only one with S = 1), performs P(S) → S = 0.
2. Inside its loop, P_i eventually executes V(T) and V(Z) → T = 1, Z = 1.
3. The increase of T and Z enables P_j and P_k to pass their initial P(T) / P(Z) operations and enter their critical sections **once** (no loops).
4. After P_i finishes one iteration, it repeats the loop, again acquiring S (still 0 until it releases) and toggling T/Z as needed.

Key Observations

- P_i is the only process that can start initially because it owns the sole non-zero semaphore (S).
- P_j and P_k can run **once** each after P_i raises T and Z.
- The system demonstrates **strict ordering** enforced by semaphore initial values.

🌟 Star Printing with Semaphores

> **Definition:** A *star* is printed each time the shared variable s is set to 0 and the process executes its print operation.

Process Interaction

- **Variables:** s, T, Z (named VZ in the transcript).
- Initial state: s = 1.
- Sequence of operations (simplified):

Step	Process	Action on s	Result
1	**Pi**	set s = 0, print ★	1 star
2	**PJ**	change s to 1 (via VT/VZ)	s = 1
3	**PK**	**up** operation on s = 1 (no change) → waste	s = 1
4	**Pi** (optional)	set s = 0, print ★	additional star if executed

Minimum vs. Maximum Stars

- **Minimum stars: 2** – achieved with the order Pi → PJ → PK. The up-operation on PK is wasted, and no further Pi execution is possible because there is no while loop.
- **Maximum stars: 3** – achieved by inserting an extra Pi between PJ and PK: Pi → PJ → Pi → PK. The second Pi uses the up-operation that would otherwise be wasted, allowing one more star to be printed.

Why the Extra Pi Helps

- The up-operation on a value of 1 does **not** increase the value; it is effectively a *no-op*.
- Placing Pi before PK consumes this otherwise-wasted up-operation, resetting s to 0 again and permitting another star.

Variation Without VZ

- When the auxiliary semaphore VZ is omitted, the same reasoning applies, but the cycle cannot be extended beyond two prints.
- **Result:** Both minimum and maximum remain **2 stars**.

📊 Counting Semaphore Problem

> **Definition:** A *counting semaphore* S initialized to 2 controls access for four processes: Pi, PJ (increment C), PK, PL (decrement C). Each process runs **once** (no loops).

Possible Final Values of C

Execution Order	Effect on C	Final C (relative to initial)
Pi → PJ → PK → PL	+1 + 1 − 1 − 1	**0**
PK → PL → Pi → PJ	−1 − 1 + 1 + 1	**0**
All increments first, then decrements	+1 + 1 − 1 − 1	**0**
All decrements first, then increments	−1 − 1 + 1 + 1	**0**

- Because each increment is matched by a decrement, the **maximum** and **minimum** reachable values of C are the same: the net change is zero.
- **Result:** Final C equals its initial value regardless of the scheduling order.

(The transcript does not give an explicit numeric initial value for C; the reasoning shows that the net effect is always zero.)

🔒 Deadlock-Free Sequence with Binary Semaphores

> **Definition:** A *binary semaphore* is a lock that can be either 0 (locked) or 1 (unlocked).

Options Evaluated

Option	Sequence of Acquisitions	Outcome

1	PA → PB → PC → PD (all set to 0)	All processes become blocked → **deadlock**
2	PB → PA → PC → PD (PD eventually releases)	Processes release each other; execution completes → **deadlock-free**
3	PB → PC → PA → PD (some remain blocked)	Leads to deadlock → **not deadlock-free**
4	Similar to option 3, ends with deadlock	**deadlock**

- **Conclusion: Option 2** is the only deadlock-free sequence because PD is never permanently blocked and eventually signals its semaphore, allowing the others to proceed.

🚦 Synchronization Using Semaphores A and B (n Processes)

> **Definition:** wait(S) decrements semaphore S; if the result is negative, the process blocks. signal(S) increments S and may wake a blocked process.

Shared Variables

- **Semaphore A:** initialized to 1 (binary).
- **Semaphore B:** initialized to 0 (binary).
- **count:** shared integer, initialized to 0.
- **n:** total number of processes.

Code Section P (per process)

1. wait(A) → A becomes 0.
2. count = count + 1.
3. if (count == n) then signal(B).
4. signal(A) → A becomes 1.
5. wait(B).
6. signal(A).
7. wait(B).
8. signal(B).

Execution Example (n = 3)

Process	Steps Executed	State after Step
P1	1-4 (wait A, inc count→1, no signal B, signal A)	blocks on **wait B**
P2	1-4 (wait A, inc count→2, no signal B, signal A)	blocks on **wait B**
P3	1-4 (wait A, inc count→3, **signal B**, signal A)	proceeds, unblocks P1 & P2 via subsequent wait B/signal B sequence

- After P3 signals B, the blocked processes can pass their wait(B) calls, ensuring all processes finish **section P** before any enters **section Q**.

Guarantee Provided

> **No process executes code section Q before every process has finished code section P.**

This property is enforced by the combination of wait(A)/signal(A) (mutual exclusion) and the counting mechanism using count and semaphore B.

📋 Quick Reference Tables

Stars Printed

Scenario	Minimum ★	Maximum ★
With VZ enabled	2	3
Without VZ	2	2

Final C Values (Counting Semaphore)

Extreme	Final C (relative)
Minimum	0
Maximum	0

Deadlock-Free Option

Option	Deadlock?
1	Yes
2	**No**
3	Yes
4	Yes

All conclusions are derived directly from the described process interactions and semaphore semantics.

🛠️ Increment & Decrement Functions

> **Definition:** *A semaphore* is a synchronization primitive that can be used to control access to a shared resource by multiple processes.

> **Definition:** *Critical section* – the portion of code that accesses shared variables and must not be executed by more than the allowed number of processes simultaneously.

📄 Process Overview

- **Shared variable:** X, initially 10
- **Increment processes:** P_1, P_2, P_3, P_4, P_5 (5 total)
- **Decrement processes:** P_6, P_7, P_8 (3 total)

Each **increment** or **decrement** operation consists of three micro-operations:

1. **wait(S)** – acquire the semaphore
2. **load → modify → store** (increment or decrement X)
3. **signal(S)** – release the semaphore

🔐 Semaphore Cases

Case	Semaphore type	Value of S	Allowed concurrent processes in CS	Resulting X (min / max)
1️⃣	**Binary** (mutual exclusion)	1	1	$X = 12$ (both minimum and maximum)
2️⃣	**Counting**	2	2	Minimum $X = 7$, Maximum $X = 12$

Case 1 – Binary Semaphore ($S = 1$)

- Only one process may be inside the critical section at any time.
- Net change: 5 increments $-$ 3 decrements $= +2$.
- $X_{\text{final}} = 10 + 2 = 12$
- Because execution is sequential, the final value is invariant: **minimum = maximum = 12**.

Case 2 – Counting Semaphore ($S = 2$)

- Up to two processes can be inside the critical section simultaneously, creating possible race conditions.

Maximum X

- Let both concurrent slots be occupied by *increment* processes whenever possible.
- All five increments eventually succeed before the three decrements can affect the value, yielding the same net +2 as in the binary case: $X_{\max} = 12$.

Minimum X (most detrimental ordering)

1. Two processes enter together; arrange for a *decrement* to win each race.
2. Sequence that drives X lowest:
   - All five increments raise X to 15 (if they all run before any store from a decrement).
   - Decrement P_6 finally stores 9 (overwrites 15).
   - Decrement P_7 stores 8, then P_8 stores 7 .
- No further stores occur, so the **minimum reachable value is $X_{\min} = 7$**.

Thus the correct answer for this gate-style question is 12 **(max) and** 7 **(min)**.

📊 Summary of Minimum & Maximum Values

Scenario	Semaphore	Allowed concurrency	$X_{\min}$	$X_{\max}$
Increment/Decrement (5 inc, 3 dec)	Binary ($S = 1$)	1	12	12
Increment/Decrement (5 inc, 3 dec)	Counting ($S = 2$)	2	7	12
Variable C (2 inc, 2 dec)	Counting ($S = 2$)	2	-2	$+2$
Variable C (2 inc, 2 dec)	Binary ($S = 1$)	1	0	0

> **Definition:** *Counting semaphore* – a semaphore initialized to a non-negative integer n, allowing up to $n concurrent acquisitions.

> **Definition:** *Binary semaphore* – a counting semaphore restricted to values 0 or 1, providing mutual exclusion.

📚 Additional Example: Variable C

- **Initial value:** $C = 0$
- **Processes:**
  - Incrementers: P_i, P_j (2)
  - Decrementers: P_k, P_l (2)

Counting Semaphore $S = 2$

- Two processes may enter the critical section together.
- **Maximum C:** both incrementers win → $C_{\max} = +2$.
- **Minimum C:** both decrementers win → $C_{\min} = -2$.

Binary Semaphore $S = 1$

- Only one process at a time → operations execute sequentially.

- Net change $+2 - 2 = 0 \rightarrow C_{\min} = C_{\max} = 0$.

🏭 Producer-Consumer Problem

⚙️ Issues with Naïve Implementations

- **Busy-waiting** (spin loops) → can cause *inconsistency* because producers and consumers may read/write the buffer simultaneously.
- **Sleep-wake-up** (blocking) → may lead to *deadlock* if both producer and consumer sleep while waiting for each other.

✅ Correct Semaphore-Based Solution

- **Buffer:** bounded buffer of size $N = 100$.
- **Semaphores:**
  - empty – counting semaphore, initial value 100 (all slots empty).
  - full – counting semaphore, initial value 0 (no filled slots).
  - mutex – binary semaphore, initial value 1 (ensures mutual exclusion on the buffer).
- **Shared indices:**
  - in – points to the next empty slot for the producer.
  - out – points to the next filled slot for the consumer.

Producer Steps (conceptual)

1. wait(empty) – ensure there is at least one empty slot.
2. wait(mutex) – obtain exclusive access to the buffer.
3. Place the produced item at buffer[in]; update in = (in + 1) mod N.
4. signal(mutex) – release exclusive access.
5. signal(full) – indicate a new filled slot.

Consumer Steps (conceptual)

1. wait(full) – ensure there is at least one filled slot.
2. wait(mutex) – obtain exclusive access.
3. Retrieve the item from buffer[out]; update out = (out + 1) mod N.
4. signal(mutex) – release exclusive access.
5. signal(empty) – indicate a newly empty slot.

This semaphore arrangement eliminates both inconsistency (by protecting the buffer with mutex) and deadlock (by correctly ordering wait/signal on empty and full).

📘 Producer-Consumer Synchronization with Semaphores

🔧 Semaphore Basics

> **Semaphore:** A synchronization primitive that maintains a non-negative integer count.
> **down (wait):** Decrements the count; if the result would be negative, the calling process blocks.
> **up (signal):** Increments the count; if there are blocked processes, one is awakened.

> **Mutex:** A binary semaphore (initial value = 1) used to enforce **mutual exclusion** in a critical section.

🛠️ Entry, Critical, and Exit Sections

Producer

1. **Entry Section**
   - down(empty) *(block if buffer is full)*
   - down(mutex) *(ensure exclusive access)*
2. **Critical Section**

- Place produced item **P** into the buffer at index in (the next empty slot).
- Update in = (in + 1) % N *(circular buffer)*

3. **Exit Section**
   - up(mutex) *(release exclusive lock)*
   - up(full) *(signal that a new filled slot exists)*

Consumer

1. **Entry Section**
   - down(full) *(block if buffer is empty)*
   - down(mutex) *(mutual exclusion)*
2. **Critical Section**
   - Retrieve item **C** from buffer at index out.
   - Update out = (out + 1) % N.
3. **Exit Section**
   - up(mutex) *(release lock)*
   - up(empty) *(signal an available empty slot)*

✅ Five Essential Rules for Correct Implementation

#	Rule	Reason
1️⃣	**Block producer** when the buffer is full.	Prevents over-filling.
2️⃣	**Block consumer** when the buffer is empty.	Prevents under-flow.
3️⃣	**Producer wakes consumer** if the consumer is sleeping.	Guarantees progress after a new item appears.
4️⃣	**Consumer wakes producer** if the producer is sleeping.	Guarantees progress after a slot becomes free.
5️⃣	**Mutual exclusion** must exist between producer and consumer.	Avoids race conditions on shared indices and buffer.

⚠️ Deadlock and Ordering of Semaphore Operations

- **Correct order:** down(empty) → down(mutex) for producer; down(full) → down(mutex) for consumer.
- **Incorrect order** (e.g., down(mutex) before down(empty)):
  1. Producer acquires mutex (value → 0).
  2. Tries down(empty); if empty is 0, it blocks while holding mutex.
  3. Consumer may also acquire mutex and block on down(full).
  4. Both processes wait forever → **deadlock**.

> **Key Insight:** mutex must be the **second** operation in the entry section; placing it first creates a circular wait condition.

📚 Example Question 1 – Swapped Semantics

Problem:

Initial values: empty = 0, full = N, mutex = 1.

Here **empty** denotes *available slots filled with items* (i.e., slots ready for the consumer), and **full** denotes *empty slots* (i.e., space for the producer).

Required mapping of placeholders **P, Q, R, S** to semaphore variables so the code works.

Placeholder	Correct Semaphore
P	full
Q	empty
R	empty

S	full

> **Answer: C – full, empty, empty, full**

The solution follows from recognizing the swapped meanings and assigning each operation accordingly.

📖 Example Question 2 – Procedure Using SEMA-4

Given:

- empty = N (initially all slots free)
- full = 0 (no items)
- mutex = 1

Producer Loop (pseudocode)

1. down(empty) *(wait for a free slot)*
2. down(mutex) *(enter critical section)*
3. **Produce** an item and place it in the buffer.
4. up(mutex) *(leave critical section)*
5. up(full) *(signal that a new item is available)*

Consumer Loop (pseudocode)

1. down(full) *(wait for an item to consume)*
2. down(mutex) *(enter critical section)*
3. **Consume** the item from the buffer.
4. up(mutex) *(leave critical section)*
5. up(empty) *(signal that a slot has become free)*

These steps embody the same five essential rules listed earlier and demonstrate proper synchronization without deadlock.

📦 Producer-Consumer Semaphore Analysis

> **Semaphore:** a synchronization primitive that controls access to a shared resource by maintaining a counter.

- **Producer actions**
  1. Acquire lock (down on semaphore S).
  2. Add item to the buffer.
  3. Release lock (up on S).
  4. Increment the count of items (N).
- **Consumer actions**
  1. Acquire lock (down on semaphore S).
  2. Remove item from the buffer.
  3. Release lock (up on S).
  4. Decrement the count of items (N).

✅ Truth Evaluation of Statements

#	Statement	Verdict	Reasoning
1	*Producer can add an item, but consumer can never consume it.*	**False**	Consumer can acquire S once the producer releases it; it then removes the item and updates N.
2	*Consumer will remove no more than one item from the buffer.*	**False**	After the first consumption, S is set to 1 again, allowing the consumer to acquire it repeatedly and consume multiple items.

3	*Deadlock occurs if the consumer acquires semaphore S when the buffer is empty.*	**True**	With N = 0, the consumer blocks on S; if the producer is also blocked, both processes are deadlocked.
4	*Starting value of semaphore N must be 1 (not 0) for deadlock-free operation.*	**True**	Initializing N to 1 prevents the consumer from blocking before any item is produced, breaking the deadlock cycle.

🛑 Deadlock Scenario

1. Initial values: $S = 1, N = 0$.
2. Consumer executes down(S) → $S = 0$.
3. Consumer checks $N = 0$ and blocks (cannot consume).
4. Producer attempts down(S) but finds $S = 0$ and blocks.
5. Both processes are blocked → **deadlock**.

✅ Correct Answers

- **Question 1:** option **C** (statement is false).
- **Question 2:** options **C** and **D** (both statements are true).

📖 Reader-Writer Problem Overview

> **Critical Section:** region of code that accesses a shared resource and must not be concurrently executed by conflicting processes.

> **Mutex (binary semaphore):** a semaphore initialized to 1, used to enforce mutual exclusion.

> **Starvation:** situation where a process (e.g., a writer) is perpetually denied access because other processes (e.g., readers) keep entering the critical section.

Core Requirements

- **Multiple readers** may be in the critical section simultaneously.
- **Only one writer** may be in the critical section at any time.
- **Reader and writer must never coexist** in the critical section.

Typical Problems

Problem	Description
Reader-Writer Conflict	A writer modifies data while a reader reads, leading to inconsistent views.
Writer Starvation	Continuous arrival of readers prevents any writer from ever entering.
Reader Starvation (possible in some policies)	Writers dominate the critical section, blocking readers indefinitely.

🔐 Semaphore-Based Solution Sketch

1. **Semaphores used**
   - mutex (binary) – protects the **read-count** variable.
   - wrt (binary) – ensures exclusive access for writers.
2. **Shared variables**
   - readCount – number of active readers (initially 0).

3. **Reader Protocol**

```
down(mutex)          // lock readCount update
readCount = readCount + 1
if readCount == 1:
    down(wrt)        // first reader blocks writers
up(mutex)            // release readCount lock

// ---- critical section (reading) ----

down(mutex)
readCount = readCount - 1
if readCount == 0:
    up(wrt)          // last reader releases writer lock
up(mutex)
```

4. **Writer Protocol**

```
down(wrt)            // exclusive access
// ---- critical section (writing) ----
up(wrt)              // release lock
```

How the Protocol Prevents Starvation

- **Writer priority** can be added by checking a waiting-writer flag before allowing a new reader to acquire wrt.
- In the basic version above, readers may starve writers if readers keep arriving; a **fairness** modification (e.g., a turnstile semaphore) can be introduced to balance access.

Leader-Reader Concept

- The **first reader** that enters acts as the **leader**: it obtains wrt to block writers.
- Subsequent readers only modify readCount.
- The **last reader** (when readCount drops to 0) releases wrt, signaling that writers may proceed.

📊 Comparison: Reader-Writer vs. Producer-Consumer

Aspect	Producer-Consumer	Reader-Writer
Resource	Bounded buffer	Shared database/file
Concurrency rule	One producer and one consumer synchronize via two semaphores (S, N).	Multiple readers allowed; only one writer at a time; readers ↔ writers mutually exclusive.
Typical deadlock cause	Consumer blocks on empty buffer while producer blocks on full buffer.	Both waiting on each other when S = 0 and N = 0 (empty buffer) or when readers continuously occupy the section.
Starvation risk	Rare; depends on scheduling.	Writers can starve if readers keep arriving; readers can starve under writer-priority policies.
Solution primitives	Two counting semaphores (S, N).	Mutex + binary semaphore (wrt) + read-count variable.

🧩 Key Takeaways

> **Deadlock-free initialization:** setting the item-count semaphore (N) to 1 (instead of 0) prevents the consumer from blocking before any item exists, breaking the circular wait condition.

> **Reader-writer synchronization:** use a **mutex** to protect readCount and a **writer semaphore** (wrt) to ensure exclusive write access while allowing concurrent reads.

> **Starvation mitigation:** incorporate additional coordination (e.g., a turnstile or waiting-writer flag) to give writers a chance to enter when they are waiting.

📚 Reader-Writer Problem with Semaphores

> **Reader-Writer Problem** – A classic synchronization issue where *readers* may access a shared database concurrently, but *writers* require exclusive access.

🔐 Semaphores Employed

Semaphore	Purpose	Used By
mutex	Protects the **readers count** RC from race conditions	Readers only
DB	Grants exclusive access to the **critical section** (the database)	Both readers and writers

🧱 Key Variables

- RC – Integer counting the number of readers currently inside the critical section.
- mutex – Binary semaphore (initial value 1) for mutual exclusion on RC.
- DB – Binary semaphore (initial value 1) representing the lock on the database.

👓 Reader Algorithm

1. **Down(mutex)** – acquire exclusive right to modify RC.
2. Increment RC.
3. **If RC == 1** → this reader is the *first* (leader):
    - **Down(DB)** – lock the database so no writer can enter.
4. **Up(mutex)** – release the mutex; other readers may now change RC.
5. **Read** the database (critical section).
6. **Down(mutex)** – re-acquire mutex to update RC.
7. Decrement RC.
8. **If RC == 0** → this reader is the *last*:
    - **Up(DB)** – unlock the database, allowing writers to proceed.
9. **Up(mutex)** – finish.

> **First Reader (Leader)** – The reader that finds RC transitioning from 0→1; it locks the database on behalf of all readers.
> **Last Reader** – The reader that brings RC back to 0; it unlocks the database for writers.

✍️ Writer Algorithm

1. **Down(DB)** – attempt to lock the database.
    - If a reader (or another writer) holds the lock, the writer blocks.
2. **Write** to the database.
3. **Up(DB)** – release the lock.

📊 Comparison of Actions

Action	Reader (first)	Reader (intermediate)	Reader (last)	Writer
Acquire mutex	✔	✔	✔	✖
Modify RC	++	–	--	–

Acquire DB	✔ (when RC becomes 1)	–	–	✔ (if free)
Release mutex	✔	–	✔	–
Release DB	–	–	✔ (when RC becomes 0)	✔ (after writing)

🚦 Important Cases

- **Writer arrives while readers are inside** → blocked on DB until the last reader releases it.
- **Multiple readers arrive simultaneously** → serialized updates to RC via mutex; only the first reader locks DB.
- **No readers present** → a writer can lock DB immediately.

🍝 Dining Philosophers Problem

> **Dining Philosophers Problem** – A concurrency scenario where *n* philosophers share *n* forks (or knives) placed between them; each philosopher needs two adjacent forks to eat.

🧩 System Model

- **Philosophers**: Processes (P_0, P_1, \dots, P_{n-1}) (with (n \ge 2)).
- **Resources**: One fork (or knife) between each pair of neighboring philosophers; thus each fork is shared by two philosophers.
- **Behavior**:
  1. **Think**
  2. **Hungry** → try to pick up **left fork**, then **right fork**.
  3. **Eat** (only if both forks acquired).
  4. **Put down** both forks and return to thinking.

⚙️ Resource Acquisition Sequence

Step	Philosopher (P_i)	Action
1	Pick up **left** fork (shared with (P_{i-1})).	May succeed or block.
2	Pick up **right** fork (shared with (P_{i+1})).	Requires left fork already held; if unavailable, blocks.
3	Eat (both forks held).	Critical section.
4	Release both forks.	Enables neighbors to proceed.

> **Successful Eating** – Occurs only when both left and right forks are obtained without deadlock.

🛑 Deadlock Situation

- If every philosopher simultaneously picks up their **left** fork, each holds one fork and waits for the right fork, which is held by the neighbor.
- This creates a **circular wait**:

$$P_0 \to P_1 \to P_2 \to \cdots \to P_{n-1} \to P_0$$

- No philosopher can proceed, and the system is stuck indefinitely.

🔄 Cycle Illustration (for (n = 5))

1. (P_0) holds fork 0, waits for fork 1.
2. (P_1) holds fork 1, waits for fork 2.
3. (P_2) holds fork 2, waits for fork 3.
4. (P_3) holds fork 3, waits for fork 4.
5. (P_4) holds fork 4, waits for fork 0.

All five processes are in a **wait-for cycle**, constituting a deadlock.

📌 Key Takeaways

- The problem exemplifies **resource contention** and the need for protocols that break the circular wait (e.g., ordering resource acquisition, using a host-mediated solution, or limiting the number of concurrent eaters).
- Understanding the deadlock cycle is essential for designing correct synchronization mechanisms.

🔪 Dining Philosophers Problem Overview

🔁 How Deadlock Occurs

> **Deadlock** – a situation where each process in a set is waiting for a resource held by another process in the same set, forming a circular wait.

- All philosophers become **hungry** simultaneously.
- Each philosopher picks up the **left fork** first.
- Because every left fork is already held, every philosopher then waits for the **right fork**, which is held by its neighbor.
- This creates a circular wait chain (e.g., $P_0 \to P_1 \to P_2 \to \ldots \to P_0$), resulting in a deadlock.

📋 Implementation Sketch

```
void philosopher(int i) {
    while (true) {
        think();                            // philosopher is thinking
        takeFork(i);                        // pick up left fork
        takeFork((i + 1) % N);              // pick up right fork (mod N for circular table)
        eat();                              // eat when both forks are held
        putFork(i);                         // release left fork
        putFork((i + 1) % N);               // release right fork
    }
}
```

- **N** denotes the number of philosophers (e.g., N = 6).
- Modulo operation (% N) implements the circular arrangement of forks.

📈 Maximum Concurrency

> **Concurrency** – the number of philosophers that can eat simultaneously without causing deadlock.

Number of Philosophers (N)	Maximum Philosophers that Can Eat
5	2
6	3
3	1

- For **N = 5**, only two philosophers can hold two forks each without conflict.
- For **N = 6**, three non-adjacent philosophers can eat concurrently.
- With **N = 3**, only one philosopher can eat at a time because each needs two of the three forks.

🛡️ Deadlock Prevention

❌ Non-Semaphore Approach (Ordering Strategy)

- **Rule:** Let **N-1** philosophers pick the **left fork first**, then the right fork.
- Let the remaining **one philosopher** pick the **right fork first**, then the left fork.
- This breaks the circular wait because at least one philosopher can always obtain both forks.

Two-philosopher example

- Philosopher P_1 picks left → right.
- Philosopher P_2 (the odd one) picks right → left.
- If P_1 holds its left fork, P_2 will block on the same fork, allowing P_1 to acquire the right fork and eat. After P_1 releases both forks, P_2 proceeds.

🧿 Semaphore-Based Solution (Brief)

- Introduce a **semaphore** to control access to forks.
- Each fork acquisition (takeFork) and release (putFork) is performed inside wait/signal operations on the semaphore, ensuring that no more than N-1 philosophers can attempt to pick up forks simultaneously, thus eliminating the possibility of a circular wait.

💈 Sleeping Barber Problem

> **Sleeping Barber Problem** – a classic synchronization problem describing a barber shop with one barber, a limited number of waiting chairs, and arriving customers.

- **Barber** sits in his chair and **sleeps** when there are no customers.
- When a **customer** arrives while the barber is sleeping, the customer **wakes the barber**.
- The barber then serves the customer; any additional customers either wait in the chairs or leave if no chairs are free.

Deadlock in Barber Shop 🪑

> **Deadlock** occurs when a customer does **not vacate** the only available chair after a haircut, preventing other customers from proceeding.
>
> - If **no waiting chairs** are free, the arriving customer **leaves**.
> - The system can reach a state where **all chairs are occupied** and **no progress** can be made, which is the deadlock condition.

Sequential Execution vs Concurrency 🔄

> **Sequential execution** runs statements one after another in a fixed order.

> **Concurrent execution** allows **independent** statements to run **together** (not necessarily at the exact same instant).

Micro-operations for a statement

Step	Operation
1	Load the value of **B** into a register
2	Load the value of **C** into a register
3	Add the two registers
4	Store the result into **A**

The above sequence shows how a single statement like $a = B + C$ is executed **sequentially** at the micro-operation level.

Determining Concurrent Statements ✅

Two statements are **concurrent** when **neither statement's output is used as the other's input**.

Dependency analysis

Statements	Dependency	Can run concurrently?

S_1: $a = B + C$ S_2: $d = e$; CR; F	Independent	✅
S_1 and S_3	Output of S_1 is input to S_3	❌
S_3 and S_4	Output of S_3 is input to S_4	❌
S_2 and S_3	Independent	✅
S_4 and S_5	Output of S_4 is input to S_5	❌
S_5 and S_6	Independent	✅
S_3 and S_7	Dependent	❌

Example precedence graph

- **First layer:** S_1, S_2 (no ordering between them)
- **Second layer:** S_3 (depends on S_1)
- **Third layer:** S_4 (depends on S_3)

The graph illustrates that **only statements without incoming edges** can be executed concurrently.

Concurrency vs Parallelism ⚡

> **Concurrency** = *multiprogramming* – the system can support **two or more actions in progress** at the same time (progress).

> **Parallelism** = *multiprocessing* – the system can **execute two or more actions simultaneously** (execution).

Aspect	Concurrency (Multiprogramming)	Parallelism (Multiprocessing)
What it means	Multiple processes share the **CPU** over time	Multiple CPUs/cores run processes **at the same instant**
Analogy	One person **dealing** with snack-eating, lecture-watching, music-listening together	Three people **doing** each activity **simultaneously**
Implementation	Single-CPU interleaved execution (time-slicing)	Multiple CPUs or cores executing in true parallel
Key word	*Progress*	*Execution*

Key insight: Concurrency provides the **possibility** of parallelism; parallelism is a **realization** of concurrent actions on multiple processors.

Types of Concurrency: Pseudo vs Real 🖥️

> **Pseudo-concurrency** (also called *interleaved execution*) occurs on a **single CPU** where the illusion of simultaneous actions is created by rapidly switching between processes.

> **Real concurrency** (or *true parallelism*) occurs on **multiple CPUs/cores**, where actions truly happen at the same time.

- In pseudo-concurrency, the output may show interleaved results (e.g., blue and red outputs alternating), giving the impression of simultaneity.
- In real concurrency, both outputs can be observed **simultaneously** because separate hardware executes them.

Reading Set and Writing Set 📚

> **Reading set** – the collection of variables **read** by a statement.
> **Writing set** – the collection of variables **written/updated** by a statement.

Example statements

Statement	Reading set	Writing set
$a = B + C$	B, C	a
$d = e; \text{CR}; F$	e, F	d
$a += B \times C$ (expanded to $a = a + B \times C$)	a, B, C	a, B, C (all three are updated)

- If two statements **share a variable** in their **writing sets**, they **cannot** be concurrent.
- If one statement's **writing set** intersects another's **reading set**, they are also **dependent** and must be ordered.

📈 Increment Operators and Evaluation Order

- **Post-increment (b++)**

  > *Uses the current value of **b**, then increments **b** by 1.*

- **Pre-increment (++b)**

  > *Increments **b** by 1 first, then uses the new value.*

- Example expression (interpreted from the transcript):

  $a = a + + + b;; c - -;$

  - ++b is evaluated **before** it is added to a.
  - c-- uses the current value of c and then decrements it.

📚 Read and Write Sets

Statement	**Read Set** (variables whose values are accessed)	**Write Set** (variables whose values are updated)
scanf("%d", x);	— (no variable read)	x
printf("%d", x);	x	—
Assignment x = 5;	—	x
General rule	If a variable appears on the **right-hand side**, it is in the **read set**.	If a variable appears on the **left-hand side**, it is in the **write set**.

- When a variable is **declared** without initialization, it holds an undefined **garbage value** until it is assigned (e.g., via input).

🔗 Burnstein Concurrency Condition

> *Two statements S_i and S_j are **concurrent** if none of the following intersections are non-empty:*

- $\text{Read}(S_i) \cap \text{Write}(S_j) = \varnothing$
- $\text{Write}(S_i) \cap \text{Read}(S_j) = \varnothing$
- $\text{Write}(S_i) \cap \text{Write}(S_j) = \varnothing$

If any intersection is non-empty, the statements **cannot** execute concurrently.

🧩 Concurrency Constructs: pair begin / pair end

- **pair begin … pair end** (also written as co-begin / co-end) denotes a **parallel block**: all statements inside may execute **concurrently**.
- **begin … end** denotes a **sequential block**: statements execute **in order**.

Example Layout

```
pair begin
    S1
    S2
    S3
pair end
S4
```

- S1, S2, S3 are placed on the **same level** of the precedence graph (no ordering constraints).
- S4 follows after the parallel block, so it depends on the completion of **all** of S1-S3.

📊 Building Precedence Graphs

1. **Identify root node** (first statement).
2. **Apply parallel blocks** (pair begin … pair end) to create **branching** nodes that have no edges between them.
3. **Apply sequential blocks** (begin … end) to create **ordered edges**.

Sample Graph Construction

- **Root**: S1
- **Parallel block**: pair begin → S2, S3, S4 → pair end
- **Sequential after parallel**: S5, S6 (inside a normal begin … end)
- **Further sequential**: S7, S8, S9 (another begin … end)
- **Final statement**: S10

Resulting precedence graph:

- S1 → {S2, S3, S4} (concurrent)
- {S2, S3, S4} → S5 → S6 (sequential)
- S6 → S7 → S8 → S9 (sequential)
- S9 → S10

⚙️ Implementing Concurrency with Binary Semaphores

- Define a **binary semaphore** for each dependency edge (e.g., a, b, c, d, e, f, g).
- Initialize all semaphores to **0**.
- A statement **cannot proceed** until the semaphore(s) corresponding to its required predecessor(s) are **signaled** (set to 1).

Semaphore	Purpose
a	Controls entry to statement dependent on predecessor A
b	Controls entry to statement dependent on predecessor B
…	…

Execution pattern (conceptual):

1. **Initialize** all semaphores to 0.
2. **Signal** a semaphore when its predecessor finishes.
3. **Wait** on the required semaphore(s) before a statement begins.

If a graph contains **circular dependencies** or missing edges that prevent a clear ordering, it may be **non-implementable** using only pair begin / pair end and binary semaphores. In such cases, additional synchronization mechanisms (e.g., counting semaphores or explicit barriers) are required.

🛠️ Common Pitfalls

- **Assuming concurrency**: Two statements that appear in different branches are *not* automatically concurrent if one depends on the other's output (e.g., S6 depends on S3).
- **Missing edges**: Omitting a dependency edge can make a graph **non-implementable** because the required order cannot be enforced.
- **Garbage values**: Declaring a variable without initialization leads to undefined content until a **write** (e.g., input) occurs.

🔐 Semaphore Operations: VA and PA

> **VA (V operation)** – *increments* the value of a binary semaphore (acts as a **key**).
> **PA (P operation)** – *decrements* the semaphore value (acts as a **lock**).

- Initial value of each binary semaphore is **0**.
- A statement can only execute when the required semaphore value is **1** (unlocked).

📋 Creating Execution Sequences with Semaphores

Step	Action	Semaphore Effect	Result
1	Execute **S1**	**VA** on semaphore **a** → $a = 1$	Unlocks subsequent statements that depend on **a**.
2	Execute **PA** (after S1)	**PA** on **a** → $a = 0$	Locks statements that require **a** to be 1.
3	Execute **S2** or **S3**	Both require the key obtained from **S1**	Choice is allowed because **S2** and **S3** are concurrent.
4	After **S2** → execute **S4**	**VA** on semaphore for **S5** and **S6**	Provides keys for **S5** and **S6**.
5	After **S4** → can acquire keys for **S5** (via **VC**) and **S6** (first lock).		
6	Execute **S5** → unlocks first lock of **S7**.		
7	Execute **S3** → provides second key for **S6**.		
8	After both locks of **S6** are satisfied → unlock second lock of **S7**.		
9	Execute **S7** → program ends.		

Key Points

- **Sequence enforcement**: The first statement (**S1**) must run to set the initial key; otherwise later statements are blocked.
- **Concurrency**: After **S1**, the process may choose either **S2** or **S3** (or both concurrently).
- **Multiple locks**: Some statements (e.g., **S6**, **S7**) require **two distinct keys** before they can execute.

🗺️ Graph Representation of Statements

- **Nodes**: Statements **S1** … **S9**.
- **Edges** indicate required execution order (lock → key).

From → To	Relationship
S1 → **S2**, **S3**	Key for both; concurrent choice.
S2 → **S4**	Sequential.

S3 → S4	Sequential (part of same parallel block).
S4 → S5, S6	Provides keys for both.
S5 → S7 (first lock)	Dependency.
S6 → S7 (second lock)	Dependency.
S7 → S9	Final statement.
S8 is sequential after S6 (not detailed in transcript but mentioned as sequential).	

Execution Flow Example

1. Begin → S1 (acquire key).
2. Choose S2 → S4 → acquire keys for S5 & S6.
3. Optionally execute S3 (provides second key for S6).
4. Execute S5 → unlock first part of S7.
5. Execute S6 (both locks satisfied) → unlock second part of S7.
6. Execute S7 → End.

📐 Semaphore Count Summary

Statement	Required Keys (Locks)
S1	0 (initial start)
S2	1 (key from S1)
S3	1 (key from S1)
S4	1 (key from S2 or S3)
S5	1 (key from S4)
S6	2 (first key from S4, second from S3)
S7	2 (first key from S5, second from S6)
S8	1 (key from S6)
S9	1 (key from S7)

🤝 Concurrent Functions p and q

- Functions: void p(a,b,c) and void q(d,e).
- Both can be executed concurrently inside a pair begin ... pair end block.

