GeeksforGeeks

GATE फर्स्ट

CSE

OPERATING SYSTEM

SHORT NOTES

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

OPERATING SYSTEM

GATE फर्रे

- Introduction and Background of OS
- Threads & Multithreading
- Process Concepts
- CPU Scheduling
- Synchronization
- Deadlock
- Memory Management
- Virtual Memory
- File Systems
- Disk Scheduling

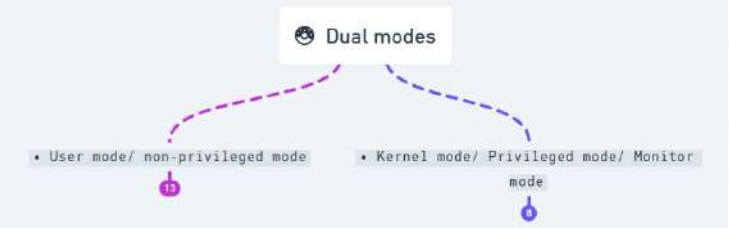
Introduction and Background of OS

1. What is OS ?

OS is a software that acts as a bridge between user and hardware, managing resources efficiently. It provides tools for application development and controls system operations like a government.

Goals of OS	Functions of OS
Easy-to-use environment for users	Manages processes (creation, scheduling, etc.)
Efficient resource usage (Resource Allocator)	Handles memory allocation & deallocation
Modularity (easy to maintain system structure)	Allocates resources like CPU, I/O, memory
Abstraction (hides hardware complexity)	Manages file system (read/write/access control)
Simplifies debugging for developers	Ensures protection & security of data & system

Dual Mode Operations



Purpose:

- Prevent user programs from directly accessing hardware
- Protect OS & system resources from unauthorized access

Mode Bit:

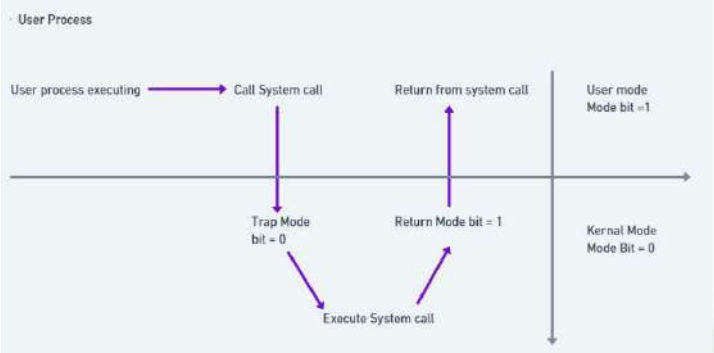
- A special bit used by the CPU to track current mode
- Mode