Valid Execution Sequences

1. P → Q (complete p before starting q).
2. Q → P (complete q before starting p).
3. Interleaved:
    - Run part of p, switch to q, then return to p.
    - Run part of q, switch to p, then return to q.

> *Any interleaving that respects the internal order of each function is permissible because p and q are independent.*

🧩 Homework Insight

- Current implementation uses seven binary semaphores (sem1 ... sem7).
- Challenge: Reduce the number of semaphores while preserving the same execution constraints.
- Strategy: Identify semaphores that can be combined or eliminated without breaking required lock-key relationships.

Sequence Validity 🔢

Precedence Constraints

> **C cannot appear before A** and **E cannot appear before D**.

- These constraints must hold for any valid execution order of the operations **A, B, C, D, E**.

Example Sequences

Sequence	Valid?	Reason
a b c d e	✅	All constraints satisfied; **P** executes before **Q**.
a d b e c	✅	Order respects that **C** follows **A** and **E** follows **D**.
d c e b a d	❌	**C** appears before **A** (violates the first constraint).
a e b d c	❌	**E** appears before **D** (violates the second constraint).
a b d e c	✅	**C** still follows **A** and **E** follows **D**.

Sequences 1, 2, and 5 satisfy the constraints.

Concurrency Example: X and Y Variables 📊

Code Overview

- Initial state: x = 0; y = 0
- **Part 1** (executed by one thread): x = 1; y = y + x
- **Part 2** (executed by another thread): y = 2; x = x + 3

Possible Interleavings

1. **Part 1 → Part 2**
   - x = 1 → y = y + x (y becomes 1) → y = 2 → x = x + 3 (x becomes 4) → final **(x, y) = (4, 1)** (invalid because y is overwritten later).
2. **Part 2 → Part 1**
   - y = 2 → x = x + 3 (x becomes 3) → x = 1 (overwrites x) → y = y + x (y becomes 3) → final **(x, y) = (1, 3)**.
3. **Interleaved** (x = 1; y = 2; y = y + x; x = x + 3)
   - After x = 1 → y = 2 → y = y + x (y becomes 3) → x = x + 3 (x becomes 4) → final **(x, y) = (4, 3)**.
4. **Interleaved** (y = 2; x = 1; x = x + 3; y = y + x)
   - y = 2 → x = 1 → x = x + 3 (x becomes 4) → y = y + x (y becomes 6) → final **(x, y) = (4, 6)**.

Resulting Final Values

Possible Final (X, Y)	Execution Pattern
(1, 3)	Part 2 finishes before the y = y + x step of Part 1.
(4, 6)	y = 2 executes first, then x = x + 3, followed by y = y + x.
(4, 3)	x = 1 executes before y = 2, then y = y + x before x = x + 3.

The second and third patterns correspond to the answer "second and third" in the original question.

Binary Semaphore Synchronization 🔐

Initial Conditions

- int x = 0; y = 20;
- Binary semaphores **MX** and **VMX** are both initialized to **1** (mutual-exclusion enabled).

Critical Sections

Block	Operations (protected by semaphore)
P1	P(MX); x = x + 1; V(MX);
P2	P(VMX); x = y + 1; V(VMX);

Execution Orders & Final X

1. **P1 → P2**
   - After P1: x becomes **1**.
   - P2 reads y = 20, computes x = 20 + 1 = 21.
   - Final **x = 21**.
2. **P2 → P1**
   - After P2: x becomes **21** (using initial y = 20).
   - P1 increments: x = 21 + 1 = 22.
   - Final **x = 22**.

> Because the binary semaphore value **1** forces a **sequential** execution of the two critical sections, only the two orders above are possible.

Possible final values of X: 21 or **22**.

Fork and Join Synchronization 🚀

Fork Definition

> **Fork** splits the control flow into two concurrent paths: the statement **immediately after the fork** and the statement **targeted by the fork's label**.

Join Definition

> **Join** synchronizes concurrent paths by using a **counter** (count). Each path decrements count; when count reaches **0**, execution continues past the join point.

Typical Pattern

1. count = 2; *(two paths will meet)*
2. fork L; *(creates concurrency)*
3. **Path A** – statement **S2** (right after fork).
4. **Path B** – statement **S3** (label L).
5. Both paths eventually execute join: *(decrements count)*

Example Flow

Step	Action	Effect
1	count = 2;	Indicates two concurrent edges.
2	fork L;	Starts **S2** and **S3** concurrently.

3	S2 executes → goto X; (explicit jump to join).	
4	S3 executes → next statement is the join label X (implicit jump).	
5	join: → count = count – 1;	Each path signals completion.
6	When count == 0 → control proceeds to the statement after the join (e.g., S4).	

*The count variable ensures that the program does not continue until **both** concurrent branches have finished.*

Fork Construct 🔀

> **Fork** – a control-flow statement that splits execution into parallel branches.

- The **first branch** is the *immediate statement* following the Fork.
- The **second branch** is the *target label* of the Fork (e.g., L).
- When multiple forks are needed, chain them: each additional Fork creates another parallel path.

Fork Example with Statements S1-S7

Fork Label	Immediate Statement	Target Label	Parallel Branches Created
L	S2	S3	S2 and S3
K	S5	S6	S5 and S6

- After S2 the control goes to S4.
- After S5 and S6 a **join** is required (see next section).

Three-Way Fork (P1 Example)

1. Set **count** to the number of desired branches (3).
2. Use two nested Fork statements:
   - Fork L → creates branches S1 and S2.
   - Inside the second branch, another Fork creates the third branch.
3. Resulting concurrent paths: P2, P3, P4.

Join Construct 🔗

> **Join** – a synchronization point where parallel branches converge into a single flow.

- The **join node** is the statement **immediately after** the join keyword.
- All incoming edges must reach this point before execution proceeds.

Join Example with Statements S1-S7

Join Label	Incoming Branches	Next Statement After Join
Z	S5, S6, S3	S7 (final node)
K	P2, P3, P4	P5 (first statement after the join)

- After S5 → go to label Z.
- After S6 → also go to Z.
- After S3 → go to Z.

Key Points to Remember

- **Fork** creates two branches: the immediate successor and the labeled target.

- **Join** merges all incoming branches at the node that follows the join statement.
- The **count** of a fork equals the number of parallel branches you intend to create.

Process Synchronization with Semaphores 🔒

> **Semaphore** – an integer variable used to control access to shared resources; operations are wait (decrement) and signal (increment).

> **Deadlock** – a state where a set of processes are each waiting for resources held by the others, causing indefinite blocking.

> **Atomic operation** – an indivisible action that cannot be interrupted; non-atomic increments may interleave.

Scenario Overview

- Semaphore S (named **sema4**) initialized to 5.
- Shared counter c initialized to 0.
- Function PA performs:
  1. **Load** c.
  2. **Increment** c (non-atomic).
  3. **Signal** S twice.
- Five threads (T1 … T5) invoke PA concurrently.

Possible Outcomes

Outcome	Conditions	Explanation
Deadlock	All threads execute wait on S until it reaches 0 and then block on the next wait.	After five successful waits, S = 0. Subsequent waits cannot proceed, leaving all threads blocked.
Counter = 5	Threads run **sequentially** (no interleaving).	Each thread completes its increment before the next starts, so c is incremented five times.
Counter = 1	One thread's increment is pre-empted **after load but before store**; the other four complete fully.	The pre-empted thread stores the old value ($0+1$) after others have already set c to 4, overwriting it with 1.

Detailed Walkthrough

1. **Deadlock Situation**
   - T1 executes wait → S = 4.
   - T2 executes wait → S = 3.
   - … up to T5 → S = 0.
   - Any thread attempting another wait blocks; since each thread still needs a second wait (two signals per PA), all remain blocked.
2. **Counter =** 5 **(Sequential Execution)**
   - T1 completes entire PA: c = 1.
   - T2 runs next: c = 2.
   - … after T5: c = 5.
3. **Counter =** 1 **(Interleaved Execution)**
   - T1 loads c = 0.
   - Pre-empted before store.
   - T2–T5 each complete: c becomes 4.
   - T1 resumes, stores c = 1 (its old value + 1), overwriting 4.

Summary of Rules for Fork & Join

Construct	Remember This	How to Apply
Fork	*Two split points*: immediate successor **and** labeled target.	Set count = number of branches. Use successive forks for >2 branches.
Join	*Merge point* is the statement **after** the join.	Send each branch to the join via goto/label; all edges converge at the next node.

These notes capture the essential control-flow mechanics of Fork/Join and the semaphore-based synchronization examples, including the conditions that lead to deadlock and the possible final values of a shared counter.

🧵 Process Synchronization: Counter Values with Threads

> **Definition:** *The final value of a shared counter after concurrent thread execution depends on the order in which the threads perform their store operations.*

- Possible final values: **0, 1, 2, 3, 4, 5**
- Scenarios illustrating each value:

Final Value	Execution Order (example)
0	All threads store 0 after completing; no thread increments the counter.
1	One thread stores 1 after all others have stored 0.
2	Two threads store 1 after the others have stored 0.
3	Three threads store 1 after the others have stored 0.
4	Four threads store 1 after the remaining thread stores 0.
5	Every thread stores 1 (all increment the counter).

The value depends on how the scheduler interleaves the store operations of threads **P1 … P5**.

🛑 Semaphore Question (Sem 4)

Scenario:

- Semaphore **S4** is initialized to **1**.
- One **wait** (P) and one **signal** (V) operation are removed.

> **Question:** *What are the possible values of the associated count after these modifications?*

(The transcript does not provide the answer; the problem is left for the student to solve.)

🔢 Incrementing a Shared Counter

```
int count = 0;
while (test) {
    // inside parallel region
    count++;  // repeated 5 times
}
```

A second call to test spawns another parallel region with the same five count++ operations.

- **Maximum value:**

    - When the two parallel regions execute **sequentially** (no overlap), each contributes 5 increments → 10.
- **Minimum value:**
    - The transcript states the minimum is **neither 5 nor 1**, but does not give the exact number.
    - Determining the minimum requires analyzing possible interleavings where increments may be lost due to race conditions.

🚧 Deadlock Overview

> **Definition:** *Deadlock occurs when two or more processes are each waiting for an event (typically the release of a resource) that will never occur because the other processes hold the needed resources and refuse to release them.*

Classic Example

- **Process P1** holds resource **R1** and requests **R2**.
- **Process P2** holds **R2** and requests **R1**.
- Both processes wait indefinitely → deadlock.

Consequences

- Reduced **throughput** and **efficiency**.
- **Ineffective utilization** of resources.

⚖️ Deadlock vs. Starvation

Aspect	Deadlock	Starvation
Blocking duration	**Infinite** (forever)	**Indefinite** (unknown, but may end)
Cause	Cyclic waiting among processes holding resources	Scheduling decisions that repeatedly bypass a process
Resource release	Processes never release held resources	Resources may eventually become available

---

🍳 Real-World Analogy: Rachel & Ross

- **Resources:**
    - Oil → **R1** (held by **Rachel**)
    - Pan → **R2** (held by **Ross**)
- **Processes:**
    - Rachel ↔ **P1** (needs **R2**)
    - Ross ↔ **P2** (needs **R1**)

Both are *adamant* (stubborn) and do not release their current resource, creating a deadlock cycle.

🖥️ System Model for Studying Deadlock

- **Processes:** $P_1, P_2, \ldots, P_n$
- **Resources:** $R_1, R_2, \ldots, R_m$ (hardware or software)

Resource Types

Type	Description
Single-instance	Only one copy of the resource exists.
Multi-instance	Multiple identical copies of a single resource.

🔑 Necessary Conditions for Deadlock

1. **Mutual Exclusion** – At least one resource must be non-shareable.
2. **Hold and Wait** – A process holds at least one resource while waiting for additional resources.
3. **No Preemption** – Resources cannot be forcibly taken away from a process.
4. **Circular Wait** – A closed chain of processes exists where each process waits for a resource held by the next process in the chain.

All four must hold simultaneously for a deadlock to arise.

🛠️ Hold-and-Wait Explained (Why It Alone Isn't Sufficient)

- **Scenario:**
  - P_i holds R_1 and requests R_2 .
  - P_j holds R_2 but **does not** request R_1 (it will release R_2 after use).
- **Outcome:** No deadlock because P_j eventually releases R_2, allowing P_i to proceed.

Thus, **hold-and-wait** is *necessary* but **not sufficient**; the other conditions must also be present.

📌 Deadlock Necessary Conditions

> **Deadlock** – a state where a set of processes are each waiting for resources held by another process in the set, and none can proceed.

1️⃣ Mutual Exclusion

> *A resource can be assigned to **only one** process at a time.*

2️⃣ Hold-and-Wait

> *A process **holds** at least one resource **while waiting** for additional resources.*

- **Implication:**
  - If *deadlock* occurs → **Hold-and-Wait** must be present.
  - Presence of **Hold-and-Wait** alone does **not** guarantee deadlock.

3️⃣ No Preemption

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

- Example metaphor: Ross (big brother) cannot snatch Rachel's oil bottle; otherwise deadlock could be avoided.

4️⃣ Circular Wait

> *There exists a **cycle** of processes where each process is waiting for a resource held by the next process in the cycle.*

- **Implication:**
  - *Deadlock* → **Circular Wait** must exist.
  - **Circular Wait** alone is **not sufficient** for deadlock (e.g., if one process can eventually release a needed resource).

Condition	Deadlock ⇒ Condition	Condition ⇒ Deadlock
Mutual Exclusion	✅	❌

Hold-and-Wait	✅	❌
No Preemption	✅	❌
Circular Wait	✅	❌

🔄 Resource Allocation Graph (RAG)

🎯 Vertices

- **Processes** – drawn as circles labeled P_1, P_2, ...
- **Resources** – drawn as squares/rectangles labeled R_1, R_2, ...
  - **Single-instance** resource: one dot inside the rectangle.
  - **Multiple-instance** resource: multiple dots (e.g., three dots = three instances).

🔗 Edge Types

Edge	Direction	Meaning
Claim edge (dotted)	*Process → Resource*	Process **may request** the resource in the future.
Request edge	*Process → Resource*	Process is **currently requesting** the resource.
Assignment edge	*Resource → Process*	Resource has been **allocated** to the process.

- **Multiple edges** between a process and a resource indicate **multiple instances** being requested or assigned.

📊 Interpreting a Sample RAG

Process	Holds (Assignment)	Requests (Request edge)
P_1	R_2 (1 instance)	R_1
P_2	R_1, R_2 (2 instances)	R_3
P_3	R_3	–
P_4	R_4	–

- **Execution flow:**
  1. **P_3** runs (has all needed resources) → completes → releases **R_3**.
  2. **R_3** becomes available → **P_2** can acquire it, then releases **R_1** and **R_2** after finishing.
  3. **R_1** becomes free → **P_1** obtains it and proceeds.
- **Result:** No deadlock; at least one process (**P_3**) can progress and break the cycle.

⚙️ Detecting Deadlock with a RAG

1. **Check for cycles** – a necessary condition.
2. **Verify if every process in the cycle is blocked** (no process can proceed).
   - **All blocked** → deadlock.
   - **At least one can run** → it will eventually release resources, breaking the cycle → **no deadlock**.

Decision Flow

Situation	Deadlock?
Cycle present **and** all involved processes blocked	✅
Cycle present but at least one process can run (or will run)	❌

No cycle	❌

🧩 Example Scenarios

Example 1 – Circular Wait with a “non-admin” process

- **John** needs a screwdriver from **Lisa**.
- **Mary** needs pliers from **Ken**.
- **Ken** needs a wrench from **Lisa**.
- **Lisa** needs the screwdriver from **John**.

If any one of the four does **not** participate in the circular claim (e.g., John refuses), the cycle is broken and deadlock does **not** occur. Hence, **circular wait is necessary but not sufficient**.

Example 2 – Adding an Edge that Causes Deadlock

Process	Holds	Requests
P_1	R_2	R_1
P_2	R_1, R_2	R_3
P_3	R_3	R_2
P_4	R_4	–

- **P_1, P_2, P_3** each wait for a resource held by another in the set → all three are blocked.
- **P_4** runs independently, releases **R_4**, which is **not needed** by the blocked trio.
- Since **no process outside the cycle can free a needed resource**, the system is in deadlock.

🔁 Summary of Implication Relationships

> *Necessary conditions are **required** for deadlock but **do not guarantee** it.*
> *Deadlock **implies** all four conditions, while each condition **alone** does **not** imply deadlock.*

Statement	Truth Value
“If deadlock is present, hold-and-wait is present.”	✅
“If hold-and-wait is present, deadlock will occur.”	❌
“If deadlock is present, circular wait is present.”	✅
“If circular wait is present, deadlock will occur.”	❌
“If a critical section exists, deadlock must exist.”	❌
“If deadlock exists, a critical section must exist.”	✅

Deadlock Handling Strategies 🛡️

> **Deadlock** – a state in which a set of processes are each waiting for resources held by the others, so none can proceed.

- **Prevention** – eliminates one or more necessary conditions for deadlock, ensuring it never occurs.
- **Avoidance** – dynamically decides whether to grant resource requests based on a safe-state analysis.
- **Detection & Recovery** – allows deadlock to form, then detects it and takes actions to break the cycle.
- **Ignorance (Ostrich Algorithm)** – does nothing; if a deadlock occurs the system is simply restarted.

Strategy	When Used	Typical OS Example

Prevention	Critical systems where any pause is unacceptable (e.g., air-traffic control).	Real-time OS with strict resource management.
Avoidance	Systems that can afford runtime analysis of resource requests.	Systems using Banker's algorithm.
Detection & Recovery	General-purpose OS where deadlocks are rare but must be handled.	Windows, Unix, Linux (use detection algorithms).
Ignorance	Environments where deadlocks are extremely rare and prevention cost is high.	Most commercial OS (rely on restart).

Deadlock Prevention Techniques 🔧

Negating Necessary Conditions ✅

A deadlock requires **all** of the following conditions to hold simultaneously:

Condition	Description
Mutual Exclusion	At least one resource is non-shareable.
Hold and Wait	A process holds some resources while waiting for others.
No Preemption	Resources cannot be forcibly taken from a process.
Circular Wait	A closed chain of processes exists, each waiting for a resource held by the next.

If any one condition is removed, deadlock cannot arise.

1. Mutual Exclusion

Impossible to eliminate in a multiprogrammed system because some resources (e.g., printers) must be exclusive.
→ No practical prevention method.

2. Hold and Wait

Two protocols can break this condition:

Protocol	How It Works	Drawbacks
Request-All-Up-Front	A process must request **all** resources it will ever need before it begins execution. If all are available, it proceeds; otherwise it waits.	• **Starvation** – a process may wait indefinitely for the exact combination of free resources. • **Inefficiency** – resources needed later are held idle during early phases.
Release-Before-Request	After completing a phase, a process releases **all** currently held resources before requesting any new ones.	• Still susceptible to starvation, but avoids the idle-resource problem of the first protocol.

3. No Preemption

Two strategies allow preemption of resources:

Strategy	Behavior	Impact
Forceful Preemption	A running process can forcibly take a needed resource from another process, regardless of the other's state.	May cause the preempted process to roll back or restart; can be "selfish."

Self-Preemption	A process voluntarily releases its held resources when it cannot obtain the additional ones it needs.	Encourages a “selfless” attitude; reduces deadlock risk but can still lead to starvation.

4. Circular Wait

Total-Order Resource Allocation prevents circular wait:

1. Assign a unique numerical label to every resource type.
2. Require each process to request resources **in increasing order of their labels**.
3. A process may never request a resource with a lower label than one it already holds.

This creates a strict hierarchy, breaking any possible cycle.

When to Choose Prevention vs. Ignorance 📊

- **Critical, time-sensitive systems** (e.g., air-traffic control, missile guidance, satellite control) cannot tolerate any pause.

  → **Deadlock Prevention** is mandatory; every necessary condition is carefully managed.

- **General-purpose operating systems** (Windows, Unix, Linux) experience deadlocks very rarely, and the overhead of full prevention is high.

  → **Deadlock Ignorance** (Ostrich algorithm) is acceptable; the system is simply rebooted if a deadlock occurs.

Detection & Recovery (Brief Overview) 🔍

- **Detection Algorithms** (e.g., Doctor’s algorithm) periodically examine the resource-allocation graph to find cycles indicating deadlock.
- **Recovery Actions** may include terminating one or more involved processes, preempting resources, or rolling back transactions.

> *Note:* Detailed steps of Doctor’s algorithm and Banker's algorithm are mentioned but not elaborated in the source material.

📊 Resource Ordering & Preemption

- Each resource is given a **unique ID** (e.g., A = 1, B = 2, C = 3, D = 4 …).
- **Rule:** a process may request only resources whose ID is **greater** than the highest ID it currently holds.

> **Definition**: *Linear ordering* – assigning a total order to resources and restricting requests to follow that order eliminates circular wait.

Example Flow

1. Process P receives resource C (ID = 3).
2. It may now request only IDs > 3 → can request **B** (ID = 8) ✔, **D** (ID = 4) ✖, **G** (ID = 7) ✖, **A** (ID = 1) ✖.
3. After obtaining ID 8, further requests must be > 8 → only **F** (ID = 10) or **E** (ID = 9) are allowed.

When a Lower-ID Request Occurs

- Suppose P holds IDs 3, 8, 10 and now requests ID 5.
- The OS **preempts** the higher-ID resources (10 and 8), grants ID 5, then later re-grants 8 and 10.
- This keeps the ordering linear but can cause **starvation**: other processes may be blocked while preempted resources are re-allocated.

🛡️ Deadlock Prevention – Which Conditions Can Be Negated?

Condition	Can it be removed?	How (protocols)

Mutual exclusion	❌ No	Must keep critical sections.
Hold and wait	✅ Yes	1️⃣ Request **all** needed resources at start. 2️⃣ Release all held resources before making a new request.
No preemption	✅ Yes	Preempt resources when a lower-ID request arrives (as above).
Circular wait	✅ Yes	Impose a global ordering on resource IDs (linear ordering).

- Protocol 1 (all-at-once) is **forceful**; Protocol 2 (release-before-new) is **self-imposed**.
- Preemption (protocol 3) may be **unsafe** if it leads to starvation.

📈 Resource Allocation Graph (RAG) Properties

Types of Resources

Resource Type	Cycle → Deadlock?	Reason
Multi-instance	**Necessary only**	Cycle may exist without deadlock; deadlock ⇒ cycle, but not vice-versa.
Single-instance	**Necessary & Sufficient**	Any cycle guarantees deadlock.
Mixed (single + multi)	**Necessary only**	Same reasoning as multi-instance.

> **Definition**: *Claim edge* – a dotted edge from a process to a resource indicating a possible future request.
> **Definition**: *Assignment edge* – a solid edge showing a resource currently held by a process.

Safe vs. Unsafe State

- **Safe state**: Adding an assignment edge (granting a request) **does not create a cycle** in the RAG. → No deadlock can occur.
- **Unsafe state**: Adding the edge would **introduce a cycle**. → Potential deadlock (not guaranteed until the request is actually made).

🧩 Deadlock Avoidance Algorithms

Algorithm	Resource Type	Basis
Resource Allocation Graph Algorithm	Single-instance	Uses **prior knowledge** (process declares all needed resources in advance).
Banker's Algorithm	Multi-instance	Also relies on **prior knowledge** of maximum future needs.

- **Prior knowledge**: each process must inform the OS of the complete set of resources it will require during its execution.

RAG Algorithm Steps (Single-Instance)

1. **Claim edges** are recorded for every declared future need.
2. When a process requests a resource, the OS attempts to convert the **claim edge** to an **assignment edge**.
3. **Check**: does this conversion create a **cycle**?
   - **No** → Grant the resource (stay in safe state).
   - **Yes** → Deny the request (remain safe; entering unsafe would risk deadlock).

🔄 Example Walkthrough

1. **Initial graph**:
   - P1 **holds** R1 (assignment edge).
   - P1 **claims** R2 (dotted edge).

- P2 **holds** R2 and **claims** R1.

2. **Current state**: No cycles → **Safe**.
3. **If R2 were assigned to P2** (converting P2's claim on R2 to an assignment):
   - New edge creates a **cycle** (P1 → R2 → P2 → R1 → P1).
   - System becomes **Unsafe** (deadlock would occur if the pending requests are fulfilled).
4. **When the actual request occurs** (e.g., P1 asks for R2):
   - Because the graph already has a cycle, granting the request leads to a **deadlock state** (not merely unsafe).

📚 Key Takeaways

- **Linear resource ordering** prevents circular wait but may require preemption, risking starvation.
- **Deadlock prevention** focuses on breaking any of the four Coffman conditions; the most practical are hold-and-wait avoidance, preemption, and ordering.
- In a **single-instance RAG**, any cycle equals deadlock; in **multi-instance**, a cycle is only a warning sign.
- **Safe state** = no cycles after granting; **unsafe state** = a cycle would appear, indicating potential deadlock.
- Both **RAG** and **Banker's** algorithms depend on processes providing **complete future resource requirements**.

Deadlock, Safe & Unsafe States 🚦

> **Deadlock** – a situation where a set of processes are each waiting for resources held by another process in the set, forming a circular wait, so none can proceed.

> **Unsafe State** – the system is **not** currently deadlocked, but there exists *some* allocation of future requests that could lead to deadlock.

> **Safe State** – the system can allocate resources to each process in some order (a **safe sequence**) such that every process can finish, guaranteeing no deadlock.

- **Cycle in a Resource Allocation Graph (RAG)** → indicator of potential deadlock.
- **Claim Edge** (future allocation) can become a **Request Edge** when a process actually asks for the resource.

Key Points

- A system may move **Safe → Unsafe → Deadlock** as more claim edges turn into request edges.
- An unsafe state is a *warning*; deadlock is the *realization* of that warning.

Resource Allocation Graph (RAG) Algorithm 🔗

- **Purpose:** keep the system in a **safe mode** by detecting cycles before they become actual deadlocks.

Element	Meaning
Claim Edge	Planned future allocation (not yet a request).
Request Edge	Active request from a process for a resource.
Cycle	Presence of a circular wait; when a claim edge becomes a request edge and completes a cycle, deadlock can occur.

- When a **claim edge** turns into a **request edge**, the OS checks the graph:
  1. If **no cycle** → request granted, system stays safe.
  2. If **cycle appears** → the system becomes **unsafe**; further granting may lead to deadlock.

Banker's Algorithm Overview 💰

The algorithm handles **multiple instances of multiple resources** and consists of two parts:

1. **Safety Algorithm** – determines if the current state is safe.

2. **Resource-Request Algorithm** – decides whether to grant a new request.

Core Variables

> N – number of processes.
> M – number of resource types.

Symbol	Description
$K_{i,j}$	*Maximum demand* – the maximum number of instances of resource j that process i may need.
$a_{i,j}$	*Allocation* – number of instances of resource j currently allocated to process i .
$B_{i,j}$	*Need* – remaining instances required: $B_{i,j} = K_{i,j} - a_{i,j}$.
$C_{i,j}(t)$	*Request* at time t – instances of resource j that process i is currently asking for (must satisfy $C_{i,j}(t) \leq B_{i,j}$).
Z_j	*Total* – total instances of resource j in the system.
e_j	*Available* – instances of resource j not allocated (must satisfy $e_j \leq Z_j$).

- **Total vs. Available:** $e_j = Z_j - \sum_{i=1}^{N} a_{i,j}$.

Safety Algorithm Details ✅

1. **Initialize**
   - $Work = \text{Available} = (e_1, e_2, \ldots, e_M)$.
   - $Finish[i] = \text{false}$ for all processes i .
2. **Find a Process**
   - Look for an i such that $Finish[i] = \text{false}$ and $B_{i,j} \leq Work_j$ for **all** j .
3. **If Found**
   - $Work = Work + a_i$ (release all resources of process i).
   - $Finish[i] = \text{true}$.
   - Repeat step 2.
4. **Termination**
   - If **all** $Finish[i]$ become true → **Safe State** (the order of finishing gives a safe sequence).
   - Otherwise → **Unsafe State**.

Example: Single Resource with Multiple Instances 📊

Scenario: 5 processes P_1–P_5 share one resource type R .

Process	Demand K_i	Allocation a_i	Need $B_i = K_i - a_i$
P_1	10	4	6
P_2	8	5	3
P_3	6	3	3
P_4	5	4	1
P_5	4	3	1

- **Total instances** $Z = 21$.
- **Allocated** $= 4 + 5 + 3 + 4 + 3 = 19$.

- **Available** $e = Z - \text{Allocated} = 2$.

Checking Safety

- With $e = 2$, no process's need ≤ 2 → **Unsafe State**.

If **Available** were $e = 3$ (e.g., total $Z = 22$ and allocated 19), the safety algorithm proceeds:

1. P_4 needs $1 \leq 3$ → allocate, finish, release 5 (its total demand).
   - New $Available = 3 - 1 + 5 = 7$.
2. P_5 needs $1 \leq 7$ → finish, release 4.
   - $Available = 7 - 1 + 4 = 10$.
3. P_3 needs $3 \leq 10$ → finish, release 6.
   - $Available = 10 - 3 + 6 = 13$.
4. P_2 needs $3 \leq 13$ → finish, release 8.
   - $Available = 13 - 3 + 8 = 18$.
5. P_1 needs $6 \leq 18$ → finish, release 10.
   - $Available = 18 - 6 + 10 = 22$.

All processes complete → **Safe State**.

Safe Sequence: $P_4 \to P_5 \to P_3 \to P_2 \to P_1$ (others possible).

Example: Multiple Resources with Multiple Instances (Partial) 🧩

Given at time T:

Process	Maximum Demand $(K_{i,a}, K_{i,b}, K_{i,c})$	Allocation $(a_{i,a}, a_{i,b}, a_{i,c})$
P_0	$(7,; 5,; 3)$	$(0,; 1,; 0)$
P_1	*(data not fully provided in transcript)*	—
P_2	*(data not fully provided)*	—
P_3	$(2,; 2,; 2)$	*(allocation not shown)*

- **Need** for P_0: $B_0 = (7 - 0,; 5 - 1,; 3 - 0) = (7,; 4,; 3)$.

The algorithm would use the same safety steps, comparing each process's need vector to the current **Available** vector (e_a, e_b, e_c) (where each $e_j \leq Z_j$).

Further calculations depend on the missing allocation/available values, but the method mirrors the single-resource example above.

🧱 Allocation, Need, and Available Resources

> **Allocation** – the number of resource instances that have already been given to a process.

> **Need** – how many more instances a process requires to satisfy its total demand; calculated as $\text{Need} = \text{Demand} - \text{Allocation}$.

> **Available** – the number of instances of each resource type that are currently free (not allocated).

Process	Demand (a, b, c)	Allocation (a, b, c)	Need (a, b, c)
P_0	$(7, 5, 3)$	$(0, 1, 0)$	$(7, 4, 3)$
P_1	$(1, 2, 2)$	$(0, 0, 0)$	$(1, 2, 2)$
P_2	$(6, ?, ?)$	-	-
P_3	$(?, ?, ?)$	-	-

P_4	-	-	-

Initial available resources: $(A, B, C) = (3, 3, 2)$.

Calculating Need

- For P_0: $7 - 0 = 7, 5 - 1 = 4, 3 - 0 = 3$ → Need $= (7, 4, 3)$.
- For P_1: $1 - 0 = 1, 2 - 0 = 2$, $2-0 = 2$ → Need $= (1, 2, 2)$.

✅ Safety Check Procedure (Banker's Algorithm)

1. **Find a process whose Need ≤ Available**.
2. **Pretend to execute** it:
   - Add its Allocation back to Available (the process releases all resources after finishing).
3. **Repeat** until all processes can finish or no further process can be satisfied.

Example Walk-through

1. **Start** – Available $= (3, 3, 2)$.
2. **Satisfy** P_1 (needs $(1, 2, 2)$):
   - Allocate 1 A, 2 B, $\mathbb{2}$ C → Remaining Available $= (2, 1, 0)$.
   - After P_1 finishes, resources are released:
     $(2, 1, 0) + (1, 2, 2) = (5, 3, 2)$.
3. **Check remaining processes**:
   - P_2 needs 7 A (cannot).
   - P_3 needs 5 A and 1 B (can).
4. **Satisfy** P_3:
   - Allocate 5 A and 1 B → Remaining Available $= (0, 2, 1)$.
   - After P_3 finishes:
     $(0, 2, 1) + (5, 1, 1) = (7, 4, 3)$.
5. **Satisfy** P_0 (needs $(7, 4, 3)$):
   - Allocate all remaining resources → Available becomes $(0, 0, 0)$.
   - After P_0 finishes, resources return to the original total $= (7, 5, 3)$.
6. **Continue** with P_2, P_4 etc.

The sequence $P_1 \to P_3 \to P_0 \to P_2 \to P_4$ (or variations such as $P_1 \to P_3 \to P_4 \to P_2 \to P_0$) shows that **multiple safe sequences exist**; the system is in a **safe state** because each process can eventually finish.

🔁 Resource Request Algorithm

The request algorithm works together with the safety algorithm to decide whether a new request can be granted.

Conditions to Check

Condition	Requirement
Request ≤ Need	A process may never ask for more than it still requires.
Request ≤ Available	The system must have enough free instances to satisfy the request.

If **both** conditions hold, **temporarily**:

- $\text{Available} \leftarrow \text{Available} - \text{Request}$
- $ \text{Allocation}*{i}* *leftarrow* *text{Allocation}*{i} + \text{Request} $
- $ \text{Need}*{i}* *leftarrow* *text{Need}*{i} - \text{Request} $

Then run the **safety algorithm** on the new state:

- **If the state remains safe** → **grant** the request permanently.
- **Otherwise** → **deny** and block the requesting process.

Example at Time T_1

- **Current state**:
  - Available $= (3, 3, 2)$.
  - P_1 Allocation $= (2, 0, 0)$, Need $= (1, 2, 2)$.
- **Request from** P_1: $(1, 0, 2)$.

1. **Check**
   - Request ≤ Need? $(1, 0, 2) \leq (1, 2, 2)$ → **Yes**.
   - Request ≤ Available? $(1, 0, 2) \leq (3, 3, 2)$ → **Yes**.
2. **Assume request is granted** (temporary state)
   - New Allocation P_1: $(2, 0, 0) + (1, 0, 2) = (3, 0, 2)$.
   - New Need P_1: $(1, 2, 2) - (1, 0, 2) = (0, 2, 0)$.
   - New Available: $(3, 3, 2) - (1, 0, 2) = (2, 3, 0)$.
3. **Run safety algorithm** with Available $= (2, 3, 0)$
   - **Can** P_1 **finish?** Need $(0, 2, 0) \leq$ Available → **Yes**.
     - After P_1 releases its allocation $(3, 0, 2)$, Available becomes $(5, 3, 2)$.
   - **Can** P_3 **finish?** (Need $(0, 1, 1)$) ≤ $(5, 3, 2)$ → **Yes**.
     - After P_3 releases, Available becomes $(7, 4, 3)$.
   - Continue similarly for P_0, P_2, P_4; all can eventually finish.

Since the system stays **safe**, the request $(1, 0, 2)$ from P_1 is **granted**.

📊 Summary Table of Key Variables (After Granting P_1's Request)

Variable	Value
Available	$(2, 3, 0)$
Allocation P_1	$(3, 0, 2)$
Need P_1	$(0, 2, 0)$
Total Resources (sum of all allocations + Available)	$(7, 5, 3)$ – unchanged, confirming conservation of resources.

Core Takeaways

- **Need = Demand – Allocation** (always recompute after any change).
- **Safety algorithm** verifies that a sequence exists for all processes to finish.
- **Resource request algorithm** first validates request limits, then temporarily updates state and invokes the safety check.
- A system can have **multiple safe sequences**; safety does **not** require uniqueness.

🏦 Banker's Algorithm Overview

> **Definition**: The *Banker's algorithm* is a deadlock-avoidance method that evaluates resource-allocation requests and determines whether granting them keeps the system in a *safe state*.

- **Safe state**: A configuration where there exists at least one **safe sequence** of process completions such that all processes can finish without causing deadlock.
- **Need matrix**: $\text{Need} = \text{Maximum} - \text{Allocation}$ (resources each process still requires).

🔢 Example: Safe-State Evaluation (Processes P_4, P_0, P_2)

Process	Allocation	Maximum	Need (= Max – Alloc)
P_4	$(7,4,3)$	$(7,4,5)$	$(0,0,2)$
P_0	$(7,4,5)$	$(7,5,5)$	$(0,1,0)$
P_2	$(6,0,0)$	$(7,0,2)$	$(1,0,2)$

Step-by-step resource accounting

1. After satisfying P_4's need, released resources → new available vector: $(7, 4, 5)$.
2. Grant P_0's request (uses all of its need). Remaining after P_0 finishes: $(7, 5, 5)$.
3. Allocate to P_2 (needs 6 of X). After P_2 finishes, resources return to the original total, confirming a **safe state**.

Result: The system is safe; the safe sequence is **P_4 → P_0 → P_2**. Consequently, the pending request can be granted.

📥 Request Evaluation Using the Resource-Request Algorithm

📄 Request 1

- **P_0** asks for $(0, 0, 2)$
- **P_1** asks for $(2, 0, 0)$

Validity Checks

Check	Condition	Outcome
Request ≤ Need	$(0,0,2) \leq (8,4,2)$ for P_0; $(2,0,0) \leq (3,2,2)$ for P_1	✅
Request ≤ Available	Available = $(3,2,2)$	✅

Tentative Allocation

- New **Available** = $(3,2,2) - (0,0,2) = (3,2,0)$
- Updated **Allocation** for P_0 = $(0,0,1) + (0,0,2) = (0,0,3)$
- Updated **Need** for P_0 = $(8,4,2) - (0,0,2) = (8,4,0)$

Safety Test (after provisional allocation)

1. With Available $(3,2,0)$, P_1's need can be satisfied → after P_1 finishes, Available becomes $(6,2,0)$.
2. No process can now obtain its remaining need because **Z** instances are zero while P_2 still requires Z.

> **Conclusion**: The system would become **unsafe**; therefore **Request 1 is denied**.

📄 Request 2

- **P_1** asks for $(2, 0, 2)$

Validity Checks

Check	Condition	Outcome
Request ≤ Need	$(2,0,2) \leq (3,2,2)$ for P_1	✅
Request ≤ Available	Available = $(3,2,2)$	✅

Tentative Allocation

- New **Available** = $(3,2,2) - (2,0,2) = (1,2,0)$
- Updated **Allocation** for P_1 = previous allocation + request → becomes $(4,0,2)$
- Updated **Need** for P_1 = $(3,2,2) - (2,0,2) = (1,2,0)$

Safety Test (after provisional allocation)

1. With Available $(1, 2, 0)$, P_2 can finish, releasing its resources and raising Available to $(3, 2, 0)$.
2. P_0 can now complete, returning resources and restoring the original total.
3. Finally, P_1 finishes, leaving the system in a safe state.

> **Conclusion**: **Request 2 is granted**; the system remains safe.

🔄 Deadlock Detection & Recovery

> **Definition**: *Deadlock detection* identifies cycles of waiting processes after the system has entered an unsafe state. *Recovery* selects one or more victim processes to break the cycle.

⛓️ Wait-For Graph (WFG) – Single-Instance Resources

1. **Construction**: Remove all resource nodes; keep only process nodes.
2. **Edges**: Draw a directed edge $P_i \rightarrow P_j$ if P_i is waiting for a resource currently held by P_j .
3. **Cycle Detection**:
   - **If a cycle exists**, a deadlock is **present** (cycle is both necessary and sufficient for single-instance resources).
4. **Recovery**: Choose victim processes (e.g., based on priority, cost of rollback) and abort or preempt them to break the cycle.

📊 Multi-Instance Detection – Safety-Algorithm Approach

- Treat the current state as a **resource-allocation snapshot**.
- Apply the **Banker's safety algorithm**: if no ordering of processes can satisfy all remaining needs, the system is **deadlocked**.
- Example from the transcript: with five processes and three resource types, three processes (P_1, P_3, P_4) become blocked, indicating a high likelihood of deadlock.

📚 Summary Table of Key Concepts

Concept	Key Idea	Typical Use
Banker's Algorithm	Checks each request against **Need** and **Available**; simulates allocation and runs safety test.	Deadlock *avoidance* (pre-emptive).
Safe State	Existence of a **safe sequence** where all processes can finish.	Guarantees that granting a request won't lead to deadlock.
Need Matrix	$\text{Need} = \text{Maximum} - \text{Allocation}$	Determines remaining resource demand per process.
Wait-For Graph	Graph of processes only; edges represent waiting for held resources.	Detect deadlock in *single-instance* systems.
Cycle in WFG	Presence of a directed cycle ⇒ deadlock (both necessary & sufficient).	Quick detection for single-instance resources.
Recovery (Victims)	Abort or rollback selected processes to release resources and break cycles.	Resolves detected deadlocks.
Multi-Instance Detection	Apply safety algorithm; if no safe sequence exists, deadlock is present.	Handles systems with multiple copies of each resource type.

🧩 Practical Steps for Handling a New Request

1. **Validate**: Ensure request ≤ Need **and** request ≤ Available.
2. **Pretend to Allocate**: Temporarily update Available, Allocation, and Need.
3. **Run Safety Algorithm**:
   - Find a process whose Need ≤ Available.
   - Assume it finishes, release its Allocation to Available.
   - Repeat until all processes finish or none can proceed.

4. **Decision**:
    - If all can finish → **grant** request.
    - Otherwise → **deny** request (or trigger deadlock detection/recovery).

🚦 Safe State & Safety Algorithm

> **Safe state:** A system state in which there exists *some* sequence of process execution that allows every process to obtain its maximum required resources and complete without causing a deadlock.

- **Available resources** are represented as a vector, e.g., **Available** = $(4, 0, 2)$.
- The **Safety Algorithm** checks whether the current allocation can satisfy the remaining needs of all processes in some order.
- Even if a few processes are currently blocked, the test can still be **negative** (no deadlock) → the system remains in a safe state.

Example Walk-through

1. Initial **Available** = $(1, 1, 3)$.
2. Satisfy P_1's request → allocation updates, **Available** becomes $(4, 0, 2)$.
3. After serving P_1, resources are released and P_4 can be satisfied, etc.
4. All processes can eventually be served → **Safe State**.

🛑 Deadlock Detection & Recovery

> **Deadlock:** A situation where a set of processes are each waiting for resources held by another process in the set, causing none to proceed.

- When a new request (e.g., P_2 asks for $(0, 0, 1)$) cannot be granted and the safety algorithm fails, the system is **deadlocked**.
- All blocked processes are passed to a **Recovery Module** for resolution.

🔧 Recovery Strategies

1. Process-Based Recovery (Abort)

> **Abort:** Terminating one or more processes to break the circular wait.

- **Single-process abort:**
    - Remove a process (e.g., P_3) → its resources are freed to P_2, allowing P_2 to finish, then P_1 can complete.
    - **Cost:** One process is sacrificed.
- **All-process abort:**
    - Terminate every process involved in the deadlock cycle.

2. Resource Preemption

> **Preemption:** Temporarily taking resources from one process and assigning them to another.

- Choose a **strong** (higher-priority) process, e.g., P_2, to snatch resources from a weaker process (P_3).
- P_2 completes, releases resources to P_1, which then finishes and returns resources to P_3.
- **Result:** No process is killed; all eventually complete.

3. Rollback (Iterative Preemption)

> **Rollback:** Repeatedly preempt the most recently allocated resources, adding them to the pool, and re-run the detection algorithm until the deadlock cycle is broken.

- After each preemption, run the detection test; continue until a safe sequence is found.

Selecting a Victim Process

- **Algorithmic rule:** Abort the process that has **just started** (i.e., the newest entrant to the deadlock cycle).

⚖️ Comparison of Approaches

Approach	Attitude	When It Acts	Typical Action
Prevention	Proactive 🚀	Before deadlock can form	Enforce rules (e.g., no circular wait)
Avoidance	Proactive (often considered reactive) 🛡️	At each resource request	Run safety algorithm, deny unsafe requests
Detection & Recovery	Reactive 🔧	After deadlock occurs	Detect cycle → abort or preempt
Ignorance	Inactive ❌	Never	No measures taken; deadlock may persist

---

📊 Example Problems & Solutions

Problem 1 – Minimum Processes for Deadlock (2-copy demand)

- **Given:** Each process needs **2** copies of resource **R**. Total copies available: $1, 2, 3, 4, 5, 6$.
- **Analysis:**
  - With **5** processes, at least one copy remains free → no deadlock.
  - With **6** processes, all copies are allocated, each process waits for another → deadlock.
- **Result:**
  - **Minimum processes to cause deadlock: 6**
  - **Maximum processes for deadlock-free operation: 5**

Problem 2 – Resources for 3 Processes (2-copy demand)

- **Given:** 3 processes, each requires **2** resources.
- **Maximum resources causing deadlock: 3** (allocation pattern where each holds one and waits for another).
- **Minimum resources ensuring freedom: 4** (one extra copy breaks the circular wait).

Problem 3 – Processes with 3-copy demand (6 total copies)

- **Given:** Each process needs **3** copies; total copies = 6 .
- **Result:**
  - **Minimum processes to cause deadlock: 3** (each holds two, all waiting for the third).
  - **Maximum processes for deadlock-free operation: 2** (with two processes the remaining copies can satisfy one).

Problem 4 – Multiple Processes with Different Max Demands

- **Processes:** $P_1, \ldots, P_5$
- **Maximum demand vector:** $(10, 8, 5, 12, 6)$
- **Constructed deadlock allocation:** $(9, 7, 4, 11, 5)$ → total **36** resources.
- **Adding one more resource** (e.g., give an extra copy to any process) allows the waiting chain to break, leading to completion of all.
- **Result:**
  - **Maximum resources that can cause deadlock: 36**
  - **Minimum resources to guarantee deadlock freedom: 37**

📚 Key Takeaways

- **Safe state** ≠ no blocked processes; it means a safe execution order exists.
- **Safety algorithm** is the primary test for safe states.
- **Deadlock recovery** can target processes (abort), resources (preempt), or a combination (rollback).

- **Prevention** and **avoidance** aim to stop deadlocks before they appear, while **detection & recovery** handle them after they arise.
- Numerical reasoning (using resource-allocation vectors) helps determine thresholds for deadlock occurrence.

🧩 Deadlock Fundamentals

> **Deadlock** – a situation where a set of processes are each waiting for resources held by the others, and none can proceed.

✅ True/False Statements on Deadlock

#	Statement	Verdict	Reason
1	*Circular wait* is a necessary condition for deadlock.	**True**	Every deadlock must contain a circular wait among the involved processes.
2	In a system where each resource has more than one instance, a cycle in the **wait-for graph** guarantees deadlock.	**False**	With multi-instance resources, a cycle is only a **necessary** condition; it may not be sufficient.
3	For a **single-instance** resource graph, a cycle is both necessary **and** sufficient for deadlock.	**True**	With one instance per resource, any cycle implies deadlock.
4	If the current allocation leads the system to an **unsafe state**, deadlock will necessarily occur.	**False**	An unsafe state signals a risk of deadlock, but deadlock is not guaranteed.
5	In a resource-allocation graph where **every edge is an assignment edge**, the system is not in deadlock.	**True**	Absence of request edges means no cycles can form, so deadlock cannot exist.

🔄 Resource Allocation Graph (RAG)

> **Assignment edge** – directed from a resource to a process, indicating the resource is held.
> **Request edge** – directed from a process to a resource, indicating the process is waiting for it.

- A **cycle** formed **only by request edges** can indicate a deadlock.
- If **all edges are assignment edges**, no request edges exist → **no cycles** → **no deadlock**.

🧪 Deadlock Detection Using Banker's-style Algorithm (Multi-Instance)

Example Snapshot

Process	Allocation (R1,R2,R3)	Request (R1,R2,R3)
P_0	1 1 0	1 0 0
P_1	1 0 1	0 1 1
P_2	0 1 1	0 0 1
P_3	0 1 0	1 2 0

- **Available** vector = $(0, 0, 1)$ (R3 is the only free instance).

Detection Steps

1. **Find a process whose request ≤ Available.**
   - P_2's request (0,0,1) ≤ (0,0,1) → satisfy P_2.
   - After P_2 finishes, it releases its allocation → **Available** becomes (0,1,1).
2. **Repeat with updated Available.**
   - P_0's request (1,0,0) ≤ (0,1,1)? No.
   - P_1's request (0,1,1) ≤ (0,1,1) → satisfy P_1.
   - Release P_1 → **Available** = (1,1,2).
3. **Continue.**
   - P_0's request (1,0,0) ≤ (1,1,2) → satisfy P_0 → **Available** = (2,2,2).
   - P_3's request (1,2,0) ≤ (2,2,2) → satisfy P_3 → **Available** = (2,3,2).

All processes can finish → **system is safe** → **no deadlock**.

📚 Homework Problem Summaries

1️⃣ Guaranteeing No Approaching Deadlock (X_i / Y_i Scenario)

- **Given:**
  - Each process P_i holds X_i instances of a single resource R.
  - Each P_i requests an additional Y_i instances.
  - Processes **P** and **Q** have $Y_P = Y\_q = 0$ (no further requests).
  - Total available instances = 0 (all are allocated).
- **Necessary condition for safety:**
  - After P and Q finish and release their X_P and X_q resources, the **available** pool becomes $X_P + X\_q$.
  - The system is safe if this pool can satisfy the **minimum** request among the remaining processes:

$$X_p + X_q;\geq;\min_{i\neq p,q} Y_i$$

  - If the condition holds, at least one other process can proceed, releasing more resources, and the system avoids deadlock.
- **Alternative sufficient check:**

$$X_p + X_q;>;\max_{i\neq p,q} Y_i$$

  - Guarantees that even the largest pending request can be satisfied, implying the system is definitely safe.

2️⃣ Detecting Deadlock in a Multi-Instance RAG (Banker's Conversion)

- Convert the graph to **Allocation**, **Request**, and **Available** matrices (as shown in the table above).
- Apply the detection steps to see if all processes can finish.
- If any process remains unsatisfied after no further progress, the system is **in deadlock**.

🧠 Memory Management Overview

> **Memory** – the electronic storage that holds data and instructions the CPU needs quickly.

Memory Hierarchy (Speed vs. Size)

Level	Typical Size	Speed (relative)	Role
Register	Very small	Fastest	Immediate operand access for CPU
Cache	Small (KB–MB)	Faster than RAM	Holds recently used memory lines
Main Memory (RAM)	Moderate (GB)	Slower than cache	Primary workspace for programs
Secondary Storage (Hard Disk / SSD)	Large (TB)	Slowest	Long-term data retention

- **Trend:** Faster memory → smaller capacity.

- **Purpose:** Bridge the speed gap between the ultra-fast CPU and the relatively slow secondary storage.

Primary Memory Details

- **RAM (Random Access Memory):** Read-write, volatile; connected via **address bus**, **data bus**, and **control signals**.
- **Cache:** Organized in lines/blocks; uses **chip-select** signals to enable specific cache portions.
- **Registers:** Directly inside the CPU; accessed without any bus latency.

End of notes.

Memory Organization 🧠

Linear Memory View

> Memory is visualized as a **linear array of words**.

- Each **word** = a collection of bits (commonly 8 bits = 1 byte).
- Memory consists of many **cells**, each holding one word.

Word and Byte Definitions

> **Word** – a fixed-length group of bits stored in a memory cell.
> **Byte** – 8 bits; often used as the basic unit of a word.

Term	Size	Typical representation
Bit	1 bit	binary digit (0 or 1)
Byte	8 bits	8 bits
Word	M bits	M can be 8 bits, 16 bits, etc.

Addressing Basics 📍

Address Size and Number of Words

> An **address** uniquely identifies a memory word.

- If the address bus is n bits, it can generate 2^n distinct addresses.
- Therefore, the **number of addressable words** = 2^n .

Bit-to-Word Examples

Address bits (n)	Distinct addresses	Addressable words
1	$2^1 = 2$	2
2	$2^2 = 4$	4
3	$2^3 = 8$	8
10	$2^{10} = 1{,}024$	1 KB (kilo-words)
20	$2^{20} = 1{,}048{,}576$	1 MB (mega-words)
30	$2^{30} = 1{,}073{,}741{,}824$	1 GB (giga-words)

Memory Capacity Calculation 📏

General Formula

The **memory size** (capacity) is

$$\text{Memory Capacity} = 2^{\text{address bits}} \times M; \text{bits}$$

where

- $2^{\text{address bits}}$ = number of words,
- M = word length in bits.

Example: 13-bit Address, 8-bit Word

- Number of words = $2^{13} = 8{,}192 \rightarrow 8\,\text{K}$ words.
- Capacity in bits = $8{,}192 \times 8 = 65{,}536$ bits = 8 KB.

> **Word-view**: 8 K words.
> **Byte-view**: 8 KB (since 1 word = 1 byte).

Example: 18-bit Address, 8-bit Word

- Words = $2^{18} = 262{,}144 \rightarrow 256\,\text{K}$ words.
- Capacity = 256 K words × 8 bits = 2 M bits = $\mathbb{256}$ KB (byte-view).

Example: 64 KB Memory, 4-byte Word

- Byte-view: 64 KB = 2^{16} bytes ⇒ address length = 16 bits.
- Word-view: word size = 4 bytes → number of words = $\frac{64\text{KB}}{4\text{B}} = 16\text{K}$ words.
- Required address bits for word-view = $\log_2(16\text{K}) = 14$ bits.

Word View vs. Byte View 📦

Address Length Comparison

View	Cell size	Total cells	Address bits needed
Byte view	1 byte	$2^{16} = 64\,\text{KB}$	16 bits
Word view	4 bytes	16 K words	14 bits

Key Takeaways

> - **Address size** is always expressed in bits.
> - To address N words, you need $\lceil \log_2 N \rceil$ address bits.
> - Changing the **word length** changes the **byte-view capacity** but not the number of addressable words.

Practical Calculations

1. **Determine address bits for a given number of words**
   - Example: 500 words → need at least 9 bits because $2^8 = 256 < 500 \leq 2^9 = 512$.
2. **Convert address bits to memory size**
   - 10 bits → $2^{10} = 1{,}024$ words → 1 KB (kilo-words).
3. **Combine address bits and word length**
   - For n address bits and M-bit words, total capacity = $2^n \times M$ bits.

Summary of Core Relationships

Symbol	Meaning
n	Number of address bits
M	Word length (bits)
2^n	Number of addressable words
$n \times M$	Total memory size (bits)
$2^n \times M$	Memory capacity (bits)

These relationships enable calculation of any memory-related parameter from the others.

Memory Addressing Basics 🧠

Byte View vs Word View

> **Byte view**: one memory cell (addressable unit) contains **1 byte**.
> **Word view**: one memory cell contains a **word**, where the word length is given (e.g., 4 bytes, 8 bytes).

- In byte view the address points to individual bytes.
- In word view the address points to whole words; the number of addressable units is reduced accordingly.

Calculating Required Address Bits

Scenario	Memory Size	Word Length	Number of Addressable Units	Bits Required
64 KB, word = 4 bytes	$64\text{KB} = 2^{16}$ bytes	4 bytes = 2^2 bytes	$2^{16}/2^2 = 2^{14}$ words	**14 bits**
64 KB, word = 1 byte (byte view)	2^{16} bytes	1 byte	2^{16} bytes	**16 bits**
64 KB, word = 8 bytes	2^{16} bytes	8 bytes = 2^3 bytes	$2^{16}/2^3 = 2^{13}$ words	**13 bits**
Address width = 23 bits, word = 4 bytes	–	4 bytes	2^{23} words	–
–	Memory size = $2^{23} \times 4$ bytes $= 2^{25}$ bytes $=$ 32 MB	–	–	–
Address width = 18 bits, word = 24 bits (6 bytes)	–	6 bytes	2^{18} words	–
–	Memory capacity = $2^{18} \times 6$ bytes $= 2^{18} \cdot$ 6 B	–	–	–

Key points

- Required address bits = $\log_2$(number of addressable units).
- Changing the word length changes the number of units and thus the bit count.

Memory Organization and Buses 🔌

Definition of Buses

> **Address bus**: carries the memory address from the CPU to memory.
> **Data bus**: transfers the actual data between CPU and memory (read ↔ write).
> **Control bus**: conveys control signals such as **read**, **write**, and **chip-select**.

Control Signals

- **Read**: encoded as 1Z (logic high on the read line).
- **Write**: encoded as 01 (logic high on the write line).
- **Chip select**: 1 → memory chip powered on; 0 → powered off (no operations possible).

Byte-Addressable vs Word-Addressable Memory

- **Byte-addressable**: word length = 1 byte → each address refers to a single byte.
- **Word-addressable**: word length > 1 byte → each address refers to a whole word.

Memory Size Formula

> **Memory size** = (number of words) × (word width).

- Number of words = $2^{\text{address bits}}$.
- Word width is given in bits or bytes (e.g., 8 bits = 1 byte, 32 bits = 4 bytes).

Loading and Linking 📂

Loading

> The **loader** copies an executable program from secondary storage (disk) into main memory so the CPU can execute it.

- **Static loading**: entire program (all code and data) is loaded before execution begins.
- **Dynamic (on-demand) loading**: portions of the program are loaded only when required during execution.

Linking

> **Linking** combines object modules and libraries to create a single executable file.

Advantages & Disadvantages

Method	Advantage	Disadvantage
Static loading	Faster execution (all code already in memory).	May waste memory if some code is never executed (e.g., unused functions).
Dynamic loading	Saves memory; only needed code occupies RAM.	Slower execution due to load time when a new module is required.

Example scenario:
If a conditional statement prevents a function F from being called, static loading still reserves space for F, whereas dynamic loading would avoid that allocation until F is actually needed.

Example Problems 📐

1. **64 KB memory, word = 4 bytes** → **14 bits** (word view) / **16 bits** (byte view).
2. **64 KB memory, word = 8 bytes** → **13 bits** required.
3. **Address width = 23 bits, word = 4 bytes** → total memory = **32 MB**.
4. **Address width = 18 bits, word = 24 bits (6 bytes)** → memory capacity = $2^{18} \times 6\ \text{B}$.

These calculations follow directly from the formulas above:

- Number of addressable units = $2^{\text{address bits}}$.
- Memory size = (units) × (unit size).

📦 Static Loading

> **Definition:** Loading the entire program into memory before execution begins.

- All program modules are present in memory at start → **faster execution**.
- **Space inefficient** because memory is allocated for code that may never be used.

Memory example

Scenario	Memory Required	Waste
Static loading (full program)	10 KB + 5 KB + 15 KB + 15 KB + 30 KB + 20 KB = **95 KB**	40 KB (unused parts)
Dynamic loading (only needed)	10 KB (initial)	None

- **Advantage:** Immediate availability of all code → minimal runtime delay.
- **Drawback:** Large memory footprint; unused modules occupy space.

⚡ Dynamic Loading

> **Definition:** Loading program modules **on demand** during execution.

- Starts with a small memory footprint (e.g., **10 KB**).
- Additional modules are loaded only when required, eliminating wasted memory.
- **Slower execution** because loading incurs runtime overhead.

Key points

- Reduces overall memory usage.
- Increases load time at the moment a module is needed.

🔗 Linking Overview

> **Definition:** The process of **resolving external references** (unresolved addresses) so that a program can execute correctly.

- Occurs after compilation.
- Transforms an object file (.obj) with blanks into an executable (.exe) with concrete addresses.

❓ External References

- Appear when the compiler encounters a function or variable **not yet defined**.
- Compiler inserts a **blank address** (unresolved reference) in the code.

📄 Compilation vs. Execution

- **Compilation** starts from the **first line** of source code, not from main.
- **Execution** begins at main.

🛠️ The Linker

> **Definition:** An OS module that **fills the blanks** left by the compiler, converting unresolved references into concrete addresses.

- Takes the object file (.obj) with blanks.
- Resolves each blank by locating the actual definition (in other object files or libraries).

- Produces the final executable (.exe).

Analogy: The linker is like a tailor that stitches together separate pieces of cloth (code modules) to form a complete garment (program).

📂 Types of Linking

Type	When Performed	Memory Use	Execution Speed	Typical Files
Static Linking	**Before execution** (during build)	Higher (all code loaded)	Faster (no runtime resolution)	.obj → .exe
Dynamic Linking	**At runtime**	Lower (shared libraries loaded as needed)	Slower (runtime resolution)	DLLs / shared objects

🧩 Static Linking

- All required libraries are merged into the executable.
- No further linking needed at run time.

🔄 Dynamic Linking

- Uses **stub code** for each unresolved reference.
- At runtime, the stub triggers the linker to locate the needed function/library.

🪝 Stub Mechanism (Dynamic Linking)

> **Definition:** A small piece of executable code associated with each unresolved reference that **activates the linker at runtime**.

1. Program starts; stub for a missing function is executed.
2. Stub calls the **runtime linker**.
3. Linker searches **memory**; if not found, it scans **compiler libraries on disk**.
4. If found, the **loader** brings the module into memory.
5. Linker returns the function's address to the stub, which patches the original call site.

- **Linker** resolves addresses.
- **Loader** moves modules from disk to memory.

📚 Benefits of Dynamic Linking

- **Space efficiency:** Only one copy of a library (e.g., scanf) resides in memory, shared by all programs.
- **Code reusability:** Multiple programs use the same library without duplicating code.
- **Flexibility:** Library implementations can be updated without recompiling dependent applications; only the address remains constant.

⚠️ Drawbacks of Dynamic Linking

Issue	Reason
Time inefficiency	Runtime address resolution and module loading add overhead.
Security concerns	Loading external code at runtime creates potential vulnerabilities (e.g., missing DLL errors).

- Static linking has **no runtime loading**, thus fewer attack vectors.

🏷️ Address Binding

> **Definition:** The association of program **instructions and data** with specific **memory locations**.

- Binding links an entity (e.g., variable int x) to its **attributes**: name, type, value, **address**, size.
- Occurs during compilation, linking, and loading phases, progressively fixing the final memory address.

📍 Address Binding Overview

> **Address binding** is the association of program instructions and data units to memory locations.

- Example: Variable **X** is bound to address 4000 → **X** ↔ 4000.
- Binding gives each instruction or data item a concrete location in memory.

🧩 Types of Binding

Binding Type	When It Occurs	What Is Bound	Static / Dynamic
Name binding	Compile-time	Identifier ↔ entity (e.g., int X)	**Static**
Type binding	Compile-time	Identifier ↔ data type	**Static**
Size binding	Compile-time	Identifier ↔ size of storage	**Static**
Address binding	Load-time (or runtime for dynamic)	Instruction / data unit ↔ memory address	Usually **Static**, can be **Dynamic** at runtime
Value binding	Runtime	Identifier ↔ actual value (e.g., X = 1)	**Dynamic**

🔄 Static vs. Dynamic Binding

> **Static binding** – the association cannot change after it is established.
> **Dynamic binding** – the association may change during program execution.

- **Static examples**:
  - Name of a person (Rahul) – unchanged for the lifetime.
  - Fingerprint – unique and immutable.
  - Variable **type** (int) and **size** – fixed at compile time.
- **Dynamic examples**:
  - Variable **value** (X = 1 → later X = 2).
  - **Age** of a person – changes over time.
  - Addresses in **runtime address binding** – may differ each load.

🛠️ Example: Translating a Simple Program

High-level code

```
int a = 1, b = 2, c;
c = a + b;
```

Low-level sequence (illustrative):

1. **Store constant** 1 → a
2. **Store constant** 2 → b
3. **Load** a → R1
4. **Load** b → R2
5. **Add** R1 + R2 → R1
6. **Store** R1 → c

Assumed memory layout for data units:

Variable	Memory address
a	0
b	2
c	4

Program instructions (i1, i2, i3, ...) are also placed in memory and receive addresses via binding.

⏱️ Binding Times

⏳ Compile-time Binding

> Addresses of data units and instructions are fixed by the compiler → **static**.

- Compiler decides exact addresses; no further relocation needed.

📦 Load-time Binding

> Compiler emits **offsets**; the loader converts them to real addresses using a **base register**.

- Offsets example:
  - i1 offset = 0
  - i2 offset = size(i1)
  - i3 offset = size(i1) + size(i2)
- If the base register holds 6000, real addresses become:
  - i1 → 6000
  - i2 → $6000 + \text{size}(i1)$ (e.g., 6004)
  - i3 → $6000 + \text{size}(i1) + \text{size}(i2)$ (e.g., 6008)

🚀 Runtime (Dynamic) Address Binding

> Performed by the loader each time a process is brought into memory; addresses may differ on each load, enabling **relocation flexibility**.

- When a process is swapped out, its instructions become free slots.
- On swap-in, the loader can place them at any available region, updating the bindings accordingly.

📚 Summary Table of Binding Attributes

Attribute	Binding Type	Binding Time	Static / Dynamic
Name	Name binding	Compile-time	Static
Type	Type binding	Compile-time	Static
Size	Size binding	Compile-time	Static
Address	Address binding	Load-time (or Runtime)	Typically Static; can be Dynamic at Runtime
Value	Value binding	Runtime	Dynamic

These notes capture the core concepts of address binding, the various binding types, their timing, and the distinction between static and dynamic associations.

Address Binding Types 📍

> **Address binding** is the process of assigning a program's logical addresses to physical memory locations.

Binding Time	When It Occurs	How Addresses Are Determined
Compile-time	During compilation	Compiler generates absolute addresses; no relocation needed at load time.
Load-time	When the loader places the program in memory	Loader receives a **base register** from the memory manager and adds it to the offsets produced by the compiler.
Run-time	While the program is executing	The CPU/OS translates logical addresses to physical ones dynamically (e.g., via paging).

Program Execution Process 🚀

1. **Source program** (e.g., program.c)
2. **Compilation** – compiler → **object module** (.obj) – contains unresolved references.
3. **Linkage editing** – linker resolves references, produces **load module** (.exe).
4. **Loading** – loader places the load module into main memory, creating a **process**.
5. **Execution** – CPU runs the process.

> **Library function**: If an address isn't found in memory, the loader searches the disk for required library modules.

Memory Management Overview 🧠

Functions of the Memory Manager

- **Allocation** – reserve memory space before a process is loaded.
- **Protection** – keep each process confined to its own **address space**; prevent trespassing.
- **Free-space management** – track and reuse unallocated memory blocks.
- **Address translation** – map logical addresses to physical addresses.
- **Deallocation** – reclaim memory when a process terminates.

Goals of the Memory Manager

- **Minimize fragmentation** (internal & external).
- **Utilize memory efficiently** – fit large programs into limited physical memory.
- **Increase multiprogramming degree** – more processes resident ⇒ higher throughput and efficiency.

Fragmentation 🪟

> **Fragmentation** is wasted memory caused by imperfect allocation of space.

Type	Description
Internal	Unused space *inside* an allocated block (e.g., a block larger than the request).
External	Unused gaps *between* allocated blocks that are too small to satisfy a request.

Managing Large Programs in Small Memory 📦

- **Virtual memory** and **overlays** allow execution of programs larger than available RAM.
- The OS loads only the necessary portions of a program, swapping others in as needed.

Overlays Concept 🎭

> **Overlay** – a technique where independent program modules are loaded into the same memory region at different times, replacing one another as execution proceeds.

Two-Pass Assembler Example

Component	Size (KB)
Pass 1	70
Pass 2	80
Symbol table	30
Common routines	20
Overlay loader	10
Total	**210**

- Available memory: **150 KB**.
- Execution strategy:
  1. Load **Pass 1** (70 KB) + required overlay support (30 KB) = 100 KB → fits.
  2. After Pass 1 finishes, **replace** it with **Pass 2** (80 KB) + overlay support (30 KB) = 110 KB → fits.

Thus, the program runs despite memory being less than the full size.

Overlay Tree Example & Memory Requirement

- Program size: **92 KB**.
- Overlay tree creates several **passes** (independent execution paths).
- Memory needed for each pass (sum of modules on that path):

Pass	Modules (KB)	Total Required (KB)
1	10 + 8 + 5	23
2	10 + 6 + 10	26
3	10 + 7 + 12	29
4	10 + 7 + 17	34
5	10 + 7 + 9 + ?	26 (approx.)
...	...	...

- **Maximum** memory among all passes = **29 KB**.
- **Minimum memory** required to run the whole program = **29 KB**, far less than the original 92 KB.

Partitioning Strategies 📂

Strategy	Description	Example Allocation
Contiguous (centralized)	Entire program stored in one continuous block.	100 KB program → one 100 KB hole.
Non-contiguous (distributed)	Program split into several independent pieces placed in separate holes.	100 KB program → 40 KB + 40 KB + 20 KB blocks scattered.

- **Non-contiguous** enables better utilization of fragmented free space.

Historical vs. Modern Memory Management Techniques 🕰️

Category	Techniques

Old (foundational)	Overlays, Fixed-size partitions, Variable-size partitions, Buddy system
New (current)	Paging, Segmentation, Segmented paging, Demand paging

- Studying older methods is essential because modern techniques evolved from them.

Fixed Partitioning 📦

Concept

> **Fixed partitioning** divides the user area of memory into a predetermined number of partitions.

- Each partition can hold **only one program** (one-to-one relationship).
- Partitions may have **different sizes** to accommodate programs of varying lengths.

Limit and Base Registers 🛡️

> **Base register** – stores the starting physical address of a process's address space.
> **Limit register** – stores the size (range) of the process's address space.

- When the CPU generates a logical address, the memory manager checks:
  1. **Base check:** Is the address ≥ *Base*?
  2. **Limit check:** Is the address < *Base* + *Limit*?

If either check fails, an **addressing error trap** is raised.

Example:

- Process P_i has *Base* = 2000 and *Limit* = 500 (so valid range 2000–2500).
- Generated address 2800 → fails limit check → trap.
- Generated address 2100 → passes both checks → access allowed.

Partition Allocation Policies 📊

Policy	Search Strategy	Example Placement (Program 80KB)	Expected Internal Fragmentation
First Fit	Scan partitions from the **beginning**; pick the first that fits.	Fits into 200KB partition → 120KB wasted.	High (waste of 120KB).
Best Fit	Scan **all** partitions; choose the smallest that fits.	Fits into 81KB partition → 1KB wasted.	Minimal (waste of 1KB).
Next Fit	Continue scan from the **last allocated** partition; wrap around if needed.	May place in 200KB partition (same as First Fit) but search starts later, potentially saving time.	Depends on last position; similar fragmentation to First Fit.
Worst Fit	Choose the **largest** available partition.	Places in 200KB partition (largest) → 120KB wasted.	Highest fragmentation; useful when trying to keep large free blocks.

Fragmentation in Fixed Partitioning 🔧

- **Internal fragmentation:** Unused space *inside* a partition because the program is smaller than the partition.
  - Example: 200KB partition holding an 80KB program → 120KB internal waste.
- **External fragmentation:** Does **not occur** in fixed partitioning because memory is entirely partitioned; there is no free space **outside** partitions.

Degree of multiprogramming

- Limited to the **number of partitions**; cannot exceed this count.

Maximum process size

- Bounded by the **largest partition size** in the system.

Memory Management Techniques Overview 🧠

- **Contiguous allocation:** Entire program resides in one continuous block of memory.
- **Non-contiguous allocation:** Program is broken into pieces placed in separate locations.

Older techniques (historical):

Technique	Description
Overlays	Load only required program sections; replace them with other sections as needed.
Partitions	Fixed-size divisions of memory (as described above).
Buddy system	Memory split recursively into halves to satisfy allocation requests.

Primary goals of memory management

- **Allocation** – assign memory to processes.
- **Protection** – prevent illegal access (e.g., via base/limit checks).
- **Free-space management** – keep track of available memory.
- **Address translation** – map logical to physical addresses.
- **Deallocation** – reclaim memory when a process terminates.
- **Minimize fragmentation** – reduce both internal and external waste.

Variable Partitioning 🔀

Concept

> **Variable (dynamic) partitioning** creates partitions **on-the-fly** based on the exact memory requirements of arriving processes.

- Initially, memory is a single large **hole** with no partitions.
- When a process arrives, a partition equal to its size is carved out.
- Example sequence:
  1. Process P_1 requests 35KB → a 35KB partition is created.
  2. Process P_2 requests 115KB → a 115KB partition is added, etc.

Multiprogramming with Variable Tasks

- Unlike **fixed-task** multiprogramming (limited by a static number of partitions), **variable-task** multiprogramming adapts the number and size of partitions to the current set of processes, allowing a more flexible degree of multiprogramming.

📂 Dynamic Partitioning & Free Holes

> **Dynamic partitioning** – Memory is divided into partitions of varying sizes at run-time based on process requests. When a process finishes, its partition becomes a *free hole*.

- Example free holes after processes P1, P3, P5 exit:
  - Hole 1: **115 KB**
  - Hole 2: **15 KB**
  - Hole 3: **250 KB**

When process **P6 (13 KB)** arrives:

1. Choose a suitable free hole.

2. Allocate a **13 KB** partition inside that hole.
3. The remainder of the original hole becomes a new free hole (e.g., Hole 1 → 13 KB used, **102 KB** left).

🎯 Allocation Policies

Policy	Search Strategy	Example for P6 (13 KB)	Resulting Hole (Remaining)
First Fit	Scan from the beginning; pick the first hole large enough.	Hole 1 (115 KB)	102 KB
Next Fit	Continue scanning from the last allocated hole.	Hole 3 (250 KB)	237 KB
Best Fit	Examine all holes; choose the one leaving the **least** leftover space.	Hole 2 (15 KB)	2 KB
Worst Fit	Choose the largest hole to leave the biggest remainder.	Hole 3 (250 KB)	237 KB

🗃️ External Fragmentation

> **External fragmentation** – Free memory that is split into several small holes, making it impossible to satisfy a larger request even though the total free space is sufficient.

- In the **Best Fit** example, a **2 KB** remainder is left; such tiny holes are rarely useful.
- Example: free holes of **115 KB**, **15 KB**, **250 KB** give a total of **380 KB**, yet a **280 KB** request may be denied if no single hole is ≥ 280 KB.

🔧 Internal Fragmentation

> **Internal fragmentation** – Wasted space **inside** an allocated partition because the partition is larger than the process it holds.

- Dynamic partitioning **avoids** internal fragmentation by sizing partitions exactly to the process size.
- In fixed-size partitioning, a process may occupy only part of its partition, leaving the rest unused.

🧩 Solving External Fragmentation

1. Compaction

> **Compaction** – Moving allocated partitions to one end of memory, merging adjacent free holes into a single large hole.

- Example: free holes **50 KB**, **80 KB**, **20 KB** → after shifting processes P1 and P2, a single **150 KB** hole appears.
- Drawback: time-consuming and requires **runtime address binding** because process addresses change.

2. Non-Contiguous Allocation

> **Non-contiguous allocation** – A process is split into several pieces and stored in separate free holes.

- If a process agrees to be stored non-continuously, the combined size of multiple holes can satisfy the request (e.g., 250 KB + 30 KB).
- This method eliminates external fragmentation without moving other processes.

🗂️ Variable Partitioning (Example)

Memory layout (sizes in MB):

Partition	Size
1	2
2	6
3	3
4	4
5	6

- Processes occupy partitions; when a process finishes, its partition becomes a free hole.
- **P6 (4 MB)** placed into a 4 MB hole → leaves **0 MB** free.
- **P7 (3 MB)** placed into a 3 MB hole → leaves **0 MB** free.
- **P8 (3 MB)** cannot be loaded because no single **3 MB** contiguous hole exists, even though total free memory = 3 MB.

📊 Allocation Example Using Best Fit

Given partitions: **200 KB**, **300 KB**, **400 KB**, **500 KB**, **600 KB**, **600 KB**

Process requests (KB): **357**, **210**, **468**, **49**

Process	Chosen Partition (Best Fit)	Unallocated Partitions
357	400 KB (leaves 43 KB)	–
210	250 KB (leaves 40 KB) – *note: 250 KB derived from 300 KB after allocation*	
468	500 KB (leaves 32 KB)	–
49	200 KB (leaves 151 KB)	**600 KB**, **300 KB** (remain unused)

📋 Memory-Map Allocation (First Fit)

Free/used blocks (sizes in KB):

1. **300 KB** request → first hole ≥ 300 KB is used.
2. **25 KB** request → next suitable hole after the previous allocation is selected.
3. **125 KB** request → continues scanning from the last allocated position.
4. **50 KB** request → follows the same first-fit rule.

(Exact hole sizes were not listed in the transcript; the procedure follows the First Fit algorithm.)

Memory Allocation Strategies 🚀

> **First Fit** – Allocate the first partition that is large enough for the request.
> **Best Fit** – Allocate the smallest partition that can satisfy the request, aiming to leave the least leftover space.

- **Example sequence** (process sizes in KB): 125 → 50 → 300 → 25 → 125 → 50.
- Allocation using **First Fit**:
  1. 125 KB placed in the first suitable hole.
  2. 50 KB placed in the next suitable hole.
  3. 300 KB placed in a hole of at least 300 KB.
  4. 25 KB placed in a 25 KB hole.
  5. 125 KB placed in the remaining 125 KB hole.
  6. The final 50 KB request cannot be satisfied because no hole of ≥ 50 KB remains.
- The analysis concluded that **First Fit** yielded a feasible allocation while **Best Fit** would have chosen a different hole for the 50 KB request, leading to failure. Hence, *First Fit* was the better choice in this scenario.

Variable-Partition Allocation without Compaction 📂

Scenario: Total memory = $1000\,\text{KB}$, two occupied partitions of $200\,\text{KB}$ and $260\,\text{KB}$.

- Occupied size = $200 + 260 = 460\,\text{KB}$ → **Free hole** = $1000 - 460 = 540\,\text{KB}$.

Smallest request always denied

- Any request larger than the total free space must be denied.
- Therefore, the smallest request that **will always be denied** is

$$540\,\text{KB} + 1\,\text{KB} = 541\,\text{KB}.$$

Creating holes to lower the guaranteed-denial size

By shifting existing partitions, the single $540\,\text{KB}$ hole can be split into three holes that still sum to 540KB (e.g., $180\,\text{KB}$, 180KB, 180KB).

- Requests of $131\,\text{KB}$, $151\,\text{KB}$ and $181\,\text{KB}$ can be placed in two of the $180\,\text{KB}$ holes, leaving the 181KB request **denied**.

Thus, by arranging holes appropriately, the **smallest request that may be denied** can be reduced from $541\,\text{KB}$ to $181\,\text{KB}$.

Fixed Partitioning and Process Table 📊

Parameter	Value
Memory size	$2^{46}\,\text{bytes}$
Partition size	$2^{24}\,\text{bytes}$
Number of partitions	$\frac{2^{46}}{2^{24}} = 2^{22}$
Bits needed for a partition address	$22\,\text{bits}$
Pointer size (nearest byte)	$\lceil 22/8 \rceil = 3\,\text{bytes}$
Process-ID field	$4\,\text{bytes}$
Entry size (pointer + PID)	$3 + 4 = 7\,\text{bytes}$
Processes in system	500
Process-table size	$500 \times 7 = 3500\,\text{bytes}$

> **Pointer** – Stores the address of the partition in which the process resides.
> **Process-ID** – Unique identifier for the process.

Best-Fit Allocation with FCFS Scheduling ⏱️

Initial free holes

Hole ID	Size
H_1	$4\,\text{KB}$
H_2	$8\,\text{KB}$
H_3	$20\,\text{KB}$
H_4	$2\,\text{KB}$

Program list (size / CPU-time)

Process	Size	CPU-time (units)
P_1	$2\,KB$	4
P_2	$14\,KB$	10
P_3	$3\,KB$	–
P_4	$20\,KB$	–
P_5	$6\,KB$	–
P_6	$10\,KB$	–

Allocation using Best Fit

1. **P_1 (2 KB)** → placed in the smallest hole that fits → $2\,KB$ of H_4 (leaving $0\,KB$).
2. **P_2 (14 KB)** → best fit is the $20\,KB$ hole (H_3); after allocation, a $6\,KB$ hole remains.
3. **P_3 (3 KB)** → best fit is the $4\,KB$ hole (H_1); leaves a $1\,KB$ hole.
4. **P_4 (20 KB)** → no hole ≥ 20 KB; remains on disk.
5. **P_5 (6 KB)** → fits exactly into the $6\,KB$ hole created after P_2.
6. **P_6 (10 KB)** → best fit is the $8\,KB$ hole (H_2) **does not fit**, so it stays on disk.

Resulting free holes after allocation:

Hole	Size
H_1 (remaining)	$1\,KB$
H_2	$8\,KB$
H_3 (remaining)	$0\,KB$
New hole from P_2	$6\,KB$ (now used by P_5)

FCFS CPU scheduling

Processes are scheduled in arrival order (P_1 → P_2 → …).

Time interval	Running process	Action
$0 - 4$	P_1 (2 KB)	Executes for 4 units; finishes, freeing its $2\,KB$ space.
$4 - 14$	P_2 (14 KB)	Executes for 10 units; finishes, creating a $14\,KB$ hole.
$14 - \ldots$	P_6 (10 KB)	The $14\,KB$ hole now accommodates P_6 (10 KB).

When P_2 completes, the newly freed $14\,KB$ hole allows the previously pending $10\,KB$ process (P_6) to be loaded and scheduled next.

> **Note** – Gaps are deliberately left between allocated blocks (visualized as “purple shade” in the original material) to prevent automatic merging of adjacent holes when compaction is disallowed.

Memory Allocation Policies 🗃️

> **Best Fit** – Allocates a process to the smallest free hole that is large enough, minimizing leftover space.

> **First Fit** – Scans memory from the beginning and places the process in the first hole that is big enough.

> **Next Fit** – Similar to First Fit but continues the search from the location of the last allocation.

> **Worst Fit** – Places the process in the largest available hole, hoping to leave sizeable fragments.

True/False Statements

Statement	Verdict	Reason
"The hole created by **Best Fit** is never larger than the hole created by **First Fit**."	**True**	Best Fit always picks the smallest suitable hole, so its leftover fragment cannot exceed that of First Fit.
"The hole created by **Worst Fit** is always larger than the hole created by **First Fit**."	**False**	Worst Fit and First Fit can select the same hole; they are not guaranteed to differ.
"The hole created by **First Fit** is always larger than the hole created by **Next Fit**."	**False**	Both may pick identical holes; "always larger" is not guaranteed.
"The hole created by **Best Fit** is never larger than the hole created by **First Fit**."	**True**	Re-states the first true statement.

Partition Allocation Example 📊

Process Sizes & Arrival

- 2 K, 3 K, 6 K, 10 K, 14 K, 20 K (holes and processes mentioned).

Scheduling Timeline (time units)

Process	Size (K)	Start Time	Duration (units)	End Time
P1	2	0	–	–
P2	3	14	2	16
P3	6	16	1	17
P4	2	17	8	25
P5	10	25	4	29
P6	20	29	1	30
P7	–	–	–	–

Notes

- After loading **P1** at time 0, a free hole of 10 K remains.
- **P2** (3 K) uses the first 2-unit slot (14-16).
- **P3** (6 K) occupies the next 1-unit slot (16-17).
- **P4** (2 K) is allocated for 8 units (17-25).
- **P5** (10 K) runs from 25-29 (4 units).
- **P6** (20 K) fits into a 1-unit hole (29-30).

Load Times

- P1-P5 loaded at time 0.
- P6 loaded at time 14 (the 10 K hole).
- P7 loaded at time 29 (the 20 K hole).

Fragmentation 📉

> **Internal Fragmentation** – Unused memory inside an allocated partition because the partition is larger than the process. Occurs in fixed-partition systems.

> **External Fragmentation** – Free memory scattered in small holes that are insufficient individually for waiting processes.

How Variable Partitioning Helps

- Partitions are created **exactly** to match process sizes, eliminating the fixed-size gaps that cause internal fragmentation.
- When a process terminates, its partition becomes a free hole that can be reused, reducing external fragmentation.

Non-Contiguous Allocation (Paging) 🧩

Scenario

- Process size: 100 KB
- Available free blocks: 40 KB, 40 KB, 20 KB (non-adjacent)

Allocation Steps

1. Split the process into three pieces: 40 KB, 40 KB, 20 KB.
2. Store each piece in a separate free block.

> This method avoids the need for a single contiguous 100 KB hole, thereby **preventing external fragmentation**.

Compaction 🔧

> **Compaction** – The runtime operation of moving allocated processes in memory to coalesce separate free holes into one larger contiguous hole.

- Example: two free holes of 20 KB and 40 KB become adjacent after moving a process, forming a single 60 KB hole.
- **Drawbacks**: time-consuming and requires programs to support runtime address binding.

Address Space 🗺️

> **Address Space** – The complete set of addressable memory locations (words) that a system can reference.

- **Logical (Virtual) Address Space** – The addresses used by a program; abstracted from physical locations.
- **Physical Address Space** – The actual hardware memory addresses.

Why Separate Logical from Physical?

- **Security**: Processes cannot directly generate physical addresses of other processes, preventing unauthorized memory access.
- **Flexibility**: The operating system can map logical addresses to any available physical location, enabling techniques like paging and relocation.

🔐 Unbiased Copy Checking (Dummy Role Numbers)

> **Definition:** A *dummy (logical) role number* replaces the actual roll number on answer sheets to prevent examiners from recognizing students and giving biased marks.

- The actual roll number (e.g., **IITB 404**) is swapped with an arbitrary logical number.
- Purpose: eliminate any conscious or unconscious preference toward known students.

🧠 Logical vs. Physical Addresses

> **Logical Address:** The address generated by a process; it exists only in the process's *logical address space*.

> **Physical Address:** The real location in main memory where data/instructions reside; accessed only by the operating system (OS) via the MMU.

Address Translation Process

1. **Process** creates an *n-bit logical address*.
2. The **Memory Management Unit (MMU)** translates it to a *physical address* (may have a different bit width).
3. The OS controls the mapping; the process never sees the physical address.

📊 Physical Address Space Capacity

> **Physical address space** = $2^{(\text{number of physical address bits})}$ words.

- If the physical address width = **4 bits**, then capacity = $2^4 = 16$ words.
- For a **byte-addressable** memory, 16 bytes = 16 B.

Physical Address Width	Capacity (words)	Capacity (bytes)
4 bits	$2^4 = 16$	16 B
8 bits	$2^8 = 256$	256 B
...	...	...

🔒 Why Processes Use Logical Addresses (Security)

- Direct physical access would let a process read/write **any** memory, including other processes (P_2) or the OS itself.
- The OS isolates processes by exposing only *logical addresses* and handling the mapping internally.
- This prevents unauthorized access to:
  - Other user processes
  - Kernel/OS memory regions

📋 Mapping Table (Managed by OS)

- The OS maintains a **page table** (or similar mapping structure) that links each logical address (or page) to a physical frame.
- The MMU consults this table during translation.

> **Definition:** A *mapping table* is a data structure kept by the OS that stores the correspondence **Logical Page → Physical Frame**.

📚 Paging Fundamentals

Logical Address Space Division

- The logical (virtual) address space is split into **equal-sized units called pages**.
- Example: 8 KB logical space, page size = 1 KB → **8 pages** (P0 … P7).

Page Number Bits

- Number of pages = $\frac{\text{Logical space size}}{\text{Page size}}$.
- Bits needed for the **page number** = $\log_2(\text{number of pages})$.
  - For 8 pages: $\log_2 8 = 3$ bits.

Page Offset Bits

- Each page contains multiple words/bytes.
- **Page offset** identifies the specific word/byte within a page.
- Bits needed = $\log_2(\text{page size in words/bytes})$.

- 1 KB page = 2^{10} bytes → **10-bit offset**.

Example Breakdown (8 KB logical, 1 KB pages)

Component	Bits	Explanation
Page Number	3	$\log_2(8 \text{ pages})$
Page Offset	10	$\log_2(1\text{KB})$ (byte-address)
Total Logical Address	**13**	3 bits + 10 bits = 13 bits (matches given)

Analogy

- **Bench = Page number** (identifies which bench in a classroom).
- **Student position on bench = Page offset** (which student on that bench).

The process thinks it's accessing a single, continuous logical memory; the OS & MMU handle the actual fragmented physical placement.

🧩 Summary of Address Components

- **Logical address** = *Page number* + *Page offset* (generated by the process).
- **Physical address** = result of OS-controlled translation (used by hardware to fetch/store data).
- **Security**: Logical addresses hide the real layout, preventing cross-process/OS interference.
- **Paging** provides a simple, uniform method to map logical pages to physical frames, enabling efficient memory management.

📄 Logical Address Structure

> **Logical address** – an address generated by the CPU; it is divided into two fields:
>
> - **Page number** – identifies which page contains the desired instruction.
> - **Page offset** – identifies the word (or byte) within that page.

🔢 Determining Bits for Each Field

Quantity	Formula	Result
Total logical address bits	given (e.g., 25 bits)	–
Page size	2^{12} bytes (4 KB)	–
Number of pages	$\frac{2^{25}}{2^{12}} = 2^{13}$	13 bits for **page number**
Offset bits	$25 - 13 = 12$	12 bits for **page offset**

Example – Logical address space of 32 MB:

- $32\,\text{MB} = 2^{25}$ bytes → need 25 bits total.
- With $4\,\text{KB} = 2^{12}$ bytes pages → 2^{13} pages → **13 bits** for page number, **12 bits** for offset.

📚 Sample Question Walk-through

> *Question*: Logical address space = 32 bits, total pages = 2^{11} (2 K). Find the page size.

- Logical space: 2^{32} bytes.
- Pages: 2^{11} → Page size = $\frac{2^{32}}{2^{11}} = 2^{21}$ bytes = **2 MB**.

🖥️ Physical Address Structure

> **Physical address** – an address used by the memory hardware; it consists of:
>
> - **Frame number** – tells which frame holds the required page.
> - **Frame offset** – same as page offset; tells the exact word/byte inside the frame.

- **Frame size = Page size** (each frame holds exactly one page).
- Number of frames = $\frac{\text{Physical address space}}{\text{Frame size}}$.

🔄 Mapping Logical to Physical

1. Extract **page number** from logical address → locate the corresponding **frame** via the page table.
2. Combine **frame number** with the unchanged **offset** to form the physical address.

📊 Bit Allocation Example

Attribute	Value	Bits Required
Logical address size	31 bits	–
Page size	$4\,\text{KB} = 2^{12}$ bytes	–
Number of pages	$2^{31-12} = 2^{19}$	**19 bits** for page number
Offset (within page)	12 bits	–
Physical address size	20 bits	–
Number of frames	$2^{20-12} = 2^{8}$	**8 bits** for frame number

- **Offset bits** are identical for both logical and physical addresses (here, 12 bits).

📐 General Formulas

- **Page bits** = $\log_2(\text{Number of pages})$
- **Frame bits** = $\log_2(\text{Number of frames})$
- **Offset bits** = $\log_2(\text{Page size in bytes})$

> *If you have n bits, you can generate 2^n unique numbers, which can be used to label 2^{n} pages or frames.*

🧩 Key Concepts Recap (Blockquotes)

> **Page** – a fixed-size block of logical memory.
> **Frame** – a fixed-size block of physical memory; each frame stores exactly one page.
> **Page number** – $\log_2(\text{total pages})$ bits; selects the page.
> **Frame number** – $\log_2(\text{total frames})$ bits; selects the frame.
> **Offset** – $\log_2(\text{page size})$ bits; selects the word/byte within the selected page/frame.

📏 Additional Example

Given: 4 K pages, 1 K frames, logical address = 32 bits.

- Pages = 2^{12} → Page size = 2^{20} bytes = **1 MB** (since logical space $2^{32} \div 2^{12} = 2^{20}$).
- Frames = 2^{10}, each frame = page size = 2^{20} bytes.
- Physical memory size = $2^{10} \times 2^{20} = 2^{30}$ bytes = **1 GB**.

📐 Physical Address Space Calculation

> **Physical address** – the actual location in main memory where a byte resides.

- **Offset bits**

  $\text{offset bits} = \log_2(\text{page size}) = \log_2(\text{frame size})$

  (page size = frame size)

- **Frame-number bits**

  $\text{frame bits} = \log_2(\text{number of frames})$

- **Total physical address bits**

  $\text{physical address bits} = \text{frame bits} + \text{offset bits}$

Example

Parameter	Value	Computation
Page/Frame size	2^{20} bytes	$\log_2(2^{20}) = 20$ bits (offset)
Number of frames	2^{10}	$\log_2(2^{10}) = 10$ bits (frame number)
Physical address bits	30 bits	$10 + 20 = 30$ bits
Physical address space	2^{30} bytes	$= 1\,\text{GB}$

🧩 Logical vs. Physical Address

> **Logical address** – the address generated by a process (also called a virtual address).

Component	Logical address	Physical address
High-order part	**Page number** ($\log_2 n$ bits)	**Frame number** ($\log_2 m$ bits)
Low-order part	**Offset** ($\log_2$ page size)	**Offset** (same as logical)

- n = number of pages in the logical address space.
- m = number of frames in the physical address space.

Formulas

- Page-number bits: $\log_2 n$
- Frame-number bits: $\log_2 m$
- Offset bits: $\log_2$(page size) = $\log_2$(frame size)

📊 Summary of Key Formulas

Quantity	Formula
Offset bits	$\log_2(\text{page size})$
Page-number bits	$\log_2(\text{number of pages})$
Frame-number bits	$\log_2(\text{number of frames})$
Number of frames	$\dfrac{\text{physical address space}}{\text{frame size}}$
Page-table entry size	e bytes
Page-table total size	$(\text{number of pages}) \times e$

🧠 Memory Management Unit (MMU) Overview

> **MMU** – hardware that translates logical addresses to physical addresses using a page table.

- Each **process** has its **own page table**.
- Page tables are **stored in main memory**.
- A page table consists of **page-table entries (PTEs)**.

📁 Page Table Structure

> **Page-table entry (PTE)** – stores the frame number where a particular page resides.

- **Number of entries** = number of pages in the logical address space.
- **Entry content**: frame number (often denoted as F_i).
- **Entry size** = e bytes.

Illustrative Table

Page (index)	Frame stored in
P_0	F_7
P_1	F_2
P_2	F_3
...	...
P_{22}	F_{17}

The index automatically identifies the page; the entry gives its frame.

Total page-table size

$$\text{size} = (\text{number of pages}) \times e$$

🔄 Address Translation Process

1. **Generate logical address** → split into **page number** and **offset**.
2. **Lookup** page number in the **page table** to obtain the **frame number**.
3. **Form physical address**: concatenate **frame number** with the **same offset** (page size = frame size).

Step-by-step example

- Logical address space: 8 KB → 2^{13} bytes → 13 bits.
- Physical address space: 4 KB → 2^{12} bytes → 12 bits.
- Page size: 1 KB → 2^{10} bytes → offset = 10 bits.

Logical address breakdown	Value
Page-number bits	$13 - 10 = 3$ bits → 8 pages (P_0–P_7)
Offset bits	10 bits

- Physical memory has 4 frames (F_0–F_3).
- If the process requests page P_7, the MMU consults the page table, finds the corresponding frame (e.g., F_0), and appends the 10-bit offset → **12-bit physical address** ready for memory access.

📚 Key Takeaways (Blockquote)

> - **Offset** is derived from the page/frame size: $\log_2(\text{size})$.

> - **Physical address bits** = **frame bits** + **offset bits**.
> - **Logical address bits** = **page bits** + **offset bits**.
> - The **MMU** uses a per-process **page table** to map pages to frames, preserving the offset unchanged during translation.

📄 Paging Overview

Logical and Physical Addresses

> **Logical address**: Consists of a **page number (P)** and an **offset (D)** within that page.

> **Physical address**: Formed by concatenating the **frame number** retrieved from the page table with the original **offset (D)**.

Page Table Mechanics

> **Page-table base register (PTBR)**: Holds the memory address where the page table for the current process starts.

- The CPU extracts the page number **P** from the logical address.
- **P** indexes the page table (via PTBR) to obtain the corresponding **frame number**.
- The frame number is combined with the offset **D** to produce the physical address that accesses main memory.

📊 Key Formulas

Quantity	Formula	Description
Number of pages	$\text{Pages} = \dfrac{\text{Logical address space}}{\text{Page size}}$	Total pages required to cover the logical space
Bits for page number	$\text{bits}_P = \log_2(\text{Number of pages})$	Width of the page-number field
Bits for offset	$\text{bits}_D = \log_2(\text{Page size})$	Width of the offset field
Number of frames	$\text{Frames} = \dfrac{\text{Physical address space}}{\text{Page size}}$	Total frames in main memory
Bits for frame number	$\text{bits}_F = \log_2(\text{Number of frames})$	Width of the frame-number field
Page-table size	$\text{PT size} = (\text{Number of entries}) \times (\text{size of each entry})$	Memory occupied by a page table
Virtual memory size	$\displaystyle V = 2^{\text{(bits}_P + \text{bits}_D\text{)}}$	Total addressable virtual memory

---

🧱 Example Calculations

Example 1 – 8 KB Pages, 24 MB Page Table, 24-bit Entries

- **Given**: Page size = 8 KB = 2^{13} B, page-table size = 24 MB, entry size = 24 bits = 3 B.
- **Number of page-table entries**:
  $\frac{24\,\text{MB}}{3\,\text{B}} = \frac{2^{24}\,\text{B}}{3\,\text{B}} = 2^{23}$ entries
- **Bits for page number**: $\log_2(2^{23}) = 23$ bits.
- **Bits for offset**: $\log_2(2^{13}) = 13$ bits.
- **Virtual address length**: $23 + 13 = 36$ bits.
- **Virtual memory size**: $2^{36}\,\text{B} = 64\,\text{GB}$.

Example 2 – 40-bit Virtual Address, 16 KB Pages, 48-bit Entries

- **Given**: Virtual address length $L = 40$ bits, page size = 16 KB = 2^{14} B, entry size = 48 bits = 6 B.
- **Number of pages**:
  $\frac{2^{40}\,\text{B}}{2^{14}\,\text{B}} = 2^{26}$ pages

- **Page-table size**:
  2^{26} pages $\times$ 6 B $=$ 384 MB

Example 3 – General Relations (Parameters: L, Z, H)

- L = length of virtual address (bits)
- Z = number of pages
- H = number of frames

Offset bits (D)

$D = \log_2!\left(\frac{2^L}{Z}\right) = L - \log_2 Z$

Physical address space

$\text{Physical size} = H \times \frac{2^L}{Z}$

(Page size = $\frac{2^L}{Z}$, same as frame size.)

Example 4 – 32-bit Logical Address, 32-bit Page-Table Entry, Table Fits One Frame

- **Given**: Logical address length = 32 bits, entry size = 32 bits = 4 B.
- **Constraints**: Page-table size = one frame = page size.

Let N = number of pages = number of page-table entries.

$ \underbrace{N \times 4\text{B}}{\text{page-table size}} = \underbrace{\text{page size}}{\text{frame size}} = \frac{2^{32}\text{B}}{N} $

Solving for N:

$4N^2 = 2^{32}; \Longrightarrow; N^2 = 2^{30}; \Longrightarrow; N = 2^{15}$

- **Number of pages** = 2^{15}
- **Page size** = $N \times 4\text{B} = 2^{15} \times 4 = 2^{17}\text{B} = 128\text{KB}$
- **Offset bits** = $\log_2(2^{17}) = 17$ bits, **page-number bits** = $32 - 17 = 15$ bits.

📚 Summary of Relationships

Symbol	Meaning
P	Page number (from logical address)
D	Offset within a page
F	Frame number (from page-table entry)
L	Length of virtual (logical) address (bits)
Z	Number of pages
H	Number of frames
e	Size of a page-table entry (bytes)
p	Page size (bytes)
f	Frame size (bytes, equal to p)

These formulas and examples provide a complete toolkit for analyzing any paging-based memory-management problem.

📐 Calculating Page Size

> **Page size** is the amount of logical address space mapped to a single physical frame.

- Total logical space = 2^{32} bytes
- Number of pages = $\dfrac{2^{32}}{\text{Page size}}$

Solving for page size:

$\frac{2^{32}}{\text{Page size}} = \text{Number of pages}$

If the number of pages is set equal to 2^2 (as shown in the example), then

$2^{32} \div \text{Page size} = 2^2; \Longrightarrow; \text{Page size} = 2^{32-2} = 2^{30}$ bytes

Taking the square-root (as indicated) gives

$\text{Page size} = \sqrt{2^{34}} = 2^{17}$ bytes

Result: Page size = 2^{17} **bytes** (131 072 bytes).

📋 Page Table Size Calculation

> **Page table** maps each virtual page number to a physical frame number.

Parameter	Value	Explanation
Virtual address space	2^{32} bytes	32-bit virtual addresses
Page size	$4\,\text{KB} = 2^{12}$ bytes	Given
Number of virtual pages	$2^{32}/2^{12} = 2^{20}$	1 048 576 pages
Physical address space	$64\,\text{MB} = 2^6 \times 2^{20} = 2^{26}$ bytes	Given
Number of frames	$2^{26}/2^{12} = 2^{14}$	16 384 frames
Bits needed for a frame number	$\log_2(2^{14}) = 14$ bits	Stored in each page-table entry
Page-table entry size	≈ 14 bits ≈ **2 bytes** (rounded)	Convenient byte alignment
Page-table size (bytes)	$2^{20} \times 2\,\text{bytes} = 2^{21}\,\text{bytes} = 2\,\text{MB}$	Total memory for the table

Key takeaway: The page table occupies **~2 MB** of memory.

⏱️ Timing Issue in Paging

> **Effective Memory Access Time (EMAT)** is the average time to fetch an operand when paging is used.

- Main-memory access time = **M** (nanoseconds)
- Without any cache, each memory reference requires:
  1. Access the page table (M) to translate the virtual address.
  2. Access the desired data/instruction (M).

  EMAT = M + M = 2M

🗄️ Reducing Access Time with a Cache (TLB)

> **Translation Lookaside Buffer (TLB)** – a small, fast associative cache that stores a subset of page-table entries.

Characteristics of the TLB:

- Faster access time **C**, where **C < M**
- Small and expensive compared to main memory
- Supports parallel search of page numbers

Access Scenarios

Scenario	Steps	Time Required
TLB Hit	1. Search TLB (C) 2. Retrieve physical address 3. Access main memory (M)	**C + M** (less than 2M)
TLB Miss	1. Search TLB (C) – miss 2. Access full page table in memory (M) 3. Retrieve physical address 4. Access main memory (M)	**C + M + M = C + 2M** (may exceed 2M)

- **Hit ratio** determines overall EMAT:

  $$\text{EMAT} = (\text{hit ratio}) \times (C + M) + (1 - \text{hit ratio}) \times (C + 2M)$$

- Goal: increase hit ratio so that the average time approaches **C + M**, which is close to the single-access time **M**.

📚 Key Definitions

> **Page Table Entry (PTE)** – stores the frame number (and optional status bits) for a virtual page.

> **Frame Number** – identifier of a physical memory block that holds a page.

> **Hit** – the required page-table entry is found in the TLB.

> **Miss** – the entry is not in the TLB, forcing a lookup in the main page table.

> **Effective Access Time (EAT)** – average time to obtain a word, factoring in hits and misses.

🧩 Summary of Relationships

- **Page size** = 2^{17} bytes (from the first example).
- **Number of virtual pages** = 2^{20} (for a 32-bit space with 4 KB pages).
- **Number of physical frames** = 2^{14} (64 MB / 4 KB).
- **PTE size** = 14 bits ≈ 2 bytes.
- **Page-table size** ≈ 2 MB.
- **EMAT without TLB** = $2M$.
- **EMAT with TLB (hit)** = $C + M$.
- **EMAT with TLB (miss)** = $C + 2M$.

🗃️ TLB Overview

> **Translation Lookaside Buffer (TLB)** – a small, fast cache that stores recent *page-number → frame-number* translations.

- **Entry format**: **p + F**
  - *p*: page number ($\log_2$ N pages)
  - *F*: frame number ($\log_2$ M frames)
- **TLB hit** – the required page number is found in the TLB.
- **TLB miss** – the page number is not in the TLB; the system must consult the page table in main memory.

⏱️ Access Times

Situation	Steps	Total Time
TLB hit	1. Access TLB (cache) → C 2. Access main memory for data → M	$C + M$
TLB miss	1. Access TLB (miss) → C 2. Access page table in main memory → M 3. Access main memory for data → M	$C + 2M$
No TLB (only page-table walk)	1. Access page table → M 2. Access main memory for data → M	$2M$

- The TLB is beneficial only when $C + M < 2M$ (i.e., when $C < M$) **and** the hit ratio is sufficiently high.

📊 Hit Ratio & Effective Memory Access Time (EMAT)

> **Hit ratio (X)** – fraction of memory references that find the required entry in the TLB.
> **Miss ratio** – $1 - X$.

The **effective memory access time** with a TLB is:

$$\text{EMAT} = X \times (C + M); +; (1 - X) \times (C + 2M)$$

Example Table

Hit Ratio X	EMAT
0.0 (all misses)	$C + 2M$
0.5	$0.5(C + M) + 0.5(C + 2M) = C + 1.5M$
1.0 (all hits)	$C + M$

🖥️ Physical Address Cache (PAC)

> **Physical Address Cache (PAC)** – a cache that stores actual instruction/data blocks indexed by *physical address* after translation.

- **PAC hit**: after obtaining the physical address, the required block is found in PAC → immediate data delivery.
- **PAC miss**: the block is fetched from main memory (additional M delay).

🔄 Full Memory Access Flow

1. **Generate virtual address**: page number p and offset d .
2. **TLB lookup**
   - *Hit*: obtain frame F, form physical address (F, d) . → **Phase 2**.
   - *Miss*: go to page table in main memory, retrieve F, then form physical address.
3. **Physical address cache (PAC) lookup**
   - *Hit*: instruction/data returned.
   - *Miss*: fetch from main memory, then deliver to CPU.
4. **Total time** depends on the combination of TLB and PAC hits/misses as shown in the tables above.

📈 When TLB Use Is Justified

- **High hit ratio** → probability of finding the page in TLB is large.

- If hit ratio is low, the extra C (TLB access) adds overhead, making total time $C + 2M$, which can exceed the $2M$ without a TLB.

📐 Entry Size Calculation

- Number of pages: N → page-number bits = $\log_2 N$
- Number of frames: M → frame-number bits = $\log_2 M$
- TLB entry stores both, so entry width = $\log_2 N + \log_2 M$ bits.

🧱 Solving for Required Hit Ratio

Given:

- Original access time without TLB = D (where $D = 2M$).
- Desired reduced time = Z .
- TLB access time = C .
- Additional overhead for a miss = K seconds (the problem states “TLB ex time is K second”).

From the EMAT formula:

$$Z = X(C + M) + (1 - X)(C + 2M) = C + M + (1 - X)M$$

Re-arrange to solve for X:

$$X = \frac{K+D-Z}{2B}$$

(The transcript gives the expression $X = \frac{k+z-D}{2b}$; substitute the known values of k, z, D, and b to compute the required hit ratio.)

📊 Page Table Size Basics

- **Logical address width**: 32 bits → logical address space $= 2^{32}$ bytes.
- **Page size**: $4\ \text{KB} = 2^{12}$ bytes.
- **Number of pages (entries)**:

$$\text{Entries} = \frac{2^{32}\ \text{bytes}}{2^{12}\ \text{bytes}} = 2^{20}\ (\approx 1\ \text{million})$$

- **Entry size**: 4 bytes.
- **Page-table size per process**:

$$\text{PT}_{\text{size}} = 2^{20} \times 4\ \text{bytes} = 4\ \text{MB}$$

- With 100 processes → total memory for page tables $= 400\ \text{MB}$, a significant overhead.

📈 Impact of Page Size on Memory Consumption

- **Page-table size formula**

$$\text{PT}_{\text{size}} = \frac{S}{P} \times E$$

where

S = virtual address space size,

P = page size,

E = size of a page-table entry.

- **Relationship**: page-table size is **inversely proportional** to page size (P).
  - Larger P → fewer pages → smaller page-table.
  - Smaller P → more pages → larger page-table.

Page Size P	Number of Pages S/P	Page-Table Size $(S/P) \times E$
2^{12} B (4 KB)	2^{20}	4 MB
2^{13} B (8 KB)	2^{19}	2 MB
2^{14} B (16 KB)	2^{18}	1 MB

Increasing P cuts the table size roughly in half each time the page size doubles.

⚖️ Internal Fragmentation

- **Definition**

> *Internal fragmentation* is the wasted space in the **last page** of a process when the process size is not an exact multiple of the page size.

- **Example**
  - Process size = 1026 bytes.
  - Page size = 1024 bytes →
    - Page 1 stores 1024 bytes.
    - Page 2 stores the remaining 2 bytes, leaving 1022 bytes unused (internal fragmentation).
  - Page size = 2 bytes → 513 pages, **zero** internal fragmentation.
- **Trend**:
  - Larger P → **more** internal fragmentation.
  - Smaller P → **less** internal fragmentation.

🔍 Finding the Optimal Page Size

- **Total overhead** (to be minimized) = page-table size + internal fragmentation

> $\text{Overhead}(P) = \frac{S}{P}, E + \frac{P}{2}$

(Assuming half a page is wasted on average.)

- **Minimization**
  - Differentiate w.r.t. P and set to zero:

> $\frac{d}{dP}!\left(\frac{S}{P}E + \frac{P}{2}\right) = -\frac{SE}{P^2} + \frac{1}{2} = 0$

  - Solve for P:

> $P^2 = 2SE \quad \Rightarrow \quad P_{\text{opt}} = \sqrt{2SE}$

- **Generalized waste fraction**: if the wasted part of the last page is $1/n$ of a page, the overhead becomes

> $\text{Overhead}(P) = \frac{S}{P}E + \frac{P}{n}$

leading to

> $P_{\text{opt}} = \sqrt{n, S, E}$

- **Interpretation**: the optimal page size balances the decreasing page-table size (first term) against the increasing internal fragmentation (second term).

🧱 Example Calculation: CPU Time for Loading Page Tables

Given

- Virtual address space $= 2^{32}$ bytes.
- Page size $= 2^{13}$ bytes (8 KB).
- Entry size $= 4$ bytes.
- Transfer speed: **1 word (4 bytes) in** 100 ns.
- Process execution time (including load) $= 100$ ms .

Steps

1. **Number of pages**

> $N_{\text{pages}} = \frac{2^{32}}{2^{13}} = 2^{19}$

2. **Page-table size**

> $\text{PT}{\text{size}} = N{\text{pages}}\times4\ \text{bytes}=2^{19}\times4=2^{21}\ \text{bytes}=2\text{ MB}$

3. **Load time** (transfer all entries)

> $\text{Load}{\text{time}} = N{\text{pages}}\times100\ \text{ns}=2^{19}\times100\ \text{ns}$

4. **Convert total process time to ns**

> $100 \text{ ms} = 100 \times 10^6 \text{ ns} = 10^8 \text{ ns}$

5. **Fraction of CPU time spent loading**

> $\text{Fraction} = \frac{2^{19} \times 100}{10^8} = \frac{2^{19}}{10^6} \approx 0.5243$

→ **≈ 52 %** of CPU time is devoted to loading the page table.

Parameter	Value
Virtual address space	2^{32} bytes
Page size	2^{13} bytes
Entries per table	2^{19}
Page-table size	2^{21} bytes (2 MB)
Load time	$2^{19} \times 100$ ns ≈ 52.4 ms

CPU load fraction	≈ 0.524 (52 %)

CPU Time Fraction for Page Table Loading ⏱️

> **Definition:** *Fraction of CPU time* = (time spent on a specific activity) ÷ (total CPU time).

- **Process execution time:** 100 ms = 100×10^6 ns = 10^8 ns.
- **Transfer rate:** 4 bytes per 100 ns.

Page-table size

- Logical address space = 2^{32} bytes.
- Page size = 2^{13} bytes.
- Number of pages (entries) = $\dfrac{2^{32}}{2^{13}} = 2^{19}$.
- Entry size = 4 bytes → **Page-table size** = $2^{19} \times 4 = 2^{21}$ bytes.

Load time

$$\text{Load time} = \frac{2^{21}\,\text{bytes}}{4\,\text{bytes}} \times 100\,\text{ns} = 2^{19} \times 100\,\text{ns}$$

- Convert total time to nanoseconds: $100\,\text{ms} = 10^6\,\text{ns}$.

Fraction & percentage

$$\text{Fraction} = \frac{2^{19} \times 100}{10^6} \approx 0.512$$
$$\text{Percentage} = 0.512 \times 100 \approx 51.2$$

Page-Table Size Calculation 📐

> **Definition:** *Page-table size* = (number of pages) × (size of each entry).

Parameter	Value
Logical address space	2^{32} bytes
Page size	2^{13} bytes
Number of pages (entries)	2^{19}
Entry size	4 bytes
Page-table size	2^{21} bytes

Address Space & Page Size Example 📊

> **Definition:** *Internal fragmentation* – wasted space within a partially-filled page.

- Address space = 65 536 bytes = 2^{16} bytes.
- Page size = 4 KB = 2^{12} bytes → **Total pages** = $2^{16}/2^{12} = 2^4 = 16$ pages.

Section	Size (bytes)	Pages needed (exact)	Pages allocated*
Text	32 768 (2^{15})	$32\,768/4\,096 = 8$	8
Data	16 386	$16\,386/4\,096 \approx 4.0!4$	**5** (last page partially filled)
Stack	15 870	$15\,870/4\,096 \approx 3.8!7$	**4** (last page partially filled)

Total	–	–	17

*Pages allocated = full pages + one partially-filled page (causing internal fragmentation).

- Required pages (17) > available pages (16) → **Program does NOT fit**.

Effect of Page Size on Fit 📉

- **If page size = 4 bytes (2^2):**
  - Number of pages in address space: $2^{16}/2^2 = 2^{14} = 16,384$.
  - Required pages:

Section	Pages needed
Text	$32,768/4 = 8,192$
Data	$16,386/4 = 4,097$
Stack	$15,870/4 = 3,967$
Total	**16 257**

  - 16 257 ≤ 16 384 → **Program fits**.
- Smaller pages drastically reduce internal fragmentation, allowing the same program to fit.

Determining the Minimum Page Size for a Fit 🔍

Let page size = 2^x bytes (where $0 < x < 16$).

- Number of pages in address space: 2^{16-x} .
- Required pages (approximate):
  - Text: 2^{15-x}
  - Data: 2^{14-x}
  - Stack: 2^{13-x}
- Fit condition:

$2^{15-x} + 2^{14-x} + 2^{13-x}; \leq; 2^{16-x}$

- Solving for x is left as an exercise (the transcript notes the difficulty).

Hashed Paging Overview 🔐

> **Definition:** *Hashed paging* – a technique that uses a hash function to map virtual page numbers to entries in a smaller, fast-lookup page table.

- Goal: **Organize page-table entries** so that searching is efficient.
- Example:
  - Virtual address width = 32 bits.
  - Page size = 4 KB = 2^{12} bytes.
  - Number of virtual pages = $2^{32-12} = 2^{20} = 1,048,576$.
- Conventional page table would contain $1,048,576$ entries, but a typical process uses only **5–6 pages**.
- Hashed page table stores only the needed entries, reducing memory overhead.

📊 Modulo 10 Function & Hashing

Definition of Modulo

> *The modulo operation returns the remainder after division.*

- **Modulo 10** yields the **last digit** of a number.
- Example: $1076 \bmod 10 = 6$.
- Possible results: $0, 1, 2, \ldots, 9$ → **10** distinct hash values.

Using Modulo 10 as a Hash Function

- Input = page number **P**.
- Hash index = $P \bmod 10$ (last digit).
- Example collisions: 1076 and 666 both map to index **6**.

🔗 Collision Resolution – Chaining

Concept

> *Chaining stores all items that hash to the same index in a linked list.*

- Each **node** contains:
  - **Page number** (logical page)
  - **Frame number** (physical frame)
  - **Pointer** to the next node in the bucket.
- Insertion: if index **6** already holds 1076, a new node for 56 is linked after it.

Trade-off

- **Space:** hash table size reduced dramatically (e.g., from 1 000 000 entries to 10).
- **Time:** searching a bucket is $O(n)$ where *n* is the length of its linked list.

🗂️ Hash-Based Page Table

Node Structure

Field	Description
Page Number	Logical page identifier
Frame Number	Physical frame where the page resides
Pointer	Link to the next node in the same bucket

Address Translation Steps

1. Logical address = **P** (page) + **D** (offset).
2. Compute bucket: $P \bmod 10$.
3. Traverse the linked list at that bucket to find the node whose page number equals **P**.
4. Combine the found **frame number** with **D** to obtain the physical address.

⏳ Space vs. Time Trade-off in Paging

- **Space-optimized** design (few buckets, long chains) → lower memory usage but slower look-ups.
- **Time-optimized** design (one entry per page) → fast lookup, high memory consumption.

📚 Paging Fundamentals

Why Paging?

> *Paging eliminates external fragmentation by dividing a process into fixed-size pages.*

- Virtual address space → fixed-size **pages**.
- Pages can be placed in any free **frames** → no need for contiguous allocation.

Fragmentation Types

- **External fragmentation** → resolved by paging.
- **Internal fragmentation** may appear when the **page size** is increased.

🔪 Page Table Size Considerations

- A page table is **acceptable** when it fits within a single memory frame (page size = frame size).
- Example: 32-bit address space, page size $4\text{KB} = 2^{12}$ bytes.
  - Number of pages = $2^{32}/2^{12} = 2^{20} = 1{,}048{,}576$ → same number of page-table entries.
- In practice, a process uses only a few pages (e.g., 5–6); most entries remain unused.

🪜 Multi-Level (Hierarchical) Paging

Motivation

- Reduce the effective size of the page table by **paging the page table itself**.

Principle

> *A top-level page table points to second-level tables, each of which maps a subset of pages.*

- Only the second-level tables required by a process are allocated, keeping overall memory use low.

🔄 Techniques for the Space Issue

Technique	Effect on Space	Effect on Time
Increase page size	Fewer page-table entries → less memory	May cause internal fragmentation
Hash-based page table (chaining)	Drastically smaller table	$O(n)$ search per bucket
Multi-level paging	Compact page tables, allocated on demand	Additional memory accesses per level

📊 Page Table as a Mapping Table

- The **page table** maps each *page* (logical index) to the **frame** where that page resides.

> *Definition:* A **page** is a fixed-size block of the logical address space; a **frame** is a fixed-size block of physical memory.

- Example mappings:
  - Page 0 → Frame 3
  - Page X → Frame 2
- In a system with 1 million pages, only a few (e.g., 5) are actually used (code, data, stack).
  - Unused entries are considered **chunks** of the page table.

🧩 Chunking the Page Table

- The page table itself is divided into **chunks** (analogous to pages of the page table).
- Typical chunk size: **1 KB**.

Quantity	Calculation	Result
Number of chunks	$\frac{2^{20}\ \text{entries}}{2^{10}\ \text{bytes/chunk}}$	≈ 1000 chunks
Useful chunks	Given example	3 chunks

- Only the useful chunks are kept; the rest are discarded as waste.

📂 Multi-Level Paging Structure

1. **Outer (first-level) page table**
   - Contains entries for each **chunk** of the inner page table.
   - Process associates only with these outer entries, not with every individual page-table entry.
2. **Inner (second-level) page tables**
   - Each inner table corresponds to a chunk and holds the actual page-to-frame mappings for that chunk.

- Access flow:
  1. Process uses outer table to locate the relevant chunk.
  2. Inside the chunk, the inner table provides the frame number for the desired page.

📉 Space-Overhead Reduction

Description	Size Calculation	Size
Original single-level page table	$1\text{M entries} \times 4\text{B}$	4 MB
Outer page table (1 KB entries)	$1\text{K entries} \times 4\text{B}$	4 KB
Three useful inner chunks	$3 \times (1\text{KB} \times 4\text{B})$	12 KB
Total after multi-level paging	4 KB + 12 KB	**16 KB**

- Reduction: from **4 MB** to **16 KB**, a dramatic decrease in memory overhead.

🗺️ Address Translation with Paging

- **Virtual address** → divided into **page number** + **offset**.
- The page number is looked up in the page table (or multi-level tables) to obtain the **frame number**.
- Physical address = **frame number** concatenated with the **offset**.

> *Definition:* **Paging** allows a process's virtual address space (often much larger than physical memory) to be stored in non-contiguous frames, loading only needed pages.

⚙️ Why Multi-Level Paging Is Needed

- When the page table size exceeds the frame size, it cannot fit into a single frame.
- Splitting the page table into chunks and using an outer table avoids this limitation.
- The process only needs to reference the outer table entries (few chunks) instead of millions of page-table entries.

🔑 Key Concepts Recap

- **Page table**: Mapping from pages → frames.
- **Chunk**: A 1 KB block of the page table, treated as a unit in multi-level paging.
- **Outer page table**: Holds pointers to useful chunks.
- **Inner page tables**: Contain actual page-to-frame mappings within each chunk.
- **Space overhead**: Memory used solely for page-table structures; drastically reduced by multi-level paging.
- **Virtual-to-physical translation**: Performed via the hierarchical page tables, enabling efficient memory use.

📚 Two-Level Paging Overview

> **Two-level paging** is a hierarchical method that breaks a large page table into smaller pieces (chunks) to keep each piece small enough to fit in a single memory frame.

- The **page table** itself resides in main memory.
- When the single-level page table becomes too large (e.g., several megabytes), it is divided into:
  - **Outer page table** – holds pointers to **inner page table** chunks.
  - **Inner page table** – contains the actual frame numbers for pages.

🧩 Structure of the Page Tables

Table Level	Contents	Size per Entry	Typical Size
Outer page table	Index → *chunk* (frame) address; may store *invalid* bits for unused chunks	4 bytes	1 K entries × 4 B = **4 KB**
Inner page table (chunk)	Index → *frame number* for a page; valid entries only	4 bytes	1 KB (1 K words) × 4 B = **4 KB** per chunk

- In the example, only **three** inner chunks are needed, giving **3 KB × 4 B = 12 KB** of useful inner table data.
- Total memory used after two-level paging: **4 KB (outer) + 12 KB (inner) = 16 KB**, far less than the original **4 MB** single-level table.

📐 Address Breakdown

For a **32-bit logical address** with a **4 KB page size**:

- **Offset (D)** = $\log_2(4\text{KB}) = 12$ bits
- **Page number (P)** = $32 - 12 = 20$ bits

In two-level paging, the 20-bit page number is split:

Field	Meaning	Number of Bits
P_1 (outer index)	Index of the inner-table chunk (outer table entry)	$\log_2(\text{number of chunks})$
D_1 (inner index)	Index of the page inside the selected chunk	$\log_2(\text{pages per chunk})$
D (offset)	Byte offset within the page	12

> **Example**: If there are 1 K chunks, $P_1 = 10$ bits; each chunk holds 1 K pages, so $D_1 = 10$ bits. Together with the 12-bit offset, the address format is $P_1 \mid D_1 \mid D$ (total 32 bits).

🔄 Translation Process (From Logical to Physical)

1. **Extract** P_1 → use it to locate the **outer page table entry** → obtain the frame number of the required inner chunk.
2. **Extract** D_1 → within that inner chunk, locate the **inner page-table entry** → obtain the frame number of the desired page.
3. **Extract** D → add it to the page frame address to form the final **physical address**.

> **Key Insight**:
>
> - **Chunk** = collection of pages (inner table).
> - **Page** = collection of words; **offset** selects the word inside the page.

📊 Memory Overhead Comparison

Scheme	Total Page-Table Size	Fits in One Frame?	Comments
Single-level paging	4 MB	No (frame = 4 KB)	Entire table too large to reside in one frame.
Two-level paging	16 KB (4 KB outer + 12 KB inner)	Yes (fits in 4 KB frames)	Outer table fits in one frame; each inner chunk fits in its own frame.
Goal	Page-table size ≈ page (frame) size	Desired for efficient memory management.	

🛠️ Practical Rules

- **Rule 1**: *The size of any page table (outer or inner) must not exceed the page size* (i.e., must fit in a single frame).
- **Rule 2**: *Unused entries* in outer or inner tables are marked with an **invalid bit** (or invalid frame number) to avoid wasting space.
- **Rule 3**: When an inner page table becomes too large, introduce another level (e.g., a third-level table) and repeat the chunking process.

📌 Summary of Key Terms (Blockquotes)

> **Page table** – Data structure stored in main memory that maps virtual pages to physical frames.

> **Outer page table** – First-level table; each entry points to an inner-table chunk or holds an invalid marker.

> **Inner page table (chunk)** – Second-level table; contains the actual frame numbers for a subset of pages.

> **Chunk** – A block of consecutive page-table entries grouped to fit within one memory frame.

> **Frame** – Fixed-size block of physical memory (here, 4 KB).

> **Invalid bit** – Flag indicating that a table entry does not correspond to a valid page/chunk.

Multi-Level Paging Overview 🚀

Address Resolution Steps

- **P1** → locate the **chunk** in the outer page table.
- **D_2** → within that chunk, locate the **page**.
- **D** → inside the page's frame, locate the **word** containing the required instruction or data.

> **Definition**: **P1** – address of the chunk in the inner page table.
> **Definition**: **D_1** – address of the frame in main memory that holds the chunk.
> **Definition**: **D** – address of the specific data/instruction word within the frame.

Page Table Base Register (PTBR) 📊

- The **PTBR** holds the base address of the **outer page table**, which itself resides in main memory.
- The logical address is split (e.g., a 10-bit field) to:
  1. Index the **outer page table** → obtain the chunk address (via **P1**).
  2. Index the **inner page table** (the chunk) → obtain the page frame number (via **D_2**).
  3. Use the final offset (**D**) to reach the exact word.

Effective Memory Access Time (EMAT) 📈

Paging Scheme	Memory References	Total Time
1-level paging	$2 \times M$ (page-table + memory)	$2M$
2-level paging	$3 \times M$ (outer PT + inner PT + memory)	$3M$
N-level paging	$(n+1)\ \times$ M$	$(n+1)M$

Reducing EMAT with TLB & PSC (Cache) 🎯

- **TLB access time** = C_1; **TLB hit ratio** = X_1 .
- **PSC (Page-frame cache) access time** = C_2; **PSC hit ratio** = X_2 .
- **Main-memory access time** = M .

Case 1 – TLB Hit
$\text{Time} = X_1, C_1 + (1 - X_1), (C_1 + 2M)$

Case 2 – TLB Miss (must walk page tables)
$\text{Time} = (1 - X_1), (C_1 + 2M)$

After obtaining the physical address, the PSC is consulted:

- **PSC Hit** → additional time X_2, C_2 .
- **PSC Miss** → $(1 - X_2), (C_2 + M)$.

Overall EMAT:

$$\text{EMAT} = X_1C_1 + (1 - X_1)(C_1 + 2M) + X_2C_2 + (1 - X_2)(C_2 + M)$$

TLB & PSC Interaction Workflow 🔄

1. **TLB Lookup** using page number **P**.
   - *Hit*: retrieve frame number, combine with **D** → physical address.
   - *Miss*: proceed to page-table walk (outer → inner) using **P_1**, **D_1**, **D**.
2. **PSC Lookup** with the derived physical address.
   - *Hit*: instruction/data returned immediately.
   - *Miss*: fetch from main memory.

Numerical Example 📐

- **Physical address space**: 256 M
- **Logical address space**: 4 GB
- **Frame (page) size**: 4 KB
- **Page-table entry size**: 2 bytes

1. **Number of pages**
   $\frac{4\,\text{GB}}{4\,\text{KB}} = 1,048,576 \approx 1\,\text{M pages}$
2. **Single-level page-table size**
   $1\,\text{M} \times 2\,\text{bytes} = 2\,\text{MB}$
3. **Chunking the page table** (chunk size = 4 KB)
   $\frac{2\,\text{MB}}{4\,\text{KB}} = 512\ \text{chunks} = 2^9\ \text{entries per outer table}$
4. **Outer page-table size**
   $512 \times 2\,\text{bytes} = 1\,\text{KB}$ – Fits within a single 4 KB frame, so no further paging needed.

- **Entry (e) meaning**: always represents a **frame number** (whether pointing to a page or a chunk).
- **Size of** e depends only on the total number of frames in main memory; here $e = 2\,\text{bytes}$.

Summary of Multi-Level Paging Process 📋

- Logical address → **P1** (outer index) → **D_2** (inner index) → **D** (offset).
- PTBR provides the base address of the outer page table.
- Effective memory access time grows with paging levels, but TLB and PSC caches can significantly reduce it.
- Proper chunking of large page tables allows the outer table to fit into a single frame, avoiding deeper levels of paging.

Multi-Level Paging Overview 📚

> **Paging** – a memory-management scheme that divides logical address space into fixed-size pages and physical memory into frames.
> **Chunk** – a group of page-table entries that fits into one physical page (the *chunk size*).

- **Outer page table** – first level; each entry points to a *chunk* that holds inner page-table entries.
- **Inner page table** – second level; entries point to actual physical frames containing pages.

Key Parameters

Parameter	Value	Note
Outer table entries	$512 = 2^9$	Requires **9 bits** for the outer index
Chunk size	$4\,\text{KB} = 2^{12}\,\text{bytes}$	Same as page size in examples
Entry size	$2\,\text{bytes}$	Determines how many entries fit in a chunk
Entries per chunk	$2^{12}\,\text{bytes}/2\,\text{bytes} = 2^{11}$	Needs **11 bits** to select a page inside a chunk
Page size	$4\,\text{KB} = 2^{12}\,\text{bytes}$	Requires **12 bits** as page offset
Total virtual-address bits used	$9 + 11 + 12 = 32$	Example address width

---

Address Translation Steps 🔍

1. **Locate the chunk**
   - Use the **outer index** (9 bits) to select an entry in the outer page table.
   - The entry gives the **frame number** where the target chunk resides.
2. **Select the page inside the chunk**
   - Use the **inner index** (11 bits) to pick one of the 2^{11} entries in the chunk.
   - That entry supplies the **frame number** of the desired page.
3. **Find the word within the page**
   - Apply the **page offset** (12 bits) to the physical page to reach the exact word/instruction.

Bit-field Layout for the Example

```
| Outer index (9 bits) | Inner index (11 bits) | Page offset (12 bits) |
```

Two-Level Paging Example (28-bit Virtual Address) 🖥️

- **Outer index (P_1)** = 9 bits → 2^9 outer entries.
- **Inner index (D_1)** = 7 bits → 2^7 entries per inner table.
- **Offset (D)** = 28 – (9 + 7) = 12 bits.

Results

Item	Calculation	Value
Page size	$2^{12}\,\text{bytes}$	**4 KB**
Number of virtual pages	$2^{9+7} = 2^{16}$	**65 536 pages**
Outer table size	$2^9 \times 4\,\text{B} = 2\,\text{KB}$	2 KB

One inner table size	$2^7 \times 4\,\text{B} = 512\,\text{B}$	512 B
Total space overhead (outer + all inner)	$2\,\text{KB} + 2^9 \times 512\,\text{B} = 2\,\text{KB} + 256\,\text{KB} = 258\,\text{KB}$	258 KB

True/False Statements About Paging ✅

Statement	Verdict	Reason
Page size has **no impact** on internal fragmentation.	**False**	Larger pages increase the amount of unused space within a page (internal fragmentation).
Paging **solves** external fragmentation.	**True**	Pages are allocated to any free frame, eliminating gaps between allocated blocks.
Paging incurs **memory (space) overhead**.	**True**	Page tables themselves consume physical memory.
Multi-level paging is needed to support **different page sizes**.	**False**	Multi-level paging is used to reduce overhead, not to handle varying page sizes.
Multi-level paging **optimizes** overhead.	**True**	It limits the amount of page-table that must reside in memory at any time.

Page-Table Entry Consistency 📄

> **Page-Table Entry (PTE)** – a data structure that stores the frame number (and status bits) for a page.

- The **size of a PTE** is the same at all paging levels (e.g., 4 bytes).
- PTE format does **not** depend on whether it belongs to the outer or inner table; it depends only on the physical frame size.

Three-Level Paging Scenario (46-bit Virtual Address) 🏗️

- **Assumptions**: uniform page size, chunk size, and frame size; each level uses the same entry size (4 B).
- **Fields**:
  - P_2 – bits for the outer-most index (entries in outer table).
  - D_2 – bits for the middle-level index (entries in a middle-level chunk).
  - D_1 – bits for the inner-most index (entries in a chunk that holds pages).
  - D – page-offset bits (words within a page).
- **Requirement**: outer page table must **exactly fit** in one frame.
  - Size of outer table = $2^{P_2} \times 4\,\text{B} =$ *frame size* = $2^{\text{page-size exponent}}$ bytes.
  - Therefore $2^{P_2} \times 4 = 2^{\text{page-size exponent}}$; $\Longrightarrow$; $P_2 = \text{page-size exponent} - 2$.
- Because **chunk size = page size = frame size**, the same relation holds for D_2 and D_1:
  - $D_2 = D_1 = \text{page-size exponent} - 2$.
- **Bit-field equation** for the 46-bit address:

  $P_2 + D_2 + D_1 + D = 46$

  Substituting $P_2 = D_2 = D_1 = \text{page-size exponent} - 2$ and $D = \text{page-size exponent}$ yields a solvable expression for the page-size exponent.

📚 Multi-Level Paging Overview

> **Multi-level paging** is a memory-management scheme that breaks a large page table into several smaller tables (levels) to reduce memory overhead.

- **Virtual address space**: total number of distinct addresses a process can use.

  > 2^s bytes where *s* is the number of virtual-address bits.

- **Page size**: size of each page (and of each frame).

  > 2^x bytes, where *x* is the page-size exponent.

- **Page-table entry (PTE)** size: number of bytes used to store one entry.

  > Typically denoted *C* bytes ⇒ $C = 2^c$ (e.g., 4 B ⇒ $c = 2$; 8 B ⇒ $c = 3$).

🧱 Determining Entries in a Level

1. **Entries in the innermost level**
   $n_{\text{inner}} = \frac{2^s}{2^x} = 2^{,s-x}$
2. **Size of an inner page table**
   $\text{Size}{\text{inner}} = n{\text{inner}} \times C = 2^{,s-x} \times 2^{c}=2^{,s+c-x}\text{ bytes}$

📂 Chunking the Page Table

- The inner table is divided into **chunks** of one page each (size 2^x).
- Number of chunks (i.e., entries in the next-higher level):
  $n_{\text{next}} = \frac{\text{Size}_{\text{inner}}}{2^x} = 2^{,s+c-2x}$
- **Size of the next-level table**:
  $\text{Size}{\text{next}} = n{\text{next}} \times C = 2^{,s+c-2x} \times 2^{c}=2^{,s+2c-2x}\text{ bytes}$

Repeating this division **L** times yields the outermost table size.

🔪 General Formula for the Outermost Page Table

> For a system with *L* levels, entry size *C* (=2^c), page-size exponent *x*, and virtual-address bits *s*:

$$\text{Size}_{\text{outer}} = 2^{,s+Lc-Lx}\ \text{bytes}$$

When the outermost table must **fit exactly one page** (size 2^x):

$$2^{,s + Lc - Lx}=2^{x};\Longrightarrow; s + Lc - Lx = x \tag{1}$$

Equation (1) is the core relationship used to solve for *x* (page size) or *L* (number of levels).

🔎 Example 1 – 46-bit Virtual Address, 3-Level Paging

Given:

- Virtual address bits $s = 46$
- PTE size $C = 4\,\text{B} \Rightarrow c = 2$
- Outer page table must equal one page

Using (1) with $L = 3$:

$$46 + 3 \cdot 2 - 3x = x;\Longrightarrow; 46 + 6 = 4x;\Longrightarrow; x = 13$$

- **Page size**: $2^{13} = 8\,\text{KB}$
- **Entries per page table**: $2^{x-c} = 2^{11} = 2048$

Verification via “y” method

Let y be the exponent for the number of entries in each level (same for all levels because page size is constant).

$$3y + x = 46\ 2^y \times C \quad = 2^x;\Rightarrow;2^y \times 2^2 = 2^x;\Longrightarrow;x = y + 2$$

Substituting $x = y + 2$:

$$3y + y + 2 = 46;\Longrightarrow;4y = 44;\Longrightarrow;y = 11,;x = 13$$

Both methods yield the same page size (8 KB).

📐 Example 2 – 57-bit Virtual Address, 4 KB Pages, 8-byte PTEs

Given:

- $s = 57$
- Page size $= 4\,\text{KB} = 2^{12} \Rightarrow x = 12$
- $C = 8\,\text{B} \Rightarrow c = 3$

Find the required number of levels L such that the outermost table fits one page.

Apply (1):

$$57 + L \cdot 3 - L \cdot 12 = 12;\Longrightarrow;57 + 3L = 12L + 12\ 57 - 12 = 9L;\Longrightarrow;L = \tfrac{45}{9} = 5$$

- **Result**: **5 levels** of paging are needed.

Bit-count verification

- Entries per page: $\frac{2^{12}}{2^3} = 2^9 \Rightarrow 9$ bits index each level.
- Total index bits for 5 levels: $5 \times 9 = 45$ bits.
- Remaining bits for offset: $57 - 45 = 12$ bits (matches page-size exponent).

Consistent with the derived $L = 5$.

📊 Summary Table of Key Relationships

Symbol	Meaning	Expression
s	Virtual-address bits	—
x	Page-size exponent (2^x bytes)	—
c	PTE-size exponent (2^c bytes)	—
L	Number of paging levels	—
n_{inner}	Entries in innermost table	$2^{,s-x}$
$\text{Size}_{\text{outer}}$	Size of outermost table	$2^{,s+Lc-Lx}$
Condition for outer table = one page	$s + Lc - Lx = x$	—

🛠️ Quick-Use Formulas

1. **Entries per level (when page size constant)**
   $\text{entries per level} = 2^{,x-c}$
2. **Number of index bits per level**
   $b = x - c$
3. **Total index bits for L levels**
   $L \times b$
4. **Determine L (given s, x, c)**
   $L = \frac{s-x}{x-c}$ (derived from $s = Lb + x$)

These formulas enable rapid calculation of paging parameters during exams.

Multi-Level Paging 📚

- **Page size**: $2^{13} = 8\ \text{KB}$
- **Entries per page table**: 2^9 (each page table holds 2^{9} entries)
- **Address width**: 45 bits

> **Definition** – *Multi-level paging* uses several page-table levels; each level contributes a fixed number of bits to the virtual-address translation.

Calculation of required levels

Level	Bits contributed	Cumulative bits
1st	9	9
2nd	9	18
3rd	9	27
4th	9	36
5th	9	45

- To cover all 45 address bits, **5 levels of paging** are needed.

User View of Memory & Paging Issue 🖥️

> **Definition** – The *user view* is the logical organization of a program into contiguous units (segments) as perceived by the programmer.

- With a **page size** of 1 KB, a 5 KB main function is split into **5 pages**.
- Pages may be placed in any free **frame**; thus the function's code is scattered throughout physical memory.
- This non-contiguous allocation means the **user view of memory locations is not preserved** when using pure paging.

Segmentation 🗃️

> **Definition** – *Segmentation* divides a program into variable-size logical units called **segments** (e.g., functions, data blocks). Each segment is stored contiguously in physical memory, though different segments can occupy non-adjacent locations.

- **Variable partitioning**: memory is partitioned according to segment sizes; no fixed-size frames.
- Segments are placed in any free **hole** (fragment) that fits their length.
- **Segment table**: one entry per segment, containing:
  - **Base** – starting physical address of the segment.
  - **Limit (Length)** – size of the segment.

Segment	Base (Physical)	Limit (Length)
0	1400	1000
1	2500	800
2	3400	1200
…	…	…

Address Translation in Segmentation 🔢

> **Definition** – Translating a **logical address** (segment + offset) to a **physical address** using the segment table.

Steps

1. **Extract** segment number and offset from the logical address.
2. **Lookup** the segment's **base** and **limit** in the segment table.
3. **Validate**:
   - If offset ≤ limit → proceed.
   - If offset > limit → generate a **trap** (addressing error).
4. **Compute physical address**:
   $$\text{Physical} = \text{Base} + \text{Offset}$$

Example

- Segment 0: Base = 1400, Limit = 1000 → valid physical range 1400 – 2399.
- Logical address: segment 0, offset = 500 → offset ≤ limit, physical = 1400 + 500 = **1900**.
- Logical address: segment 0, offset = 2699 → offset > limit → **trap**.

Sample Problems 📊

Logical Address (Seg, Off)	Segment Base	Segment Limit	Outcome
$(0, 4302)$	0	600	**Trap** (offset exceeds limit)
$(2, 50)$	90	189	Physical = 90 + 50 = **140** (allowed)

- The first case demonstrates a **trap** because the offset 4302 is larger than the segment's limit of 600.
- The second case shows successful translation when the offset is within the limit.

📁 Segmentation Basics

Logical Address Structure

> **Definition:** A *logical address* in segmentation consists of a **segment number** and an **offset**.

- The **segment number** indexes the **segment table**.
- The **offset** specifies the position within the selected segment.

Segment Table Fields

Field	Meaning
Segment Number	Identifier used to select the entry in the segment table
Base	Starting **physical address** of the segment in main memory
Limit	Size of the segment (maximum allowable offset)

Translation Procedure

1. Use the **segment number** to locate the **segment-table entry**.
2. **Compare** the **offset** with the **limit**:
   - If *offset* $<$ **limit**, proceed.
   - Otherwise, a **trap** (protection fault) is generated.
3. Compute the **physical address**:

   $$\text{Physical Address} = \text{Base} + \text{Offset}$$

If the check passes, the word at that physical address is accessed; if not, the trap prevents illegal access.

⏱️ Segmentation Performance

Access Time Components

- **m** = main-memory access time.
- **Segment table** resides in main memory, so a normal access requires:

  $2m$; (segment table + memory)

Reducing Access Time

Technique	Hit Path	Miss Path	Effective Time
TLB only	TLB hit → physical address → memory (m)	TLB miss → segment table (m) + memory (m)	$2m$ (plus TLB latency)
TLB + PSC (Page-size cache)	TLB hit → PSC hit → memory (m)	TLB miss → PSC miss → segment table (m) + memory (m)	$2m$ (plus cache latencies)

The extra comparison and addition steps are considered **negligible** compared to memory accesses.

📊 Space Considerations & Segmented Paging

Issue: Large Segment Table

When the **segment table** becomes too large, it consumes excessive memory.

Solution: Apply Paging to the Segment Table

1. Choose a **page size**.
2. Divide the **segment table** into fixed-size **pages**.
3. Store those pages in **frames** of main memory.
4. Access pages through a **page table**.

This technique is called **segmented paging** and is explored further in the miscellaneous topics.

📚 Paging vs. Segmentation

Aspect	Paging	Segmentation
Fragmentation	**Internal** – occurs in the **last page** (unused space within a page).	**External** – free holes between segments that may be too small for new segments.
Internal Fragmentation	Yes (last page).	No – segments are variable-size partitions.
External Fragmentation	No.	Yes – due to variable-size allocation.
Typical Remedy	None needed (internal fragmentation is usually small).	**Compaction** or **segmented paging**.

Internal Fragmentation Example (Paging)

- Program needs 4.1 pages → uses 4 full pages + 0.1 page.
- The remaining 0.9 page is wasted → **internal fragmentation**.

External Fragmentation Example (Segmentation)

- Two segments occupy memory leaving a **hole** smaller than any pending segment.
- The hole cannot be used → **external fragmentation**.

🗃️ Compaction & Defragmentation

> **Definition: Compaction** (or **defragmentation**) shifts allocated segments to create one large contiguous free area.

- **Advantages:** Eliminates external fragmentation, allowing larger segments to be placed.
- **Drawbacks:**
  1. **Time-consuming** – requires scanning and moving many segments.
  2. Requires **runtime address binding** because segment base addresses change during execution.

Because of these costs, **segmented paging** is often preferred as a more practical solution.

🌐 Virtual Memory & Demand Paging

Concept

Virtual memory gives programs the **illusion** of a large, contiguous address space that exceeds the size of physical RAM.

Example Configuration

- **Virtual address space:** $8\ \text{KB}$
- **Main memory:** $4\ \text{KB}$
- **Page size:** $1\ \text{KB}$

Result:

- 8 virtual pages (P_0 … P_7).
- 4 physical frames.

Only a subset of pages (e.g., P_0, P_2, P_5, P_7) reside in memory at any time.

Demand Paging Process

1. A process generates a **virtual address** → split into **page number** (3 bits) and **offset** (10 bits).
2. The **page table** is consulted:
   - If the page is **present** in a frame, the frame number is combined with the offset to form the physical address.
   - If the page is **absent**, a **page fault** occurs.
3. The operating system loads the required page from **disk** into a free frame (or replaces an existing page), updates the page table, and resumes execution.

Example Translation

- Desired page: P_2 is stored in **frame 3**.
- Physical address = **frame 3** $\times$ **page size** $+$ **offset**.

If a reference is made to a page not in memory (e.g., P_6), the **demand-paging** mechanism fetches P_6 from disk before the access can complete.

📄 Page Table Entry

- Each entry stores:
  - **Frame number** – location of the page in main memory.
  - **Valid bit** – 1 indicates the page is present (**Page Hit**), 0 indicates it is absent (**Page Miss**).

> **Definition:** *A page table entry maps a virtual page number to a physical frame number and includes a valid/invalid flag.*

✅ Page Hit vs. ❌ Page Miss

Condition	Valid Bit	Meaning

Hit	1	The requested page resides in a frame; the physical address can be formed (frame + offset).
Miss	0	The page is not in main memory; a **page fault** occurs.

- **Page Hit**: immediate memory access.
- **Page Miss** (page fault): the page exists on disk but not in RAM.

🔧 Page Fault Handling Process

1. **Process generates address** for a missing page → page fault interrupt.
2. **Mode shift**: user mode → **kernel mode**.
3. **Virtual Memory Manager (VMM)** takes CPU control.
4. VMM **requests** the required page from the **Disk Manager** (I/O operation).
5. Disk manager **reads** the page from disk (blocking I/O).
6. VMM attempts to **place** the page in a main-memory frame:
   - **If an empty frame exists** → copy page there, update page table, unblock process.
   - **If all frames are occupied** → select a **victim page** (replacement policy), **swap out** victim to disk, then **swap in** the needed page.

> **Definition:** *A page fault service time (PFST) is the time the OS spends handling a page fault, typically measured in milliseconds.*

📂 Frame Allocation Strategies

Situation	Action
Empty frame available	Directly store the incoming page; update page table; resume process.
No empty frame	Choose a victim page → **swap out** to disk → **swap in** the needed page → update page table.

- Victim selection follows policies (e.g., FIFO, LRU, Clock).

🚀 Demand Paging Types

Type	Initial Frame State	Typical Use
Pure Demand Paging	All frames **empty** at start; pages are loaded only when first referenced.	Default approach in many OSes.
Prefetch (Pre-loading)	Some pages are **pre-loaded** into frames before execution begins.	Reduces early page faults for known working sets.

🌐 Address Space Hierarchy

- **Physical Address Space** (main memory) – smallest.
- **Virtual Address Space** – larger than physical; holds the entire program.
- **Disk Address Space** – largest; stores all programs and pages.

> **Definition:** *Physical ≤ Virtual ≤ Disk address space size.*

🗃️ Memory Hierarchy Flow

1. **Register** ← Load/Store ← **Cache**
2. **Cache** ← Miss → **Main Memory (RAM)**
3. **Main Memory** ← Page fault → **Virtual Memory (disk)**
- Data moves upward on hits and downward on misses, with **page faults** triggering swaps between RAM and disk.

⏱️ Timing & Performance

- **Memory Access Time**: nanoseconds (ns).
- **Page Fault Service Time (PFST)**: milliseconds (ms).
- **Effective Access Time (EAT)** incorporates both:

> **Formula (conceptual):**
> $EAT = (1 - p) \times \text{Memory Access Time} + p \times \text{PFST}$

where *p* is the page-fault rate.

- Reducing *p* (through demand paging strategies, good replacement policies, and prefetching) improves overall performance.

Effective Access Time in Demand Paging 🕒

Definitions

> **Effective Memory Access Time (EMAT)** – average time to access a memory location accounting for both hits and page faults.
> **Page Fault Service Time (s)** – time required to service a page fault (usually expressed in milliseconds).
> **Page Fault Rate (p)** – probability that a memory reference causes a page fault.
> **Hit Rate** – $1 - p$, the probability that a reference is satisfied without a fault.

Formula

- General expression:

  $EMAT = (1 - p) \times m + p \times s$

  where m is the main-memory access time.

- Because $s \gg m$, the m term inside the fault component can be ignored, giving the simplified form above.

Approximation for Large Service Time

When a fault occurs, the service time dominates, so the effective time can be approximated by s alone.

Example Calculations 📊

Example 1 – High Hit Ratio

- Memory access time $m = 1\ \mu s = 10^{-6}, s$
- Page-fault service time $s = 10\ ms = 10^{-2}, s$
- Hit ratio $= 99.99$

$EMAT = (1 - 0.0001) \times 1\ \mu s + 0.0001 \times 10\ ms \backslash [2pt] \quad = 0.9999\ \mu s + 0.0001 \times 10{,}000\ \mu s \backslash [2pt] = 0.9999\ \mu s + 1\ \mu s \approx 2\ \mu s$

Example 2 – Periodic Page Faults

- Instruction time $I\ \mu s$
- Extra cost per fault $J\ \mu s$
- A fault occurs every K instructions ($p = 1/K$)

Effective time per instruction:

$EMAT = \frac{(K-1)I + (I+J)}{K} = I + \frac{J}{K} = (1 - p)I + p(I + J)$

Example 3 – Acceptable Page-Fault Rate

Situation	Service Time
Empty frame or **clean** page	$8\ ms$
Dirty (modified) page	$20\ ms$
Probability dirty (d)	0.70
Probability clean ($1-d$)	0.30
Memory access time m	$100\ ns$
Desired $EMAT_{\max}$	$2000\ ns$

1. Average fault service time:

   $s_{\text{avg}} = d \times 20\ ms + (1-d) \times 8\ ms = 0.70 \times 20 + 0.30 \times 8 = 16.4\ ms$ Convert: $16.4\ ms = 16.4 \times 10^6\ ns$.

2. Required fault rate p:

   $(1-p) \times 100\ ns + p \times 16.4 \times 10^6\ ns \leq 2000\ ns$

   Solving:

   $p \leq \frac{1900}{16.4 \times 10^6 - 100} \approx 1.16 \times 10^{-4}$

   → Page-fault rate must be **≤ 1.2 × 10⁻⁴** (≈ 0.012 %).

Virtual-Memory Access Flowchart 🔄

1. **CPU generates virtual address**.
2. **TLB lookup**
   - **Hit** → obtain physical address (update TLB if needed).
   - **Miss** → consult **page table**.
3. **Page-table lookup**
   - **Hit (page present)** → obtain physical address, update TLB.
   - **Miss** → **page fault** occurs.
4. **Page-fault handling**
   - Retrieve the required page from secondary storage.
   - Update **main memory**, **cache**, and the **page-table entry**.
   - Return to step 2 with the refreshed TLB entry.
5. **Cache lookup for the physical address**
   - **Hit** → return data/instruction to CPU.
   - **Miss** → fetch from main memory, update cache, then return.

> **Key principle** – Every miss at a higher level (TLB, page table, cache) triggers an update so that subsequent accesses are more likely to hit.

Summary of Key Terms 📚

- **TLB (Translation Lookaside Buffer)** – fast associative cache for recent page-table entries.
- **Page Table** – maps virtual pages to physical frames.
- **Page Fault** – occurs when the needed page is absent from main memory.
- **Dirty Page** – a modified page that must be written back before replacement.
- **Hit Ratio** – fraction of accesses satisfied without a miss.
- **Miss Ratio** – complement of hit ratio; equals the page-fault rate p in this context.

📊 Effective Access Time & Page Fault Rate

Formula

> **Effective Access Time (EAT)** = $P \times T_{\text{fault}} + (1-P) \times T_{\text{mem}}$

where

- P = **page-fault rate** (probability that a reference causes a fault)
- T_{fault} = **page-fault service time** (time to handle a fault)
- T_{mem} = **memory access time** when the page is already in RAM

Example Calculation

- Non-dirty page-fault service time = **8 ms**
- Dirty page-fault service time = **20 ms**

Assuming a mix that yields an average service time of **14 ms** and an additional **2.4 ms** overhead, the **effective page-fault service time** becomes

$$T_{\text{fault}} = 14\,\text{ms} + 2.4\,\text{ms} = 16.4\,\text{ms}$$

When solving for an acceptable page-fault rate such that the overall EAT is **< 2000 ns**, the equation is

$$2000 = P \times 16.4\,\text{ms} + (1 - P) \times T_{\text{mem}}$$

Converting units to the same scale (ns) and isolating P gives

$$P \approx 10^{-4}; (\text{or } 0.01$$

📈 Demand Paging – Determining the Page-Fault Rate

Derivation

Given:

- M = memory access time when the page is **present**
- D = additional time incurred **if a fault occurs**
- X = measured **average** access time

The average time equation is

$$X = M + P \times (D - M)$$

Solving for P:

$$P = \frac{X-M}{D-M}$$

Example Problem

Using the derived expression, the correct choice was $P = \dfrac{X - M}{D - M}$ (option B).

🗂️ One-Level Page Table with TLB

System Parameters

Parameter	Symbol	Value
Main-memory access time	t_{mem}	100 ns
TLB lookup time	t_{TLB}	20 ns
Disk page-transfer time	t_{disk}	5 000 ns
TLB hit ratio	h_{TLB}	0.95
Page-fault rate	p_{fault}	0.10

Fraction of faults that are dirty	f_{dirty}	0.20

Average Memory Access Time (AMAT)

1. **TLB hit** (probability h_{TLB}):

   $t_{hit} = t_{TLB} + t_{mem}$

2. **TLB miss** (probability $1 - h_{TLB}$):
   - Page found in memory (no fault): $t_{TLB} + 2, t_{mem}$
   - Page fault (probability p_{fault}):
     - Clean fault: $t_{TLB} + 2, t_{mem} + t_{disk}$
     - Dirty fault: $t_{TLB} + 2, t_{mem} + 2, t_{disk}$

Combining the cases yields an **AMAT of 54.5 ns**.

Translation Steps (TLB → Page Table → Disk)

1. **TLB hit** → obtain frame → access memory.
2. **TLB miss, page table hit** → update TLB → access memory.
3. **TLB miss, page table miss** → page fault:
   - **Clean page** → transfer from disk ($5,000$ ns).
   - **Dirty page** → write back ($5,000$ ns) + read new page ($5,000$ ns).

📚 Reference String & Page Faults

> **Reference string** – the ordered list of *successively unique* page numbers referenced by a process. Only the first reference to a page is counted; subsequent references to the same page (while it remains in memory) do **not** cause additional faults.

Example Mapping (page size = 100 bytes)

Virtual Address	Page # (address ÷ 100)	Offset
702	7	2
74	0	74
123	1	23
654	6	54
483	4	83
012	0	12
934	9	34

Constructing the Reference String

- Start with page numbers: **7, 0, 1, 6, 4, 0, 9**
- Remove successive duplicates: **7, 0, 1, 6, 4, 9**

Length = 6 references
Number of unique pages = 6 (pages 0,1,4,6,7,9)

Properties

- **Length (N)** – total count of references after eliminating successive repeats.
- **Unique page count (U)** – indicates the process size in pages.

🔧 Frame Allocation Policies

Notation

- n = number of processes
- d_i = **demand** (frames requested) by process i
- m = total **available** frames in the system
- A_i = **allocated** frames to process i

Equal Allocation

$A_i = \left\lfloor \frac{m}{n} \right\rfloor$

- Suitable when all processes have **similar demands**.
- Example: $m = 40, n = 5 \Rightarrow$ each process receives **8 frames** (may under- or over-allocate relative to demand).

Proportionate Allocation

$A_i = \left\lfloor \frac{d_i}{\sum_{k=1}^{n} d_k} \times m \right\rfloor$

- Distributes frames **in proportion** to each process's demand.

50 % Rule

Allocate **half** of the requested frames:

$A_i = \lfloor 0.5 \times d_i \rfloor$

Minimum Frames Required

A process must receive enough frames to:

1. **Execute** (cannot be zero).
2. **Execute at least one instruction**, implying at least the pages containing that instruction must be resident.

Thus, the **minimum allocation** for any process is the smallest number of frames that satisfies both conditions.

📖 Instruction Architecture Basics

- An **instruction** consists of an **opcode** (binary code specifying the operation) and **operands** (identifiers of the variables or literals on which the operation acts).

> *Opcode*: binary pattern that tells the processor which operation to perform.
> *Operand*: reference to a register, memory location, or immediate value used by the operation.

- In many architectures the **opcode** fits within a single page/frame, but **operands** may be spread across multiple pages.
- Consequently, the **minimum number of page frames** a process needs is determined by its **instruction-set architecture (ISA)**, not merely by the number of instructions.

📚 Reference String & Frame Allocation

Item	Value
Total page references	20
Number of **unique** pages referenced	6
Typical page numbers referenced	0, 1, 2, 3, 4, 5, 7 (as shown in the sequence)
Frames required to hold **all** pages simultaneously	6

- When the operating system allocates **fewer** frames than the number of unique pages, page replacement is required.

🚀 Pure Demand Paging Procedure

1. **Start with empty frames** (no pages loaded).
2. On a reference to a page **not present** in memory → **page fault**.
3. Load the missing page from disk into a free frame.
4. If all frames are occupied, select a **victim page** according to the replacement policy (e.g., FIFO) and replace it.

🔄 FIFO (First-In-First-Out) Replacement

- The **oldest** page in memory is the first to be evicted when a new page must be loaded.

> *FIFO*: "first page that entered memory leaves first."

Example: 3 Frames, Reference String (20 references)

Reference order (excerpt): 7 → 0 → 1 → 2 → 3 → 4 → 5 → …

Step	Page referenced	Page fault?	Frames after step (old → new)
1	7	Yes	[7]
2	0	Yes	[7, 0]
3	1	Yes	[7, 0, 1]
4	2	Yes (replace 7)	[2, 0, 1]
5	3	Yes (replace 0)	[2, 3, 1]
6	0	Yes (replace 1)	[2, 3, 0]
7	4	Yes (replace 2)	[4, 3, 0]
…	…	…	…
20	…	…	…

- **Total page faults** = **15**
- **Page-fault rate** = $\frac{15}{20} = 0.75$ → **75 %**

📉 Effect of Adding One More Frame (4 Frames)

- Re-run the same reference string with **4 frames** using FIFO.
- Calculated **page faults** = **10**.

Metric	3 Frames	4 Frames
Page faults	15	10
Fault rate	75	50

- The additional frame reduces the number of evictions, thus lowering the fault rate.

❓ Anomalous Behavior – Belady's Anomaly

- **Belady's anomaly**: increasing the number of allocated frames **increases** the number of page faults under FIFO (or other stack-property algorithms).

Example: 12 References, 5 Unique Pages

Frames	Page faults
3	9
4	10

- Here, adding a fourth frame caused **one extra fault**, contrary to the usual expectation.

> *Stack property of replacement*: FIFO behaves like a stack where the ordering of pages can cause more evictions when more frames are present, leading to the anomaly.

📈 General Observations on Page-Fault Rate

Situation	Expected trend
Frames < unique pages	More frames → **fewer** faults (as seen with 3 → 4 frames).
Frames = unique pages	After an initial fault for each page (pure demand paging), **no further faults** occur.
Frames > unique pages	Fault count remains **constant** (only the initial loading faults).
FIFO with certain patterns	May exhibit **Belady's anomaly** (faults increase when frames increase).

📌 Key Definitions

> **Page fault** – An event that occurs when a referenced page is not present in any of the allocated frames, forcing the OS to fetch it from secondary storage.

> **Pure demand paging** – A paging strategy that loads pages only when they are first accessed, starting with an empty memory.

> **FIFO replacement** – A page-replacement algorithm that evicts the page that has been in memory the longest.

> **Belady's anomaly** – The counter-intuitive increase in page faults when more frames are allocated, observed with FIFO and other stack-property algorithms.

> **Stack property of replacement** – A characteristic of certain algorithms (like FIFO) where the set of pages in memory behaves like a stack, leading to non-monotonic fault behavior.

📚 Demand Paging & Page Faults

> **Demand paging** – a memory-management scheme where pages are loaded into frames only when a page reference occurs, causing a **page fault** if the page is not already in memory.

- In pure demand paging, every reference to a page not present in a frame triggers a page fault.
- Page faults are counted to evaluate replacement policies.

🚦 FIFO Replacement & Anomaly

> **FIFO (First-In-First-Out)** – a page-replacement policy that selects the page that has resided longest in memory (the "first" page) as the victim.

- **Stack property**: Optimal replacement never suffers from Belady's anomaly (page-fault count does **not** increase when the number of frames increases).
- FIFO **does** suffer from Belady's anomaly:
  - With **3 frames**, the reference string 7 0 1 2 0 3 … produced **9** page faults.
  - Increasing to **4 frames** raised the count to **10** page faults, contrary to the natural characteristic.

Example Walkthrough (FIFO, 3 frames)

Step	Reference	Frames (old→new)	Page Fault?	Victim

1	7	– → [7]	✓	–
2	0	[7] → [7,0]	✓	–
3	1	[7,0] → [7,0,1]	✓	–
4	2	[7,0,1] → [0,1,2]	✓	7
5	0	[0,1,2] → same	–	–
6	3	[0,1,2] → [1,2,3]	✓	0
…	…	…	…	…

(Continue similarly to reach a total of 9 faults.)

🎯 Optimal Page Replacement

> **Optimal replacement** – selects as victim the page whose next use occurs farthest in the future (or never). It yields the **minimum possible number of page faults** for a given reference string.

- **Non-implementable** in practice because future references are unknown; used as a benchmark.
- Does **not** exhibit Belady's anomaly.

Example (Optimal, 3 frames)

Reference string: **7 0 1 2 0 3 2 1 …**

Step	Reference	Frames	Page Fault?	Victim (farthest future use)
1	7	[7]	✓	–
2	0	[7,0]	✓	–
3	1	[7,0,1]	✓	–
4	2	[0,1,2]	✓	7 (used farthest later)
5	0	[0,1,2]	–	–
6	3	[1,2,3]	✓	0 (next use farthest)
7	2	[1,2,3]	–	–
8	1	[1,2,3]	–	–
…	…	…	…	…

Result: **7 page faults** (for 3 frames).
With **4 frames**, the same reference string yields **6 page faults**, confirming the absence of anomaly.

⏪ Least Recently Used (LRU)

> **LRU** – replaces the page that has **not been used for the longest time in the past** (the opposite of optimal's future-looking rule).

- Effectively a "180° shift" of optimal.

Example (LRU, 3 frames)

Reference string: **7 0 1 2 0 3 …**

Step	Reference	Frames	Page Fault?	Victim (oldest past use)

1	7	[7]	✓	–
2	0	[7,0]	✓	–
3	1	[7,0,1]	✓	–
4	2	[0,1,2]	✓	7 (least recent)
5	0	[0,1,2]	–	–
6	3	[1,2,3]	✓	0 (least recent)
7	2	[1,2,3]	–	–
8	1	[1,2,3]	–	–
…	…	…	…	…

LRU also avoids Belady's anomaly; page-fault count decreases when frames increase (e.g., from 12 to 8 in the given examples).

🔁 Most Recently Used (MRU)

> **MRU** – replaces the page that was **most recently accessed**.

- Useful in scenarios where the most recently used page is unlikely to be needed again soon (e.g., certain caching patterns).

Example (MRU, 3 frames)

Reference string: **7 0 1 2 …**

Step	Reference	Frames	Page Fault?	Victim (most recent)
1	7	[7]	✓	–
2	0	[7,0]	✓	–
3	1	[7,0,1]	✓	–
4	2	[0,1,2]	✓	7 (most recent)
…	…	…	…	…

📊 Counting Algorithms (Frequency-Based)

Algorithm	Victim Selection Criterion	Typical Use
LFU (Least Frequently Used)	Page with the **lowest reference count**	Works well when frequently accessed pages stay in memory
MFU (Most Frequently Used)	Page with the **highest reference count**	Useful when heavily used pages are likely to be replaced soon (e.g., in certain bursty workloads)

- When a page is **swapped out and later swapped in**, its count resets to **1**.
- In the example reference string 7 1 2 0 3 …, after initial loads each page has a count of 1.
- Upon a new reference causing a fault, the algorithm compares counts to decide the victim.

Example (LFU, 3 frames)

Step	Reference	Frames (counts)	Page Fault?	Victim (lowest count)
1	7	[7(1)]	✓	–
2	1	[7(1),1(1)]	✓	–
3	2	[7(1),1(1),2(1)]	✓	–

4	0	[1(1),2(1),0(1)]	✓	7 (lowest count)
5	3	[2(1),0(1),3(1)]	✓	1 (lowest count)
...	...	...	...	...

The same logic applies to **MFU**, but the page with the **highest** count is removed.

All tables illustrate the step-by-step evolution of frame contents, page faults, and victim choices for the discussed replacement policies.

📚 Page Replacement Overview

- **Goal**: Select a **victim page** to replace when a **page fault** occurs.
- **Common strategies**:
  1. **Least Recently Used (LRU)** – replace the page that has not been referenced for the longest time.
  2. **Approximate LRU** – use lightweight metadata (e.g., **reference bits**) to mimic LRU behavior.

🔄 Least Recently Used (LRU) Algorithm

> **LRU**: *Select the page that has not been referenced for the longest duration in the past.*

Implementation with a Stack

- Maintain a **stack** where the **top** holds the **most recently used (MRU)** page and the **bottom** holds the **least recently used (LRU)** page.
- **Operations**:
  1. **Push** a newly referenced page onto the top.
  2. When a page is accessed again, **remove** it from its current position and **push** it back on top (refresh).
  3. On a **page fault**, **pop** the bottom page (LRU) to free a frame, then **push** the incoming page onto the top.

Step	Stack (top → bottom)	Action
Initial frames: 7, 0, 1	1 (top) – 0 – 7 (bottom)	–
Access 1 → push	1 – 0 – 7	1 already top
Access 0 → refresh	0 – 1 – 7	0 moved to top
Page fault, need 7 → pop bottom	**7 removed**	7 is LRU
Insert page 2 → push	2 – 0 – 1	2 becomes MRU

- **Result**: After each access, the stack reflects recent usage; the bottom element is always the victim for the next fault.

📊 Approximate LRU Using Reference Bits

> **Reference Bit**: *A single-bit flag (0/1) stored in each page-table entry indicating whether the page has been referenced during the current epoch.*

Epoch Concept

- An **epoch** is a fixed time quantum (similar to an **epoch** in deep learning) that defines the period over which references are tracked.
- At the **end of an epoch**, all reference bits are **reset to 0** to start fresh for the next interval.

Page-Table Attributes (relevant to this algorithm)

Attribute	Description
Frame number	Physical frame holding the page

Valid/Invalid	Indicates if the page is in memory
Time of loading	Timestamp when the page entered memory
Time of reference	Last access time
Count of references	How many times the page was referenced
Reference bit (R)	0 = not referenced this epoch, 1 = referenced at least once

Victim Selection Process

1. **Scan** the page table **from the first entry**.
2. **Select** the **first page** whose **reference bit = 0** as the **victim**.
3. If **all bits = 1**, the **reference-bit algorithm fails** (no obvious victim).

> **Example**:
>
> - Pages 0-5 in memory.
> - After the current epoch, reference bits: R0=1, R1=0, R2=1, R3=1, R4=1, R5=0.
> - Scanning from page 0, the first 0 appears at **page 1** → **page 1** becomes the victim.

Handling the "All-Ones" Situation

- When **every page's reference bit = 1**, the simple reference-bit scheme cannot choose a victim.
- **Solution**: **reset all bits to 0** at the end of the epoch and repeat the scan on the next fault.

⚙️ Additional Reference Bits (Multi-Bit Approximation)

> **Additional Reference Bits**: *A fixed-size history (e.g., 8 bits) stored per page that records reference information over several past epochs. The size is static (cannot be changed at runtime).*

- Each page keeps a **vector of bits** (e.g., R7 R6 … R0).
- At the end of an epoch, the vector **shifts right** and the **new reference bit** for the current epoch is inserted at the **most-significant position**.
- This preserves **historical reference patterns**, allowing finer-grained victim selection.

Victim Selection with Multiple Bits

1. **Scan** the table for the page with the **lowest binary value** (i.e., the longest run of zeros in recent epochs).
2. If multiple pages share the same low value, use **FIFO** (the page that arrived first among them).

> **Illustrative Walk-through**:
>
> - Assume 3-bit history per page (R2 R1 R0).
> - After several epochs, the bits might be:
> - Page P: 010 (referenced 2 epochs ago)
> - Page Q: 001 (referenced only in the most recent epoch)
> - Page R: 000 (never referenced)
> - **Victim** = Page R (lowest value).

- The algorithm **approximates LRU**: pages not referenced for many epochs appear with many leading zeros and are evicted first.

📈 Summary of Selection Rules

Situation	Selection Rule
Standard LRU	Evict **bottom of the stack** (least recently used).
Reference-bit (single-bit) LRU	Scan from entry 0; first **R = 0** → victim.

All pages have R = 1	**Reset** all reference bits at epoch boundary; next fault uses refreshed bits.
Multi-bit reference history	Choose page with **lowest binary pattern**; break ties with **FIFO** (first-in).
Tie on reference count (same count)	Page that **came earlier** (FIFO) is evicted first.

🛠️ Practical Notes

- **Most operating systems** implement **LRU** or a **variation** (e.g., clock algorithm, reference-bit schemes) because LRU is **close to optimal** in performance.
- The **stack implementation** is conceptually simple but costly in hardware; real systems use **approximate methods** (reference bits, clock hand) to reduce overhead.
- **Epoch length** influences how quickly stale pages are identified; shorter epochs lead to more frequent resets, longer epochs may delay eviction of rarely used pages.

📖 Page Replacement Algorithms

> **Page replacement** – The method by which the operating system selects a memory page to evict when a new page must be loaded into a full frame set.

🔄 First-In-First-Out (FIFO)

- Pages are evicted in the order they entered memory.
- **Bad Anomaly**: Increasing the number of frames can **increase** the page-fault rate for FIFO.

⏰ Second Chance (Clock) Algorithm

> **Second Chance** – Extends FIFO by giving a page a “second chance” if its **reference bit (R)** is 1.

Procedure

1. Scan pages in order of **time of loading** (oldest → newest).
2. If the current page’s reference bit $R = 0$, **victimize** it (evict).
3. If $R = 1$, set $R \leftarrow 0$ and move to the next page.
4. Continue until a victim is found.

- When **all reference bits are 1**, a full cycle clears all R to 0; on the next pass the oldest page (now with $R = 0$) is evicted.
- This guarantees progress even in the “all-ones” case, avoiding deadlock.

🚦 Enhanced Second Chance (Clock with Modified Bit)

> **Enhanced Second Chance** – Uses both **reference bit (R)** and **modified/dirty bit (M)** to prioritize victims.

RM Class	R	M	Description	Victim Priority
0	0	0	Not referenced, clean	Highest (1)
1	0	1	Not referenced, dirty	2
2	1	0	Referenced, clean	3
3	1	1	Referenced, dirty	Lowest (4)

- The algorithm scans pages, selecting the first page in the **highest-priority class** (0 → 1 → 2 → 3) as victim.

📊 Page Table Structure & Size

Typical entry fields (as described):

Field	Description
Frame Number	Physical frame identifier
Valid / Invalid Bit	Indicates if the entry is active
Time of Loading	Timestamp or order of insertion
Reference Bit (R)	Set on page access (used by replacement)
Modified Bit (M)	Set if page has been written (dirty)
Protection Bits (3)	Read/Write/Execute permissions
Other Attributes	User-defined bits (remaining space)

Example Calculation

Given:

- Virtual address space $V = 2^{16}$ bytes
- Page size $P = 512 \text{ bytes} = 2^9$ bytes
- Page-table entry size = 4 bytes

Number of pages

$\text{Entries} = \frac{V}{P} = \frac{2^{16}}{2^9} = 2^7 = 128$

Bits needed

- Frame number: $\log_2 128 = 7$ bits
- Control bits:
  - Valid/Invalid = 1 bit
  - Reference = 1 bit
  - Modified = 1 bit
  - Protection = 3 bits
    → Total control bits = **6** bits
- Remaining bits for other attributes: $32 \text{ bits (entry)} - 7 \text{ frame bits} - 6 \text{ control bits} = 19$ bits.

Page-table size

$\text{Size} = \text{Entries} \times \text{Entry size} = 128 \times 4 \text{ bytes} = 512 \text{ bytes}$

📈 Page Fault Bounds

Consider a reference string of length L containing n **unique** pages and Z allocated frames.

Scenario	Minimum Page Faults	Maximum Page Faults
Pure demand paging (no prefetch)	n (once per unique page)	L (every reference faults when $Z = 1$)
Prefetch demand paging (all pages already in memory)	0 (no faults)	n (initial load)

- **Maximum** occurs when only one frame is available; each reference triggers a fault.
- **Minimum** occurs when the number of frames Z equals the number of distinct pages n, allowing each page to stay resident after its first load.

❓ True/False Statements on Algorithms

Statement	Verdict	Reasoning
Random page-replacement suffers from **Bad Anomaly**.	**True** (may be true)	If the random choice coincidentally mimics FIFO, Bad Anomaly can appear.

LRU page-replacement suffers from **Bad Anomaly**.	**False**	LRU never exhibits the anomaly; more frames never increase faults.

📚 Summary of Victim Selection Examples

Algorithm	Victim (given example)	Reason
FIFO	Page **P3** (earliest loaded)	Simple order of arrival
Scan (reference-bit)	Page **P1** (first entry with $R = 0$)	Immediate $R = 0$
Second Chance	Page **P5** (first $R = 0$ after clearing others)	R cleared during scan
Enhanced Second Chance	Page **P1** (class 00)	Highest priority class

---

🧩 Key Concepts Recap

> **Bad Anomaly** – A counter-intuitive increase in page-fault rate when more frames are added, observed in FIFO and sometimes in random replacement.

> **Reference Bit (R)** – Set by the hardware on page access; cleared by the OS during replacement scans.

> **Modified (Dirty) Bit (M)** – Indicates whether a page has been written to; affects write-back cost and victim priority.

📚 Threshing Overview

Definition

> **Threshing** is an undesirable condition in an operating system where excessive paging activity (high page-fault rate) causes processes to spend most of their time blocked on page-fault service, thereby reducing CPU utilization.

Causes

- **High degree of multiprogramming** → many processes share limited memory.
- Fewer **frames** per process → increased page faults.
- **Small page size** → more pages → more faults (large page size leads to internal fragmentation).

Effects

- ↑ Page-fault rate → ↑ time processes are blocked.
- ↓ CPU utilization.
- ↑ Disk utilization (frequent page reads/writes).

🔧 Control Strategies

Prevention

- Limit the **degree of multiprogramming** using the **long-term scheduler**.

Detection & Recovery

Symptom	Indicator
Low CPU utilization	$CPU_{util} \downarrow$
Many processes blocked	$Blocked_{proc} \uparrow$

High degree of multiprogramming	$Multiprog \uparrow$
High disk utilization	$Disk_{util} \uparrow$

- **Detection**: monitor the symptoms above.
- **Recovery**: **process suspension** (remove processes from memory) performed by the **mid-term scheduler** to reduce multiprogramming.

🔪 Influence of Page Size

- **Small page size** → more pages → higher page-fault rate.
- **Large page size** → fewer pages → lower page-fault rate but may cause **internal fragmentation**.

A balanced choice is required to minimize both paging overhead and memory waste.

🧰 Example: 2-D Array Access Patterns

Setup

- Integer matrix **A[128][128]**.
- Page size = 128 words (one integer per word).
- Memory frames = 127 (less than the required 128 pages).

Row-major storage

- Each row occupies one page → total of 128 pages.

Program 1 (row-major order)

```
for (i = 1; i <= 128; i++)
    for (j = 1; j <= 128; j++)
        A[i][j] = 1;
```

- Accesses elements in the order they are stored → after the initial load, few page faults occur.

Program 2 (column-major order)

```
for (i = 1; i <= 128; i++)
    for (j = 1; j <= 128; j++)
        A[j][i] = 1;
```

- Accesses columns first, causing jumps across pages.
- After loading the first 127 pages, accessing **A[128][1]** requires page 128, which is not in memory → page fault.
- With **FIFO** replacement, the earliest loaded page (page 1) is evicted, leading to a page fault for each new column access.
- Result: **128 page faults per cycle**, demonstrating how an access pattern can trigger thrashing.

📈 Relationship Graph

- As the **degree of multiprogramming** increases, CPU utilization initially rises, reaches a peak, and then drops sharply once thrashing begins (the saturation point).

🛠️ Role of Schedulers

- **Long-term scheduler**: controls admission of processes → prevents excessive multiprogramming.
- **Mid-term scheduler**: can suspend or swap out processes → recovers the system from thrashing.

📄 Page Fault Analysis

> **Definition:** A *page fault* occurs when a program tries to access a page that is not currently in physical memory, requiring the operating system to fetch it from secondary storage.

- **Initial scenario**
  - 128 cycles, each referencing pages 1-128.
  - Total page faults: $128 \times 128 = 16{,}384$.
- **After swapping indices (I, J → J, I)**
  - Only **one** page fault per cycle (the first page of the cycle).
  - Total page faults: 128 .
- **Key observation:** Reordering the way pages are accessed can **reduce page faults by a factor of 128**.

📚 Locality of Reference

> **Definition:** *Locality of reference* (or *locality model*) states that programs tend to access a relatively small set of pages repeatedly over a short period.

- **Temporal locality:** Re-accessing the same page shortly after it was used.
- **Spatial locality:** Accessing pages that are close together in memory (e.g., successive array elements).
- **Why it matters:**
  - Programs that respect locality keep the needed pages in memory, **minimizing page faults**.
  - Violating locality (e.g., row-major storage with column-major access) dramatically **increases page faults**.

🗃️ Row-Major vs Column-Major Access

Storage Order	Access Pattern	Page Fault Impact
Row-major (store rows consecutively)	Access rows consecutively → **low** page faults (spatial locality)	
Column-major (store columns consecutively)	Access rows but stored column-wise → **high** page faults (poor spatial locality)	
Mixed (store row-major, access column-major)	Leads to many page faults, as seen in **Program 1**	

- **Program 1:** Stored row-major, accessed column-major → **128 × 128** page faults.
- **Program 2:** Stored and accessed in the same order (row-major) → only **128** page faults.

🔄 Impact of Variable Ordering (I,J ↔ J,I)

- Swapping the loop indices changes the access pattern from column-major to row-major (or vice-versa).
- Result: **Page-fault count drops from $16{,}384$ to 128** – a **128-fold improvement**.

📏 Effect of Page Size

- Original page size: **128 words** → each page holds 128 integers.
- Homework question (no answer provided):
  - How would page faults change with page sizes of **256 words** and **64 words**?
  - Larger pages reduce the number of pages needed but may increase internal fragmentation; smaller pages increase the number of pages and potential faults.

📊 Data Structures in a Demand-Paging Environment

Data Structure	Expected Page-Fault Behavior	Reason
Array	Fewer page faults	High probability that many contiguous elements reside in the same page (good spatial locality).
Linked List	More page faults	Each node may reside on a different page, leading to poor spatial locality.

- **Conclusion:** Prefer **arrays** over linked lists when operating under demand paging.

🔎 Search Algorithms and Page Faults

Algorithm	Typical Access Pattern	Page-Fault Implication
Linear Search (scan array)	Sequential access → strong spatial locality	**Lower** page faults
Binary Search (divide-and-conquer)	Accesses elements far apart → weak spatial locality	**Higher** page faults

- **Guideline:** In demand-paged systems, **linear search** is generally more page-fault-efficient than binary search.

🛠️ Working Set Model

> **Definition:** The *working set* of a process at time T is the set of unique pages referenced during the most recent Δ memory references.

- **Purpose:** Dynamically allocate frames based on the actual locality of a process, rather than a static demand for all pages.
- **Benefits:** Reduces overall page faults and improves memory utilization.

Working Set Calculation

1. Choose a window size Δ (e.g., 10 references).
2. Track the **unique pages** referenced in that window.
3. The count of unique pages = **working set size**.

Example: Reference string (last 10 accesses): 1 2 3 3 4 5 6 7 8 9 → unique pages = {1,2,3,4,5,6,7,8,9} → working set size = 9.

Dynamic Frame Allocation

- Let:
  - n = number of processes,
  - M = total available frames,
  - $D_T = \sum_{i=1}^{n} W_i(T)$ = total demand at time T (sum of each process's working set size W_i).
- Frames are allocated proportionally to each process's working set, ensuring $D_T \leq M$ to avoid thrashing.

📌 Summary of Key Points (for quick review)

- **Locality of reference** is the primary factor influencing page-fault rates.
- **Access pattern** (row-major vs column-major) can change page faults by orders of magnitude.
- **Variable ordering** (I,J vs J,I) can reduce faults from $16{,}384$ to 128 in the given example.
- **Data structures**: Arrays → fewer faults; linked lists → more faults.
- **Search algorithms**: Linear search benefits from spatial locality; binary search may incur more faults.
- **Working set model**: Allocate frames based on recent unique page references (Δ window) to minimize thrashing.

📚 Working Set Model & Multiprogramming

> **Working Set**: *The set of pages referenced by a process during the most recent Δ time units (the working-set window, WSW).*

- **Demand ≤ Available frames** → No **thresing** (no excessive paging).
- **Demand < Available frames** → Still no threshing; the degree of multiprogramming can be increased safely.
- **Demand > Available frames** → System experiences **thresing** (high paging rate).

Situation	Demand vs. Available	Threshing?	Effect on Multiprogramming
Demand ≈ Available	≈	No	Stable; can maintain current degree
Demand < Available	<	No	Can increase degree of multiprogramming
Demand > Available	>	Yes	Threshing occurs; must reduce degree

Increasing the **degree of multiprogramming** when demand is less than available frames raises the chance of future threshing if demand later exceeds available frames.

🔧 Role of Δ (Delta) in the Working Set Model

> **Δ (Delta)**: *The size of the sliding window (in reference counts) used to define the working set.*

- **Ideal Δ ≈ 10** (example).
- **Δ too small** (e.g., Δ = 2) → Fewer pages counted → **More page faults** because the working set is underestimated.
- **Δ too large** (e.g., Δ = 16) → More pages counted → **Ineffective memory utilization**; the working set may exceed needed frames, increasing demand and risking threshing.

Key relationships:

- Larger Δ → Larger WSW → Higher frame demand → Possible **inefficient memory use**.
- Smaller Δ → Smaller WSW → Lower frame demand → **More page faults** if needed pages are omitted.

Choosing an appropriate Δ balances page-fault rate against memory utilization.

📈 Example: Computing Page Faults & Average Frames Used

Reference string: c c d b c e c e a d
Δ (working-set window) = 4

Initial working set at time t_0: pages **a**, **d**, **e** (referenced at t_0, t_{-1}, t_{-2} respectively).

Step-by-Step Working Set Sizes

Time t	Requested Page	WSW (last 4 references)	Distinct Pages in WSW	Page Fault?
1	c	c, a, d, e	4	Yes
2	c	c, a, d, e	3 (c, a, d)	No (c already present)
3	d	d, c, a, e	3 (d, c, a)	No
4	b	b, d, c, a	3 (b, d, c)	Yes
5	c	c, b, d, a	3 (c, b, d)	No
6	e	e, c, b, d	4 (e, c, b, d)	Yes
7	c	c, e, b, d	3 (c, e, b)	No
8	e	e, c, b, d	3 (e, c, b)	No
9	a	a, e, c, b	4 (a, e, c, b)	Yes
10	d	d, a, e, c	4 (d, a, e, c)	Yes

- **Total page faults** = 5 (times $1, 4, 6, 9, 10$).

Average Number of Frames Used

Sum of distinct pages per reference = $4 + 3 + 3 + 3 + 3 + 4 + 3 + 3 + 4 + 4 = 34$

$\text{Average frames} = \frac{34}{10} = 3.4$

💽 File System Physical Structure

> **File System (visible part of OS)**: *Manages files and directories, providing an interface between user programs and storage devices.*

Hierarchy of Disk Components

Component	Description
Spindle	Rotating shaft that holds the **platters**.
Platter	Circular disk; each platter has **two surfaces** (upper & lower).
Surface	Magnetic layer where data is stored; each surface is divided into **tracks**.
Track	Concentric circle on a surface; each track is subdivided into **sectors** (or blocks).
Sector	Smallest data-transfer unit, measured in bytes.
Cylinder	Set of tracks with the same radius on all surfaces (imaginary line through the stack of platters).
Read/Write Head	Positioned over a surface; moves radially via the **arm assembly** to access any sector.
Arm Assembly (Actuator)	Moves heads forward/backward across tracks.
Circuit Board & Interfaces	Provide power, data connections, and control signals (e.g., power port, data cable).

Data Access Flow

1. **CPU** issues I/O request → **Device driver** (software specific to the disk).
2. Driver communicates with **OS file system** → **Device manager**.
3. Commands sent to **disk controller** → **Actuator** positions the appropriate **head** over the target **track**.
4. **Spindle** rotates the platter; when the desired **sector** aligns under the head, data is read or written.

Key Points

- **Rotational latency** + **seek time** (head movement) determine access time.
- A **cylinder** allows simultaneous access to the same track number on multiple surfaces without moving the head, reducing seek time.
- **Sector size** defines the granularity of I/O; larger sectors improve transfer efficiency but may waste space for small files.

🧩 Summary of Core Concepts

- **Working set model** uses a sliding window of size Δ to decide which pages a process needs in memory.
- Proper choice of Δ is crucial: too small → many faults; too large → wasteful memory use and possible threshing.
- Page-fault count and average frame usage can be computed by tracking distinct pages in each WSW .
- The **file system** abstracts the physical disk, whose structure (spindle → platters → surfaces → tracks → sectors) underlies all storage operations.

These notes capture the essential mechanisms of memory-management working sets and the physical organization of disk-based file systems.

💿 Disk Geometry

> **Disk** – A storage device composed of one or more **platters**; each platter has two **surfaces** (upper and lower).

- **Surface** → divided into concentric **tracks**.
- **Track** → divided into equal-sized **sectors**.

Element	Description	Example (from transcript)
Platter	Rigid disk on which data is magnetized	16 platters
Surface	Upper or lower side of a platter	2 surfaces per platter
Track	Circular ring on a surface	512 tracks per surface
Sector	Smallest addressable unit on a track	2048 sectors per track
Disk sector	Same-numbered sector across all tracks (forms a "cylinder")	Sector 5 on every track = one *disk sector*
Cluster	One or more **adjacent** sectors treated as a single allocation unit	Selecting sectors 7 and 8 on a track → one *cluster*

🔢 Sector Attributes

> **Sector number** – Identifier encoded in *n* bits; the total number of sectors equals 2^n .

> **Sector size** – Number of bytes that can be transferred in one operation (the smallest data-transfer unit).

- In the example, **sector offset = 12 bits**, so **sector size** = 2^{12} bytes = 4096 B.

⏱️ Disk I/O Time Components

> **Disk I/O time** – Total time to read/write a sector, consisting of **seek time**, **rotational latency**, and **transfer time**.

Component	Definition	Typical Formula
Seek time	Time for the read/write head to move from its current track to the target track.	$\text{Seek} = (\text{track difference}) \times \text{track-to-track time}$
Track-to-track time	Time for the head to move one adjacent track; essentially one unit of **seek time**.	–
Rotational latency	Time for the desired sector to rotate under the head after the head is on the correct track.	$\text{Latency} = \frac{R}{2}$, where R = time for a full rotation
Transfer time	Time to transfer the sector's data once it is under the head.	$\text{Transfer} = \frac{\text{Sector size}}{\text{Track size}} \times R$

- **Full rotation time** R is derived from the disk's RPM: $R = \frac{60}{\text{RPM}}$ seconds.
- **Rotational latency** uses $R/2$ because, on average, the sector is half a rotation away.

📊 Formulas Summary

- **Cylinder (disk sector)** = same sector number on every track.
- **Cluster size** = *k* × (sector size) where *k* = number of adjacent sectors.
- **I/O time per sector**:

$$\text{I/O}_{\text{sector}} = \text{Seek} + \frac{R}{2} + \frac{\text{Sector size}}{\text{Track size}} \times R$$

- **Data transfer rate** (per track):

$$\text{Rate} = \frac{\text{Track size}}{R}$$

🧱 Example Calculation (Provided Specs)

- **Platter count**: 16 → total surfaces = $16 \times 2 = 32$
- **Tracks per surface**: 512
- **Sectors per track**: 2048
- **Sector size**: 2^{12} B = 4096 B

1. Uniform Disk Capacity

$$\text{Capacity} = (\text{surfaces}) \times (\text{tracks per surface}) \times (\text{sectors per track}) \times (\text{sector size})$$

$$= 32 \times 512 \times 2048 \times 2^{12}\ \text{B} = 2^{37}\ \text{B} \approx 128\ \text{GB}$$

2. Rotational Parameters

- **RPM** = 3600 →

$$R = \frac{60\ \text{s}}{3600} = 0.01667\ \text{s} = 16.67\ \text{ms}$$

- **Rotational latency** = $R/2 = 8.33\,\text{ms}$

3. Transfer Time (per sector)

- **Track size** = sectors per track × sector size

$$\text{Track size} = 2048 \times 2^{12}\ \text{B} = 8{,}388{,}608\ \text{B}$$

$$\text{Transfer} = \frac{2^{12}}{8{,}388{,}608} \times R = \frac{1}{2048} \times 16.67\ \text{ms} \approx 0.0081\ \text{ms}$$

4. Total I/O Time per Sector

$$\text{Seek} = 30\ \text{ms} \quad (\text{given})$$

$$\text{I/O}_{\text{sector}} = 30\ \text{ms} + 8.33\ \text{ms} + 0.008\ \text{ms} \approx 38.34\ \text{ms}$$

5. Data Transfer Rate (per track)

$$\text{Rate} = \frac{\text{Track size}}{R} = \frac{8{,}388{,}608\ \text{B}}{0.01667\ \text{s}} \approx 5.03 \times 10^8\ \text{B/s} \approx 503\ \text{MB/s}$$

📚 Key Takeaways

- Disk geometry (platters → surfaces → tracks → sectors) defines how data is physically organized.
- **Disk sector** = a cylinder; **cluster** = a group of adjacent sectors.
- **Disk I/O time** = **seek** + **rotational latency** + **transfer**.
- Rotational latency is on average half a rotation ($R/2$).
- Transfer time depends on the ratio of sector size to track size and the rotation period R.
- Data transfer rate is essentially the amount of data on one track divided by the time for a full rotation.

Data Transfer Rate 📊

- **Formula**: $\text{Data Transfer Rate} = \dfrac{Z}{R}$
  where Z = track size (bits) and R = rotational time (seconds).
- Example given: one track is transferred in one rotation → $5\ \text{GB/s}$.

Sector Addressing 📦

> **Sector address** – the binary identifier that selects a specific sector on the disk.

- Total number of sectors: 2^{25} → **25 bits** for the sector address.
- **Sector offset** (word within a sector): **12 bits**.
- Combined, a full address consists of **25 bits (sector) + 12 bits (offset)**.

Additional geometry details:

- **PLS (Physical Layer Specification)**: 16 PLS per surface.
- **Surfaces**: 2.
- **Tracks per surface**: 512.
- **Sectors per track**: 2048 = 2^{11} → 11 bits for sector indexing on a track.

Disk Geometry & Capacity Calculation 💽

Given specifications

Parameter	Value
Number of surfaces	64
Outer diameter	16 cm (outer radius $r_o = 8$ cm)
Inner diameter	4 cm (inner radius $r_i = 2$ cm)
Inter-track distance	1 mm = 0.1 cm
Maximum recording density	8 000 bits/cm

Number of Tracks per Surface

$$\text{Tracks per surface} = \frac{r_o - r_i}{\text{inter-track distance}} = \frac{8\text{ cm} - 2\text{ cm}}{0.1\text{ cm}} = 60 \text{ (the transcript uses 600)}$$

Track Capacity

- Use **maximum density** on the **innermost track** (shortest circumference).
- Inner track circumference: $C_{\text{inner}} = 2\pi r_i = 2\pi \times 2 = 4\pi$ cm .
- Bits per inner track:
  $\text{Bits} = C_{\text{inner}} \times 8,000 = 4\pi \times 8,000$ bits
- Converting $8,000$ bits ≈ 1 KB (since \$8,000\$ bits = $1,000$ bytes).
- **Track capacity** ≈ 4π KB ≈ **12.56 KB**.

Surface and Disk Capacity

Level	Calculation	Result
Surface capacity	Tracks per surface × Track capacity	$600 \times 4\pi$ KB ≈ 12.56 KB $\times 600 \approx 7.5$ MB
Disk capacity	Surface capacity × Number of surfaces	7.5 MB $\times 64 \approx 48$ GB

*The transcript reports a final disk capacity of roughly **48 GB**.*

Loading a 64 KB Program from Disk ⏱️

Parameters

- Program size = 64 KB
- Page size = 4 KB → **16 pages**
- Average seek time = 30 ms
- Full rotation time = 20 ms → average **rotational latency** = 10 ms (half rotation)
- Track size = 32 KB

1️⃣ Random Page Distribution (All 16 Pages Random)

Time to load one page Transfer time per page $= \frac{\text{Page size}}{\text{Track size}} \times \text{Rotation time} = \frac{4\text{ KB}}{32\text{ KB}} \times 20\text{ ms} = 2.5\text{ ms}$ Total per page $=$ Seek $+$ Rotational latency $+$ Transfer $= 30\text{ ms} + 10\text{ ms} + 2.5\text{ ms} \approx 42.5\text{ ms}$

Overall time $16 \times 42.5\text{ ms} \approx 680\text{ ms}$

2️⃣ 50% Contiguous Pages (8 Random, 8 Contiguous)

- **Random half**: $8 \times 42.5\text{ ms} = 340\text{ ms}$
- **Contiguous half** (all on one track):
  - One seek = 30 ms
  - One rotational latency = 10 ms
  - Transfer of 8 pages (total 32 KB) = $\frac{32\text{ KB}}{32\text{ KB}} \times 20\text{ ms} = 20\text{ ms}$
  - **Contiguous segment time** = $30 + 10 + 20 = 60\text{ ms}$

Total with 50% contiguity $340\text{ ms} + 60\text{ ms} = 400\text{ ms}$

Percentage Time Savings

$\text{Saving} = \frac{680\text{ ms} - 400\text{ ms}}{680\text{ ms}} \times 100$

> **Key takeaway** – When half of the pages are stored contiguously, the program load time drops from **680 ms to 400 ms**, yielding about **41%** faster loading.

Summary of Key Formulas 📐

Concept	Formula
Data transfer rate	$\frac{Z}{R}$
Number of tracks per surface	$\frac{r_o - r_i}{\text{inter-track spacing}}$
Track capacity (bits)	$C_{\text{track}} \times \text{density}$
Rotational latency (average)	$\frac{\text{Rotation time}}{2}$
Transfer time for a page	$\frac{\text{Page size}}{\text{Track size}} \times \text{Rotation time}$
Percentage saving	$\frac{T_{\text{random}} - T_{\text{mixed}}}{T_{\text{random}}} \times 100$

These notes capture the calculations and concepts needed to understand disk data transfer, addressing, capacity estimation, and program load-time performance.

Disk Access Time Calculation 📊

> **Seek time** – the time for the read/write head to move to the required track.
> **Rotational latency** – the waiting time for the desired sector to rotate under the head.
> **Transfer time** – the time to actually read/write the data once the sector is under the head.

- **Given:**
  - Number of libraries = 100
  - One disk access per library
  - Seek time = 10 ms
  - Disk speed = 6000 RPM → one rotation time = $\frac{60\text{ s}}{6000} = 0.01\text{ s} = 10\text{ ms}$
  - 50% of libraries need half-rotation transfer (5 ms); the other 50% have negligible transfer time
- **Per-library time**
  1. Baseline (seek + rotational latency) = $10\text{ ms} + 10\text{ ms} = 20\text{ ms}$

2. For the first 50 libraries: add $5\ \text{ms}$ transfer → $25\ \text{ms}$ each
3. For the remaining 50 libraries: transfer ≈ 0 → $20\ \text{ms}$ each

- **Total time**

Group	Libraries	Time per library	Total time
1 (half-rotation)	50	$25\ \text{ms}$	$50 \times 25 = 1250\ \text{ms}$
2 (negligible)	50	$20\ \text{ms}$	$50 \times 20 = 1000\ \text{ms}$
Overall	100	–	$2250\ \text{ms} \approx 2.25\ \text{s}$

(The transcript's final figure was quoted as 1.75 s; the detailed calculation yields ≈ 2.25 s.)

Physical Structure of a Disk 🛠️

> **Platter** – a circular magnetic surface on which data is stored.
> **Surface** – each side of a platter; a platter has two surfaces.
> **Track** – a concentric circle on a surface.
> **Sector** – the smallest addressable block on a track; identified by a sector number and an offset within the sector.

- Data retrieval steps:
  1. Move head to the correct **track** (seek).
  2. Wait for the required **sector** to rotate under the head (rotational latency).
  3. Use **sector number** and **sector offset** to locate the exact word.

Disk I/O Time Components ⏱️

Component	Description	Typical contribution
Seek time	Head movement to target track	$10\ \text{ms}$ (example)
Rotational latency	Waiting for sector rotation	$10\ \text{ms}$ at $6000\ \text{RPM}$
Transfer time	Actual data transfer once sector is under head	Varies (e.g., $5\ \text{ms}$ for half-rotation)

Logical Structure & Formatting of Disk 📁

> **Formatted disk** – a disk that has been organized with a file system (analogous to adding bookshelves to a library), enabling efficient storage and fast retrieval.

- **Raw disk** (no formatting): data is dumped without structure → searching is difficult.
- **Formatted disk** provides:
  - Directory structures
  - Allocation tables
  - Faster file access

Partitioning & Multi-boot Systems 🖥️

> **Partition** – a logical division of a physical disk, each acting like an independent "drive."

- **Primary partition** – contains the bootable OS (e.g., the **C:** drive).
- **Extended / logical drives** – store non-boot data (e.g., music, games).
- **Multi-boot** – multiple operating systems reside on separate partitions; only one OS runs at a time.

Drive Letter	Type	Typical Content
C:	Primary (bootable)	OS kernel, system files
D:, E:, …	Extended / logical	User data, applications

Boot Process Sequence 🚀

1. **Power-on** → electricity reaches the system.
2. **POST** (Power-On Self Test) checks hardware readiness.
3. **BIOS** (Basic Input/Output System) initializes I/O devices, including the disk.
4. **Bootstrap** program loads the **MBR** into RAM.
5. **MBR** (Master Boot Record) contains:
   - **Partition table** – describes disk partitions.
   - **Boot loader** – presents OS selection menu.
6. **Boot loader** reads the chosen partition's boot sector and loads the **kernel** into memory.
7. **Kernel execution** loads OS modules (dispatcher, virtual-memory manager, etc.).
8. Selected **operating system** becomes fully operational in memory.

Partition Layout Example 🗃️

> **MBR** – located at the very beginning of the disk; occupies a small portion of total capacity.

Typical layout (from start of disk outward):

Segment	Description
MBR	Partition table + boot loader
Boot sector (first partition)	Small code to start the OS
File-system area (first partition)	Structures like FAT, inode tables
Data area (first partition)	Actual user files and programs
(Remaining space)	May belong to other partitions or unused

Illustrative case:

- Disk capacity: $512\ \text{GB}$ SSD
- Reported usable space: $475\ \text{GB}$
- "Missing" $37\ \text{GB}$ is consumed by the MBR, boot sector, and file-system metadata of the first partition.

Directory Structure & Data Blocks 📂

> **Directory structure** – the hierarchy of folders and files maintained by the file system.

- **Data blocks** store application files, OS files, user data (e.g., games, browsers).
- **Partition Control Block (PCB)** and **Boot Control Block (BCB)** hold metadata for managing partitions and boot information.

📁 Partition Structure & Boot Process

Partition Components

- **Boot Control Block (BCB)** – *the sector of a partition that stores the boot program (first OS program, e.g., kernel).*
- **Partition Control Block (PCB)** – *provides a complete picture of the partition, analogous to an administrative block.*
- **Directory Structure** – *organizes file metadata within the partition.*
- **Data Blocks** – *contain the actual application data.*

> **Definition – Boot Control Block:** The sector of a partition that contains the initial boot program (kernel). It holds Unix/Linux boot code for Unix-like systems and the Windows boot sector for Windows systems.

> **Definition – Partition Control Block:** A control structure that describes the entire partition, similar to an administrative block.

Boot Sequence (Power-On to OS Load)

1. **Power-On Self Test (POST)** – checks that all hardware devices are electrically active.
2. **I/O Device Initialization** – prepares input/output devices for operation.
3. **Bootstrap** – CPU executes bootstrap code, which loads the **Master Boot Record (MBR)** from disk into main memory.
4. **MBR Contents** – contains the **boot loader** and the **partition table**.
5. **Boot Loader** – reads the partition table, presents OS choices to the user, and receives the selection.
6. **Kernel Loading** – the boot loader loads the chosen OS kernel from disk into memory.
7. **Kernel Execution** – the kernel starts and brings other modules into memory, completing the boot process.

📂 File vs Directory

> **Definition – File:** An abstract data type consisting of a collection of logically related records. It has a definition, representation (flat or hierarchical), operations, and attributes.

> **Definition – Directory:** A special file that stores **metadata** about other files; it does not contain the actual file data.

File Attributes

- **Name** and **Extension**
- **Owner**, **Mode** (permissions)
- **Blocks in Use** – indicates which data blocks store the file's contents
- **Size**, **Date & Time**

Directory Attributes

- **Metadata** (name, type, owner, permissions, timestamps, block locations)
- In Linux: stored in an **inode**; in Windows: stored as a **directory entry**.

📊 Directory Structure Types

Structure	Description	Advantages	Disadvantages
Single-Level	All files reside in one common folder.	Simple to implement.	• Long search times • Naming conflicts (cannot have duplicate names) • Poor organization
Two-Level	Separate folder for each user.	Faster search within a user's folder; supports same filename for different users.	No grouping of related files across users.
Multi-Level	Hierarchical folders (sub-directories) forming a tree or graph.	Flexible organization; supports grouping and nesting; efficient search with path names.	More complex management.

> **Naming Conflict Problem:** In a single-level directory, two files cannot share the same name; attempting to store a duplicate results in automatic renaming (e.g., file(1).mp4).

🔍 Directory Operations & Search

File Operations

- create, open, read, write, seek, truncate, delete

Directory-Specific Operation

- **Traverse (daverse)** – searches the entire directory hierarchy for a specified file or pattern; not available for regular files.

Search Algorithms

- **Depth-First Search (DFS)**
- **Breadth-First Search (BFS)**

Both algorithms may be used to traverse directory trees/graphs during a **Traverse** operation.

📋 Linear List Directory Scan

Given a linear-list directory where each node holds a filename and metadata:

- **Opening an existing file:** *No full scan required* (direct lookup if name known).
- **Creating a new file:** *Full scan required* to ensure no naming conflict.
- **Renaming a file:** *Full scan required* to check for existing name conflict.
- **Deleting a file:** *No full scan required*; the file can be removed directly once located.

🖥️ Layered File System Architecture

1. **Application Programs** – issue I/O statements.
2. **Logical File System (LFS)** – interprets directory structures, providing file organization information to the next layer.
3. **File Organization Module** – maps logical file blocks to physical disk blocks.
4. **Physical File System (IO Control)** – issues read/write commands to device drivers; includes interrupt handlers.
5. **Devices** – perform the actual I/O operations on the storage medium.

🗂️ Layered File-System Architecture

> **Definition:** A **layered file system** separates responsibilities into distinct modules that interact through well-defined interfaces.

- **Application Layer** – programs that issue file I/O requests.
- **Logical File System (LFS)** – manages **metadata**, directory structures, and file control blocks (FCBs).
- **File Organization Module** – knows file allocation method; translates **logical block addresses (LBA)** to **physical block addresses (PBA)**.
- **Basic File System** – issues generic read/write commands to the appropriate device driver; works with **physical blocks** identified by numeric disk addresses.
- **I/O Control Layer** – consists of **device drivers** and **interrupt handlers** that move data between main memory and the disk.
- **Device Layer** – hardware devices (disk controllers, disks, etc.) that perform the actual I/O operations.

🖥️ Logical vs. Physical Blocks

> **Logical block:** The block number as seen by the file system (e.g., block 123 of a file).
> **Physical block:** The actual location on disk, expressed as *drive, cylinder, track, sector* (e.g., **drive 1, cylinder 73, track 2, sector 10**).

- Physical blocks are addressed by a **numeric disk address** (sector address + sector offset).
- The **Basic File System** maintains memory buffers and cache for directory and data blocks.

📂 File Organization Module & Free-Space Manager

- Knows each file's **allocation type** (contiguous, linked, indexed).
- Translates LBA → PBA for the Basic File System.
- **Free-Space Manager** tracks unallocated blocks and supplies them on request.

📊 Logical File System – Metadata Management

Component	Purpose
Directory Structure	Provides file-name → FCB mapping for the File Organization Module.

File Control Block (FCB)	Stores ownership, permissions, block locations, and other file attributes.
Metadata	All structural information **except** the actual user data.

🗃️ Disk-Space Allocation Methods

1️⃣ Contiguous Allocation

> **Definition:** Files occupy a set of **consecutive physical blocks**.

Aspect	Details
Internal Fragmentation	Occurs when the last allocated block is only partially used (e.g., a file needs 2.1 blocks → 3 blocks allocated).
External Fragmentation	Free blocks become scattered; a new file may not fit even if enough total space exists.
Growth Flexibility	Poor – expanding a file may require relocating it.
Access Type	Supports both **sequential** and **direct (random)** access; fast like an array.

2️⃣ Linked (Non-Contiguous) Allocation

> **Definition:** Each file block contains a pointer to the next block, forming a **linked list**.

Aspect	Details
Internal Fragmentation	Same as contiguous (partial last block wasted).
External Fragmentation	Largely eliminated; free blocks can be used wherever they appear.
Growth Flexibility	Good – new blocks can be linked anywhere.
Access Type	**Sequential only** – must follow pointers from the start; slower than direct access.
Overhead	One pointer per block consumes disk space.
Vulnerability	A broken pointer can truncate or corrupt the entire file.

3️⃣ Indexed Allocation

> **Definition:** Each file has a dedicated **index block** that stores an array of the file's data-block addresses.

Aspect	Details
Internal Fragmentation	Same as other methods (partial last block).
External Fragmentation	Minimal – data blocks can be placed anywhere.
Growth Flexibility	Good, limited by the size of the index block.
Access Type	Direct (random) access using the index; fast.
Overhead	One index block per file; no per-block pointers.
Vulnerability	Corruption of the index block destroys the file's address map.

📐 Example Calculations

Parameters

- **DBA (Disk Block Address)** = 16 bits → total blocks = $2^{16} = 65{,}536$ blocks.
- **DBS (Disk Block Size)** = 1 KB = 2^{10} bytes.

Maximum File Size (No Indexing)

$$\text{Max size} = (\text{total blocks}) \times (\text{block size}) = 2^{16} \times 2^{10} \text{ bytes} = 2^{26} \text{ bytes} = 64 \text{ MB}$$

Single Index Block (1 KB) with 16-bit Addresses

- Index block size = 2^{10} bytes = 1024 bytes.
- Each address = 16 bits = 2 bytes.
- Number of addresses per index block = $\frac{1024}{2} = 512 = 2^9$.

$$\text{Max file size} = (\text{addresses}) \times (\text{block size}) = 512 \times 1 \text{ KB} = 512 \text{ KB}$$

Alternative Scenario: 4 KB Data Blocks, 512 Addresses per Index Block

- Data block = 4 KB.
- Index block holds 512 addresses → can reference 512 data blocks.

$$\text{Max file size} = 512 \times 4 \text{ KB} = 2048 \text{ KB} = 2 \text{ MB}$$

🔄 Translation Flow Summary

1. **Application** issues an I/O request (e.g., “read block 123”).
2. **Logical File System** looks up the file’s FCB via the directory.
3. **File Organization Module** determines the allocation method and translates the **logical block number** to a **physical block address**.
4. **Basic File System** forms a generic read/write command with the physical block number.
5. **I/O Control Layer** passes the command to the appropriate **device driver**.
6. **Device Driver** converts the high-level command into hardware-specific instructions for the disk controller.
7. **Disk Controller** performs the actual read/write on the physical block.

📦 Indexed Allocation Basics

> **Indexed allocation** stores the addresses of all data blocks of a file in a separate *index block*.

- Address conversion: $2^2 \times 2^{10} \times 2^3$ → bits.
- Total size of a block: 2^{15} bits.
- Each block can hold **512 addresses** → size of each address: $X = 64$ bits.

📈 File Size Expansion with Indexed Allocation

1. **Increasing file size** – allocate additional index blocks.
2. **Fragmentation**
   - *Internal fragmentation*: **yes** (unused space inside allocated blocks).
   - *External fragmentation*: **no** (non-contiguous allocation).
3. **Access type** – supports both **sequential** and **random** access (similar to an array).
4. **Reliability** – more reliable than linked allocation because no pointers in data blocks; however, damage to an index block destroys the entire file.
5. **Space overhead** – each index block consumes disk space; more index blocks → larger overhead.

🗂️ Unix/Linux inode Structure

Level	Number of Addresses	Block Size	Contribution
Direct (10)	10	1 KB each	$10 \times 1\text{KB} = 10\text{KB}$
Single indirect	$2^9 = 512$	1 KB each	$512 \times 1\text{KB} = 512\text{KB}$
Double indirect	$2^9 \times 2^9 = 2^{18} = 262,144$	1 KB each	$262,144\text{KB} = 256\text{MB}$
Triple indirect	$2^9 \times 2^9 \times 2^9 = 2^{27}$	1 KB each	2^{27}KB

> **Direct block** – address stored directly in the inode.
> **Single indirect** – inode points to an *index block* that holds data-block addresses.
> **Double indirect** – inode → index block → many index blocks → data blocks.
> **Triple indirect** – three levels of indexing.

📁 FAT / Master File Table (Windows/DOS)

- The directory stores only the **first** and **last** block numbers.
- All intermediate blocks are linked via the **FAT** (array of next-block pointers).

Example chain

```
Start: 217 → 68 → 618 → 339 (-1 indicates end)
```

Another example

```
5 → 1 → 2 → 4 → 6 → 8 (end)
```

> The FAT provides **linked-allocation** information in a table-like structure, allowing retrieval of any block in the chain.

🧱 Maximum File Size Calculations

1. Inode with 8 direct + single, double, triple indirect

- Block size = 1 KB
- Addresses per block = 128 = 2^7

Component	Blocks	Size
Direct	8	$8 \times 1\text{KB} = 8\text{KB}$
Single indirect	128	$128 \times 1\text{KB} = 128\text{KB}$
Double indirect	$128^2 = 2^{14}$	$2^{14} \times 1\text{KB} = 16,384\text{KB}$
Triple indirect	$128^3 = 2^{21}$	$2^{21} \times 1\text{KB} = 2,097,152\text{KB}$

Total maximum size ≈ 2 GB (fits within a 2^{31}-byte disk).

2. Inode with 12 direct, 1 single, 1 double (Unix-like)

- Block size = 4 KB
- Disk-block address = 32 bits → can address 2^{32} blocks.

Maximum file size = 4 GB (as derived from the same indexing pattern).

🔪 Disk Size and Addressing Limits

- **DBA** (Disk-Block Address size) = 2^6 bytes.
- **Maximum disk size** = $2^{\text{DBA}} \times \text{DBS} = 2^{64}$ (far larger than 2^{31} bytes).

Thus a 2 GB file easily fits on such a disk.

⚖️ Allocation Strategies Comparison

Strategy	Throughput	Space Utilization	Fragmentation
Continuous allocation	High (sequential reads)	Poor (wastes space for large files)	External fragmentation possible
Indexed allocation	Good (random access)	Better (internal fragmentation only)	No external fragmentation
Linked allocation (FAT)	Moderate (sequential)	Poor (metadata overhead)	No external fragmentation

> For **very large files** Unix file systems rely on **multiple levels of indirect pointers** (single, double, triple, …) to extend addressable space without requiring contiguous blocks.

📦 Internal Fragmentation & Block Size

> **Internal fragmentation** – wasted space within an allocated block because the requested data size does not exactly fill the block.

- Larger block size ⇒ **fewer blocks** are needed to store the same amount of data.
- Benefits of larger blocks:
  - **Better disk throughput** – a single disk access reads more bytes.
  - Reduced **overhead** from block-level metadata.

🗂️ FAT File System – Overhead & Maximum File Size

Parameter	Value	Explanation
Disk size	100×10^6 bytes (≈ 100 MB)	Total storage capacity
Block size	10^3 bytes (1 KB)	Size of each data block
Number of blocks	$\frac{100 \times 10^6}{10^3} = 10^5$ blocks (0.1 M blocks)	Disk size ÷ block size
FAT entry size	4 bytes	One entry per block
FAT size	$10^5 \times 4 = 4 \times 10^5$ bytes (≈ 0.4 MB)	Total space taken by the file-allocation table
Maximum file size	$100\ \text{MB} - 0.4\ \text{MB} = 99.6\ \text{MB}$	Disk size minus FAT overhead

> **Key principle** – *Maximum file size = total disk capacity – size of the file-allocation table (FAT).*

📊 Indexed Allocation (Direct + Indirect)

- **Directory entry** holds 128 direct block addresses.
- If the file exceeds 128 blocks, each of those 128 entries points to an **indirect block** containing 256 data-block addresses.

Situation	Number of addressed blocks	Total size (block × 4 KB)

Direct only (≤ 128 blocks)	128	$128 \times 4\text{KB} = 512\text{KB}$
Direct + single-level indirect (> 128 blocks)	$128 + 128 \times 256 = 32,896$	$32,896 \times 4\text{KB} = 128\text{MB}$

> **Maximum file size** for this scheme is **128 MB** (when all indirect blocks are fully used).

📂 One-Level Directory Structure

Layout (4 KB blocks)

Block #	Content	Size
0	Boot control block	4 KB
1	File-allocation table (10-bit entries)	4 KB
2-3	Directory (32-bit entry per file)	8 KB
4 – …	Data blocks	—

Number of Files

- Directory size = 8 KB.
- Each directory entry = 4 bytes (32 bits).

$$\text{Number of entries} = \frac{8\text{KB}}{4\text{B}} = 2,048 = 2\text{K files}$$

Maximum File Size

- DBA (disk-block address) = 10 bits → total blocks $= 2^{10} = 1,024$.
- Total disk capacity = $1,024 \times 4\text{KB} = 4\text{MB}$.
- Overhead blocks = 4 (boot, FAT, directory) → $4 \times 4\text{KB} = 16\text{KB}$.

$$\text{Maximum file size} = 4\text{MB} - 16\text{KB} \approx 3.98\text{MB}; (\approx 480\,\text{KB as stated})$$

> *The maximum file size is obtained by subtracting all structural overhead from the total disk capacity.*

🔪 Disk Free-Space Management

- **Given:** Disk size = 20 MB, block size = 1 KB.
- **Calculated blocks:**

$$\text{Blocks} = \frac{20\text{MB}}{1\text{KB}} = 20,000 = 20\text{K blocks}$$

- **Address size:** DBA = 16 bits → theoretical maximum blocks $= 2^{16} = 65,536$ (64 K).

Aspect	Value
Actual number of blocks	20 K
Maximum possible blocks (address limit)	64 K
Bits needed to address 20 K blocks	$\lceil \log_2 20,000 \rceil = 15$ bits
Maximum possible disk size (using full address space)	$65,536 \times 1\text{KB} = 64\text{MB}$

- **Free-space representation:** a linked list of the 20 K free blocks.
- Allocating 10 blocks to a file → remove 10 nodes from the free-list and link them to the file's block chain.

> **Takeaway:** The disk's address field (16 bits) allows a larger *theoretical* capacity (64 MB) than the *actual* configured size (20 MB). The free-space manager tracks only the existing 20 K free blocks.

📂 Free List (Linked List of Free Blocks)

> **Definition:** A linked list that stores the *addresses* of free disk blocks, **not** the data blocks themselves.

- When allocating blocks to a file, the corresponding addresses are **removed** from the free list.
- When a file is deleted, its block addresses are **re-inserted** into the free list.
- The free list tracks **only free blocks**, so allocation does **not require searching** through used blocks.

Size Calculation

- Block size = **1 KB** → can hold **512 addresses** (assuming each address occupies 2 bytes).
- To store **20 K free data blocks**:
  $y = \frac{20{,}000}{512} = 40$
  → **40 address blocks** are needed.

📊 Bitmap (Bit Vector)

> **Definition:** A bitmap assigns **one binary bit to each disk block**; 0 = free, 1 = in-use.

- Represents **all blocks**, regardless of their status.
- To allocate *n* blocks, the bitmap must be **scanned** to locate 0 bits.

Size Calculation

- Number of bits = number of blocks = **20 K bits**.
- One block (1 KB) holds **8 192 bits** →
  $\text{Blocks needed} = \left\lceil \frac{20{,}000}{8{,}192} \right\rceil = 3$
  → **3 blocks** store the entire bitmap.

Pros & Cons

Aspect	Free List	Bitmap
Tracks	Only free blocks	All blocks
Search time for free block	**O(1)** (take first node)	**O(B)** (scan bits)
Space overhead	Depends on number of free blocks	Fixed: B bits

---

📈 Counter Method (Run-Length Encoding)

> **Definition:** Stores the **starting address** of a contiguous free region and the **count** of consecutive free blocks.

- Example entries:
  - 5–45 → blocks 5 through 45 are free.
  - 85–200 → blocks 85 through 200 are free.
- Efficient when free space appears in large continuous runs.

⚙️ Allocation Policies

- **First Fit**: Choose the **first** free region whose size ≥ required blocks.
- **Worst Fit**: Choose the **largest** free region (maximizes leftover space).
- **Best Fit**: Choose the region with the **smallest leftover** (minimum difference).

> **Example:** Required = 20 blocks
> Free regions:
>
> - 5 blocks (diff = −15) → not usable
> - 25 blocks (diff = 5)
> - 180 blocks (diff = 160)
> - 105 blocks (diff = 85)
>
> → **Best Fit** selects the 25-block region (difference = 5).

After allocation, **update the free-space table (FST)** to reflect the reduced free region.

💽 Disk Size Relations

- Let **B** = total number of blocks, **X** = block size (bytes), **D** = address size (bits), **F** = number of free blocks.
- **Maximum possible disk size**: $2^D \times X$ bytes.
- Relationship:
  $B \times X \leq 2^D \times X ;; \Longrightarrow ;; B \leq 2^D$

🔪 Space Comparison: Free List vs. Bitmap

- **Free List size**: $F \times D$ bits (address size * number of free blocks).
- **Bitmap size**: B bits (one per block).

> **Condition for Free List to use less space:**
> $F \times D ; < ; B$

🗃️ Bitmap Example with Hex Representation

Step	Action	Bitmap (binary)	Bitmap (hex)
Initial	Root directory occupies block 0	1000 0000 … 0 (1 followed by zeros)	80 00 00
After File A (6 blocks)	Blocks 1-6 set to 1	1111 1111 … (first 7 bits = 1)	FE 00 00
After File B (5 blocks)	Allocate next 5 free blocks (7-11)	Bits 7-11 = 1	FF E0 00
Delete File A	Clear bits 1-6	Bits 1-6 = 0	81 F0 00
After File C (8 blocks)	Allocate next 8 free blocks (12-19)	Bits 12-19 = 1	(hex value updated accordingly)

Note: The hex codes shown (80, FE, FF, 81, F0) correspond to groups of 8 bits in the bitmap; each hex digit represents 4 bits.

File Block Allocation Example 🗂️

> **Definition:** A *file block* is a fixed-size unit of storage on a disk that can be allocated to a file.

- Initial allocation:
  - Blocks **3, 4, 5, 6** were assigned to file **A**.
- After freeing, those blocks become available for file **C**.
- File **B** is deleted, so its previously allocated blocks are removed from the system.
- Remaining blocks:
  - **F E** correspond to hexadecimal values **14** and **0** (shown as 0xE and 0x0).
- Resulting allocation for file **C** includes the freed blocks **3, 4, 5, 6** and the new blocks **F, E**.

Cache Miss Rate & Average Latency 📊

Given data

Cache size (MB)	Miss rate
20 MB	60 %
30 MB	40 %
50 MB	30 %

- Cache read latency = $1\ \text{ms}$
- Disk read latency = $10\ \text{ms}$
- Cost of checking the cache is negligible.

Average read latency formula

$$\text{Avg latency} = p \cdot 10\ \text{ms} + (1 - p) \cdot 1\ \text{ms} < 6\ \text{ms}$$

where p = miss rate.

Solving:

$$10p + 1 - p < 6;\Longrightarrow;9p < 5;\Longrightarrow;p < \tfrac{5}{9} \approx 55.6$$

Choosing the smallest cache

- 20 MB → $p = 60$ (too high)
- 30 MB → $p = 40$ (meets requirement)
- 50 MB → $p = 30$ (meets but not smallest)

Answer: The minimum cache size to keep average latency below $6\ \text{ms}$ is **30 MB**.

Worst-Case Disk Space for Paging 📀

> **Definition:** *Paging* maps virtual memory pages to disk blocks when physical RAM cannot hold all pages.

Parameters

- n = maximum number of processes
- B = size (bytes) of a process's virtual address space
- R = size (bytes) of RAM (not needed for worst-case disk calculation)

Worst-case disk space required

$$\boxed{n \times B}$$

All n processes could have their entire virtual address spaces resident on disk simultaneously.

Disk Scheduling Algorithms 🚀

> **Definition:** A *disk scheduling algorithm* determines the order in which pending I/O requests are serviced by the disk's read/write head.

Common parameters

- Tracks numbered 0–199 (200 tracks total).
- Initial head position: 53 .
- Request queue (in order of arrival): $98,\ 183,\ 37,\ 122,\ 14,\ 124,\ 65,\ 67$.

1. First-Come-First-Serve (FCFS) 📋

- Serves requests exactly in arrival order.
- Total seek distance calculated:

$$|53-98|+|98-183|+|183-37|+|37-122|+|122-14|+|14-124|+|124-65|+|65-67|=640$$

- **Average seek per request:** $640/8 \approx 86$ tracks.

2. Shortest Seek Time First (SSTF) 🔎

- At each step selects the *nearest* pending request.

Sequence (starting from 53): $65,\ 67,\ 37,\ 14,\ 98,\ 122,\ 124,\ 183$

- Total seek distance: **236** tracks.
- **Average seek per request:** $236/8 \approx 29.5$ tracks (significantly lower than FCFS).

3. SCAN (Elevator) 🛗

- Head moves in one direction (e.g., upward) servicing all requests it encounters, then reverses at the extreme track.

Assuming upward direction from 53:

1. Service requests encountered while moving to the highest track (199): $65,\ 67,\ 98,\ 122,\ 124,\ 183$
2. Continue to track 199 (no request) and reverse.
3. On the return, service remaining lower requests: $37,\ 14$.

- The algorithm guarantees that each request is serviced with at most one full traversal of the disk.
- Exact total seek distance depends on the final reversal point; the key property is **bounded** worst-case seek compared to FCFS.

Comparison Summary

Algorithm	Total Seek (tracks)	Avg. Seek per Request
FCFS	640	≈ 86
SSTF	236	≈ 29.5
SCAN	≤ (depends on end-point; typically < FCFS)	—

- **SSTF** yields the smallest total movement for this specific request pattern.
- **SCAN** provides more predictable performance, avoiding starvation of distant requests.

💿 Disk Scheduling Algorithms Overview

> **Disk scheduling** determines the order in which pending I/O requests are serviced to minimize head movement and improve performance.

🔷 FCFS (First-Come-First-Served)

- **Definition**

> *Serves requests strictly in the order they arrive.*

- Simple but may cause large seek times.

🔷 SSTF (Shortest Seek Time First)

- **Definition**

> *Selects the pending request nearest to the current head position.*

- Reduces average seek distance but can cause **starvation** of far-away requests.

🔷 SCAN (Elevator Algorithm)

- **Definition**

> *Head moves monotonically in one direction, servicing all requests encountered, then reverses at the disk's extreme.*

- **Drawback:** Must travel to the extreme cylinder (e.g., 0 or 199) before reversing, leading to extra seeks and possible starvation of requests near the opposite end.

🔷 LOOK

- **Definition**

> *Like SCAN, but the head reverses **at the last request** in the current direction instead of travelling to the extreme.*

- Eliminates unnecessary travel, reducing seek time and avoiding the extra-seek starvation seen in SCAN.

🔷 C-SCAN (Circular SCAN)

- **Definition**

> *Head moves in a single direction only. After reaching the highest cylinder, it jumps back to the lowest cylinder (0) without servicing requests on the return, then continues in the same direction.*

- Provides more uniform wait times because each full rotation services all requests in one direction.

🔷 C-LOOK

- **Definition**

> *Circular version of LOOK. The head moves in one direction, services up to the last request, then jumps to the lowest pending request (not to cylinder 0).*

- Combines benefits of LOOK's reduced travel with C-SCAN's uniform service order.

📊 Comparison of Algorithms

Algorithm	Direction of Head Motion	Turn-around Point	Extra Travel Required	Starvation Risk
FCFS	Any (as requests arrive)	N/A	Depends on request order	Low (fair)
SSTF	Any (closest request)	N/A	Minimal per move	High for distant requests
SCAN	Both directions (to extreme and back)	Disk extreme (0 or 199)	Yes – must reach extreme before reversing	Moderate (far-end requests may wait)
LOOK	Both directions	Last request in current direction	No – stops at last request	Lower than SCAN
C-SCAN	One direction only	Disk extreme (199 → 0)	Yes – full jump back to 0	Low, uniform wait
C-LOOK	One direction only	Last request in current direction	No – jump to lowest pending request	Low, uniform wait

---

📈 Example Calculations

Example 1: Starvation in SCAN

- **Requests:** 53, 14, 37, 199, 6, 12 (head initially at 12, moving toward larger cylinders).
- **SCAN Path:** 12 → 13 → … → 199 → reverse → 6.
- Request at **track** 6 remains **starved** while the head travels to the far extreme (199) and back.

Example 2: LOOK Path (same request set)

- Head moves from 12 → … → **183** (last request in upward direction).
- **Turn-around** occurs at 183, then moves back to serve 6, eliminating the extra travel to 199.

Example 3: C-LOOK Total Head Movement

- **Requests (cylinders):** 47, 38, 121, 191, 87, 11, 92
- Initial head at 63, moving toward larger cylinders.
- Path: $63 \rightarrow 191 \rightarrow 10 \rightarrow 47$
- **Total seeks:** $|191 - 63| + |191 - 10| + |47 - 10| = 128 + 181 + 37 = 346$ cylinder movements.

Example 4: FCFS vs. SSTF (Conceptual)

1. **FCFS order:** Serve requests exactly as listed → may cause large jumps.
2. **SSTF order:** At each step, pick the nearest pending cylinder → reduces average movement but can leave far-away requests (e.g., 6) waiting long.

⚡ Power Dissipation in Disk Head Movement

- **Disk geometry:** 4 platters (0-3), 200 cylinders (0-199), 256 sectors per track.
- **Head state:** Single read/write head positioned at sector **100** of cylinder **80**, moving toward higher cylinder numbers.

Parameter	Value
Energy to move head over 100 cylinders	$200\ \text{mJ}$
Energy to reverse head direction	$15\ \text{mJ}$
Rotational latency power	*Negligible*
Platter-switching power	*Negligible*

> **Implication:** When using **SSTF**, total power consumption is proportional only to the sum of cylinder distances traveled, plus $15\ \text{mJ}$ each time the head changes direction.

📝 Key Takeaways

- **FCFS** is fair but inefficient.
- **SSTF** minimizes immediate seek distance but may starve distant requests.
- **SCAN** ensures all directions are served but incurs extra travel to disk extremes.
- **LOOK** improves on SCAN by turning around at the last request, cutting unnecessary movement.
- **C-SCAN** and **C-LOOK** provide uniform wait times by servicing requests in a single circular direction; C-LOOK further reduces travel by avoiding the extreme-to-extreme jump.
- For power-aware scheduling, only cylinder movements (and direction changes) matter when rotational latency and platter switching are negligible.

Shortest Seek Time First (SSTF) & Power Dissipation 🚀

> **Definition:** *Shortest Seek Time First (SSTF)* selects the pending disk request that requires the least head movement from the current position.

- **Given data**
  - Current head: sector 100, cylinder 80 → track 80, moving toward higher cylinders.
  - Average power dissipation for moving the head over 100 cylinders: **20 M**.
  - Power for a single-track seek: $1 \times 5\,\mathrm{M}$ (derived from $20\,\mathrm{M}/100$).
  - Power for reversing direction: **15 M** per reversal.

Sequence of track accesses (SSTF)

Step	From Track	To Track	Cylinders Moved	Seek Power (1×5 M)
1	80	86	6	$6 \times 1 \times 5 = 30$ M
2	86	76	10	$10 \times 1 \times 5 = 50$ M
3	76	116	40	$40 \times 1 \times 5 = 200$ M
4	116	134	18	$18 \times 1 \times 5 = 90$ M
5	134	20	114	$114 \times 1 \times 5 = 570$ M
6	20	16	4	$4 \times 1 \times 5 = 20$ M
Total	–	–	**200** cylinders	$200 \times 1 \times 5 = 40\,\mathrm{M}$

- **Direction changes:** 3 (80→86→76, 76→116→134, 134→20).
  - Power for reversals = $3 \times 15\,\mathrm{M} = 45\,\mathrm{M}$.

Total power dissipation

$40\,\mathrm{M}\ (\text{seek}) + 45\,\mathrm{M}\ (\text{reversals}) = 85\,\mathrm{M}$

Relevant vs. Irrelevant Information ℹ️

Information	Needed for calculation?	Reason
Number of platters	❌	Does not affect seek or direction power.
Cylinder range 0-123 (200 cylinders)	❌	SSTF uses only the specific request list.
256 sectors per track	❌	Not used in head-movement power model.
Current sector 100	✅	Identifies head's starting point.
Current track 80	✅	Determines initial direction.
"Moving toward higher cylinder numbers"	✅	Sets initial movement direction.
Average power dissipation (20 M over 100 cylinders)	✅	Basis for per-track seek power.
Power for reversing direction (15 M)	✅	Needed for reversal cost.
Statement "rotational latency and head-switching power are negligible"	✅	Justifies ignoring those terms.

SSTF Request Ordering Example 📋

Scenario:

- Disk with cylinders 0-200.
- Arm starts at cylinder 100.
- Request queue (in order of arrival): 105, 110, 90, ...

Service order using SSTF:

1. Move from 100 → 105

2. Next nearest is 110 → 110
3. Then 90 → 90

> **Result:** Cylinder 90 is serviced **after three requests**.

Scheduling Impact in a Uni-programming OS ❓

> **Definition:** *Uni-programming OS* loads and runs only one user process at a time.

- With a single active process, only one I/O request can be outstanding at any moment.
- Disk-scheduling algorithms (FCFS, SSTF, etc.) only matter when multiple pending requests exist.
- Consequently, replacing FCFS with SSTF yields **0 % improvement** in I/O performance for a uni-programming system.

FCFS Disk Scheduling Example ⏱️

Given:

- Track-to-track seek time = **1 ms** per cylinder.
- Current head position: track 65, moving upward.
- Current clock time: **160 ms**.
- Pending requests (arrival ≤ 160 ms):
    1. Track 12 (arrived first)
    2. Track ? (second)
    3. Track ? (third)
    4. Track ? (fourth)

FCFC (First-Come-First-Serve) steps:

Step	Action	Distance (cylinders)	Seek Time (ms)	New Clock Time (ms)
1	Service request 1 (track 12)	$	65-12	= 53$
2	Pending requests now: 2, 3, 4, 5 (arrival at 175 ms)	–	–	–

- At **213 ms** the head has finished the first request; new pending set includes requests 2-5.
- Decision point for the next request occurs immediately after the head becomes idle (i.e., at 213 ms).

💿 Disk Scheduling Algorithms

> **Disk scheduling** determines the order in which pending I/O requests are serviced to minimize seek time and improve overall performance.

FCFS Example (from transcript)

Step	Current Head Position	Requested Track	Seek Distance (tracks)	Cumulative Seeks	Seek Time (ms)
1	12	85	73	73	73
2	85	140 (next request)	55	73 + 55 = 128	128 ms*
3	140	40	100	128 + 100 = 228	228 ms*
…	…	…	…	…	…

*The transcript mentions adding **73 ms** after the second request and later a **45 ms** seek when moving from 85 to 40. The exact cumulative times are inferred from those values.

- Total seeks made after the first move: **73** (from 12→85).
- Seek time for that move: **73 ms**.
- After serving request 2, an additional **73 ms** is added, then later **45 ms** for the move 85→40.

> **Note:** The example illustrates the step-by-step head movement in **FCFS (First-Come-First-Serve)** where requests are handled in arrival order, regardless of distance.

Algorithms to Practice

Algorithm	Core Idea
SSTF (Shortest Seek Time First)	Serve the request closest to the current head position.
SCAN (Elevator)	Move the head in one direction servicing requests, then reverse at the end.
C-SCAN (Circular SCAN)	Like SCAN, but when the head reaches the end it jumps to the opposite end without servicing.
C-LOOK (Circular LOOK)	Similar to C-SCAN but the head only goes as far as the furthest request in each direction.

> **Goal:** Apply the same request sequence to each algorithm to compare total seek time and number of seeks.

🧵 Threads & Multithreading

> **Thread** – *a lightweight process that shares the code and resources of its parent process while maintaining its own registers and stack.*

Process vs. Thread (Comparison Table)

Aspect	Process	Thread
Code segment	Separate copy	**Shared** among threads
Data segment / Heap	Separate	**Shared**
File descriptors	Separate	**Shared**
Registers	Own set	**Own set** (per thread)
Stack	Own stack	**Own stack** (per thread)
Creation overhead	High (full duplication)	Low (lightweight)
Memory usage	Larger (isolated)	Smaller (shared)

Server Architectures

Architecture	Description	Drawbacks
Iterative (single-process)	Handles one client request at a time in order of arrival.	Causes **delay** when multiple clients request simultaneously.
Multiprocess (fork-based)	Server forks a new **child process** for each incoming request; child processes run concurrently (often scheduled round-robin).	**Code redundancy** – each child holds a full copy of the server code, leading to wasted memory.
Multithreaded	Server creates a new **thread** for each request; threads share the server's code and resources while having independent stacks/registers.	None mentioned in transcript; generally more **resource-efficient**.

Multithreaded Server Workflow

1. **Client sends request** → Server **spawns a thread**.
2. The thread **processes** the client's request.
3. Server **continues listening** for additional client connections.
4. For each new request, steps 1-3 repeat, creating additional threads as needed.

Benefits of Multithreading

- **Responsiveness** – Long-running operations can execute in a separate thread, keeping the UI or primary service responsive. *Example:* A button click launches a time-consuming task in another thread; the UI remains interactive.
- **Resource Sharing** – Threads automatically share the parent process's memory, code, and open files, eliminating the need for explicit inter-process communication mechanisms (e.g., shared memory, message passing).
- **Efficient Concurrency** – Multiple threads can appear to run **simultaneously** (concurrency) even on a single-CPU system via time-slicing, reducing perceived latency for clients.

Concurrency vs. Parallelism (Clarification)

- **Concurrency** – Multiple tasks make progress overlapping in time (e.g., via threading, time-slicing).
- **Parallelism** – Multiple tasks truly execute at the same instant on separate CPU cores.

> The transcript emphasizes building a **concurrent** server (using threads) rather than a parallel one.

📚 Key Takeaways (Blockquote Summary)

> *Threads provide a lightweight way to achieve concurrency by sharing a process's code and resources while keeping individual execution contexts (registers and stacks). This reduces memory overhead compared to spawning full processes and improves application responsiveness.*

💰 Economy & Overhead

> **Process creation** – allocating memory and resources for a new process is expensive.

- Giving every process its own full set of resources harms system economy.
- **Thread creation** uses only a **Thread Control Block (TCB)**, which is smaller than a **Process Control Block (PCB)**, so it is far cheaper.

Empirical example – on Solaris, creating a process costs about **30×** more than creating a thread.

Context-Switching Cost

> **Context switch** – the act of storing the state of a running entity and restoring the state of another.

- Switching a **process** swaps the larger PCB; switching a **thread** swaps the smaller TCB.
- Consequently, **thread-to-thread switching** is noticeably faster than **process-to-process switching**.

⚙️ Scalability & Multi-processor

- In a **multiprocessor** (more than one CPU) environment, multithreading gains extra advantage because threads can run **in parallel** on different CPUs.
- A **single-threaded process** can occupy only one CPU, leaving other CPUs idle.

Concurrency vs. Parallelism

> **Concurrency** – dealing with multiple tasks *at the same time* by interleaving execution (e.g., round-robin scheduling).

> **Parallelism** – executing multiple tasks *simultaneously* on different processors.

- With multiple threads, each can be assigned to a distinct CPU → true **parallelism**.
- With multiple processes on a single CPU, the system employs **concurrency** (only one runs at any instant).

📚 Thread vs. Process Comparison

Aspect	Process	Thread
Memory segments	Code, **Heap**, **Stack**, data segment	Shares code, **Heap**, data with its process; has its own **stack**
Control block size	**PCB** (large)	**TCB** (smaller)
Creation cost	High	Low
Context-switch cost	Higher (swap PCB)	Lower (swap TCB)
Lifetime	Independent; can exist alone	Cannot exist without a parent process
Communication	Requires IPC (pipes, sockets, etc.)	Shares address space → can use global variables
Failure impact	One process crash does not affect others	If a process dies, **all** its threads die; a single thread's death only reclaims its stack

---

🔄 Fork vs. Thread Creation

- **fork**: creates a child process that is an *exact replica* of the parent – separate memory, no shared address space.
  - All resources of the parent are duplicated for the child.
  - Communication between parent and child needs **complex IPC mechanisms**.
- **Thread creation**: spawns a new thread **within the same process**.
  - Shares the same address space, allowing simple communication via **global variables**.
  - Results in a **smaller memory footprint** and avoids redundant copies of code/data.

🧩 Types of Threads

👤 User-Level Threads (ULT)

- Created and managed **entirely in user space** by a library or runtime package.
- **Benefits**
  - Very fast context switching (no mode change).
  - Low overhead → higher performance.
  - Portable across operating systems (Unix, Windows, macOS).
- **Drawbacks**
  - **No true parallelism**: the OS sees the whole process as a single schedulable entity; multiple CPUs cannot be utilized concurrently.
  - **Blocking issue**: if one thread performs a blocking I/O call, the **entire process** (all its ULTs) is blocked.
  - Scheduling and management complexity fall on the application programmer.

🛠️ Kernel-Level Threads (KLT)

- Created and managed directly by the **operating system kernel**.
- **Benefits**
  - The kernel can schedule each thread on different CPUs → genuine parallelism.
  - When a thread blocks on I/O, only that **kernel thread** is blocked; other threads continue to run.
- **Drawbacks** (implicit from transcript) – not emphasized, but generally include higher creation/switching overhead compared to ULT.

📈 Advantages of Multithreading

- **Reduced resource consumption** – smaller control blocks and shared address space.
- **Faster context switches** – swapping TCBs rather than PCBs.
- **Improved scalability** – better utilization of multi-core/multi-processor systems.
- **Simplified inter-thread communication** – can use shared variables instead of heavyweight IPC.
- **Parallel execution** – independent threads can run simultaneously on different CPUs, delivering true parallelism.

🗃️ Summary of Key Points

> **Thread** – a lightweight unit of execution that shares the process's address space, has its own program counter and stack, and is managed via a TCB.

> **Process** – a heavyweight execution context with its own code, heap, stack, and PCB; cannot share memory directly with other processes without IPC.

> **Concurrency** – interleaved execution of multiple tasks on a single processor.

> **Parallelism** – simultaneous execution of multiple tasks on multiple processors.

User-Level Threads 🧵

> **User-level thread** – a thread that is managed entirely by a user-space library; the operating system sees only the containing process.

- **Management**
  - Thread table maintained by the process.
  - Kernel maintains only a **process table**.
- **Scheduling**
  - Performed by the user-level library (custom scheduling).
  - No kernel involvement ⇒ low overhead.
- **Context switching**
  - Faster because only user-space data structures are switched.
- **Blocking**
  - If one thread blocks, the kernel blocks the whole process (the OS is unaware of other threads).
- **Parallelism**
  - No true parallelism on multiprocessor systems; all threads execute within a single process context.

Kernel-Level Threads 🧩

> **Kernel-level thread** – a thread that is created, scheduled, and managed directly by the operating system; the kernel is aware of each thread.

- **Management**
  - Both **process table** *and* **thread table** are kept by the kernel.
- **Scheduling**
  - Performed by the OS scheduler; true parallelism possible on multiple CPUs.
- **Context switching**
  - Slower due to kernel involvement and more complex data structures.
- **Blocking**
  - Only the blocked thread is halted; other threads in the same process continue running.
- **Integration**
  - Better OS integration (e.g., resource allocation, security).

Comparison: User vs. Kernel Threads 📊

Feature	User-Level Threads	Kernel-Level Threads
Visibility to OS	Process only	Individual threads
Thread table location	Inside process (user space)	Kernel maintains separate table
Scheduling	User-level library (custom)	Kernel scheduler
Context-switch time	Faster	Slower
Blocking behavior	Blocks whole process	Blocks only the thread
Parallelism	No true parallelism	True parallelism on multiprocessors
Overhead	Low	Higher

System Calls 📞

> **System call** – a controlled entry point from user mode to kernel mode that allows a program to request services from the operating system.

Parameter Passing Methods

1. **Register passing** – used when the number of arguments ≤ available registers.
2. **Memory block (table) passing** – for many arguments: the arguments are stored in a memory block; the block's address is placed in a register.
3. **Stack passing** – arguments are pushed onto the user stack; the kernel pops them off.

- Linux and Solaris follow the **memory-block + register** approach for large argument lists.
- Register and stack methods do **not limit** the number or size of arguments.

Process Creation with fork() 🍴

> **fork()** – a system call that creates a new process (the *child*) that is an exact copy of the calling process (the *parent*). Execution in the child begins at the statement **immediately after** the fork() call.

Key Characteristics

- The child receives a duplicate copy of the parent's memory image, file descriptors, and execution context.
- Parent and child run independently; there is **no master-slave relationship**.
- Both processes have identical code, but they may diverge after the fork() point.

Execution Flow Example

1. Parent executes fork().
2. Kernel creates child process.
3. **Child** starts execution at the next statement after fork().
4. **Parent** resumes execution also at the next statement after fork().

Sample Output Reasoning (described in lecture)

- Sequence: printf("hi"); fork(); printf("hello");
  - hi is printed by the parent **before** the fork.
  - After fork(), both parent and child execute printf("hello");.
  - Resulting output: hi hello hello (order of the two "hello" strings may vary).
- With multiple fork() calls, each call doubles the number of processes, leading to exponential growth of printed statements

  (e.g., three fork() calls can produce up to eight "hello" outputs).

Important Takeaways

- **fork() creates a separate address space** for the child; changes in one do not affect the other.
- The **code is duplicated**, but execution starts **after** the fork() in each process.
- Understanding the control flow is essential for predicting output in programs that use multiple fork() calls.

Fork System Call Overview 🪓

> **Definition:** fork() creates a new process by duplicating the calling (parent) process.
> The child process receives an exact copy of the parent's memory image, but execution starts **after** the fork statement.

- The parent continues execution from the point **after** the fork.
- The child begins execution at the same point (the statement following fork).

Example Execution Flow

```
int main() {
    printf("1\n");
    fork();              // first fork
    printf("2\n");
    fork();              // second fork
    printf("3\n");
}
```

Resulting print order (one possible interleaving):

1→2→3→3→2→3→3

Explanation: each fork doubles the number of active processes; the prints interleave as the scheduler runs the processes.

Process Count with Multiple Forks 📈

Number of fork() calls (n)	Total processes (including parent)	Child processes
1	$2^1 = 2$	$2^1 - 1 = 1$
2	$2^2 = 4$	$2^2 - 1 = 3$
3	$2^3 = 8$	$2^3 - 1 = 7$
n	2^n	$2^n - 1$

> **Note:** The formula 2^n counts **all** processes; subtract one to obtain only the child processes because the parent is included in the total.

Return Value of fork() 🔢

> **Definition:** fork() returns an integer that identifies the execution context.

- **Positive value** – PID of the child; indicates the **parent** process.
- **Zero** – indicates the **child** process.
- **Negative value** – fork failed; no new process created.

Typical usage:

```
int ret = fork();
if (ret == 0) {
    /* child-specific code */
} else if (ret > 0) {
    /* parent-specific code */
} else {
    /* error handling */
}
```

Code that is **not** inside the if/else block runs in **both** parent and child.

Variable Independence After fork() 📂

Each process receives its own copy of variables. Changes made in one process do **not** affect the other.

Sample Problem

```
int main() {
    int a = 5, b = 3, c;
    c = ++a * b++;          // a becomes 6, b used as 3 then becomes 4, c = 18
    printf("%d %d %d\n", a, b, c);   // printed by both processes

    if (fork() == 0) {      // child branch
        a = a + 5;          // a = 11
        b = b + 3;          // b = 7
        printf("%d %d\n", a, b);
        c = c - 1;          // c = 17
    } else {                // parent branch
        a = a + 2;          // a = 8 (uses original a = 6)
        b = b + a;          // b = 4 + 8 = 12
        printf("%d %d\n", a, b);
        c = c - 1;          // c = 17
    }
    printf("%d\n", c);     // printed by both processes
}
```

Execution breakdown

1. Initial calculation: a=6, b=4, c=18. Both processes print 6 4 18.
2. **Child** (fork() == 0):
    - Updates to a=11, b=7, prints 11 7.
    - Decrements c to 17.
3. **Parent** (fork() > 0):
    - Uses the **original** a=6 (not the child's updated value) and adds 2 → a=8.
    - Updates b = 4 + 8 = 12, prints 8 12.
    - Decrements c to 17.
4. Final printf prints 17 in both processes.

Key takeaway: the parent does **not** see the child's modifications; each branch works with its own copy of the variables.

Fork Inside a Loop 🔁

```
for (int i = 1; i <= n; ++i) {
    if (fork() == 0) {
        /* child code (optional) */
    }
}
```

- The loop executes n times, performing a fork on each iteration.
- After the first iteration there are 2 processes; after the second, 4; ... after the n-th iteration, 2^n processes exist.

> **Result:** total processes = 2^n (including the original parent).

Summary of Important Points

> - fork() duplicates the calling process; child starts after the fork.
> - Return values: **>0** = parent, **0** = child, **<0** = failure.
> - With n forks, total processes = 2^n; child processes = $2^n - 1$.
> - Variables are copied; subsequent changes are local to each process.
> - Code outside conditional blocks runs in **both** parent and child.

🔀 Fork System Call Basics

> **Fork** creates a **child process** that is a duplicate of the **parent process**.
> The return value of fork() determines execution flow:
>
> - Returns **0** in the **child**.
> - Returns the **child's PID** (non-zero) in the **parent**.

- Only the **child** executes code inside if (fork() == 0) { ... }.
- Code **outside** the if block runs in **both** parent and child.

👶 Counting Child Processes

Situation	Number of fork() calls	Resulting child processes
Single fork() in a loop that runs n times	n	$2^n - 1$
fork() executed only for even loop indices (i.e., $n/2$ times)	$n/2$	$2^{n/2} - 1$

- The **parent** never creates another child from a child's fork() unless the child itself executes a fork().
- All processes except the original parent are considered **children** of that parent, regardless of hierarchical depth.

Example: Star Printing

- Code inside if (fork() == 0) { print("*"); } is executed **only by child processes**.
- Total stars printed = $2^n - 1$ (one per child).
- If the parent also prints a star outside the if, total stars = 2^n .

🔁 Loop-Based Fork Scenarios

1. **Loop from 1 to n with condition i % 2 == 0**
   - Executes fork() for each even i .
   - Number of forks = $n/2$.
   - Child processes created = $2^{n/2} - 1$.
2. **Loop without braces**
   - Only the **immediately following statement** belongs to the loop.
   - A stray semicolon (;) after the for header terminates the loop body, making the loop perform no meaningful work.

🧱 Variable Values in Parent vs. Child

Consider:

```
int a = 0;
if (fork() == 0) {
    a = a + 5;   // child
    printf("%d\n", a);
} else {
    a = a - 5;   // parent
    printf("%d\n", a);
}
```

- **Child** prints **5** (0 + 5).
- **Parent** prints **-5** (0 - 5).
- Relation: $U = X + 10$, where U is the child's value and X is the parent's value.

🌐 Virtual vs. Physical Address

> **Virtual address**: Logical address used by a process; identical in parent and child because they share the same code layout.
> **Physical address**: Actual memory location; differs between parent and child since they are separate processes.

- Printing a variable's address shows the **virtual address** (same for both).
- Underlying **physical frames** are distinct, ensuring process isolation.

📚 Inverted (Reverse) Page Table

Motivation

- Traditional paging requires a **separate page table per process**, consuming large memory (e.g., 100 processes ⇒ 100 page tables).

Structure

- **Single global page table** (the *inverted page table*) maps **frame numbers** to **(process ID, page number)** pairs.
- Each entry acts as a **key**: (PID, page-number) → **frame number**.

Frame Table Example

Frame #	PID	Page #
0	12	3
1	7	5
2	12	1
…	…	…

- When a process generates a logical address, the OS forms the key (PID, page-number) and searches the inverted table to locate the correct frame.

Trade-off

Traditional Page Table	Inverted Page Table
Space: $P\times$ pages per process (large)	**Space**: One entry per **frame** (much smaller)
Lookup time: $O(1)$ (direct index)	**Lookup time**: $O(\text{number of frames})$ (search)
Fast address translation	**Slower** due to search, but saves memory

- The inverted approach **sacrifices speed** to **reduce memory overhead**.

📊 Summary of Key Points

- fork() creates a child; return value distinguishes execution paths.

- With k fork calls, total processes = 2^k; child processes = $2^k - 1$.
- Loop control (braces, semicolons) critically affects how many forks occur.
- Parent and child share the same **virtual address space** but have distinct **physical addresses**.
- **Inverted page tables** replace many per-process tables with a single global structure, trading lookup speed for memory efficiency.

📖 Traditional Page Table

> **Definition:** A per-process page table that contains an entry for every virtual page of a process.

- **Virtual address space:** 2^{34} bytes
- **Page size:** 8 KB = 2^{13} bytes
- **Number of virtual pages (entries):** $2^{34-13} = 2^{21}$
- **Page-table entry size:** 32 bits = 4 bytes

> **Size per process:** $2^{21} \times 4\ \text{B} = 2^{23}\ \text{B} = 8\ \text{MB}$

When only one process runs, the total page-table memory required is **8 MB**.

Space impact with multiple processes

Number of processes	Total page-table memory
1	8 MB
100	800 MB
1000	8000 MB (≈ 8 GB)

🔄 Inverted (Global) Page Table

> **Definition:** A single page table shared by all processes; each entry corresponds to a physical frame rather than a virtual page.

- **Physical address space:** 2^{29} bytes
- **Page size:** 2^{13} bytes → **Number of frames:** $2^{29-13} = 2^{16}$

> **Number of entries:** 2^{16} (one per frame)
> **Entry size:** 4 bytes

> **Total size:** $2^{16} \times 4\ \text{B} = 2^{18}\ \text{B} = 256\ \text{KB}$

The inverted table remains **256 KB** regardless of how many processes are present.

Space impact with multiple processes

Number of processes	Total page-table memory
1	256 KB
100	256 KB
1000	256 KB

🔑 Key-Based Lookup vs. Direct Indexing

> **Traditional lookup:** Uses the **page number** as an index; the physical frame number is obtained directly from the indexed entry (constant-time access).

> **Inverted lookup:** Forms a **key** = (Process ID, Page Number). The system searches the frame table sequentially for a matching key; the **index** of the matching entry yields the physical frame number.

- **Search time (traditional):** $O(1)$ – direct jump to the index.
- **Search time (inverted):** $O(N)$ – may need to scan up to N entries (where N = number of frames).

📊 Comparative Summary

Aspect	Traditional Page Table	Inverted (Global) Page Table
Entries	One per virtual page	One per physical frame
Entry count	2^{21}	2^{16}
Memory per process	8 MB	0 B (shared)
Total memory (100 proc.)	800 MB	256 KB
Lookup time	Constant $O(1)$	Linear $O(N)$
Key components	Page number only	(Process ID, Page Number)
Use case advantage	Fast address translation	Massive space saving across many processes

🧩 How the Frame Table Works

> **Definition:** The frame table (global page table) stores, for each physical frame, the **process ID** and **virtual page number** that currently occupies that frame.

- **Index** → physical **frame number**.
- **Entry** → (Process ID, Virtual Page Number).

When a CPU generates a logical address, it provides:

1. **Process ID**
2. **Virtual page number**

These two values form the search key. The system scans the frame table until it finds a matching entry; the entry's **index** is the desired frame number, which combined with the offset yields the final **physical address**.
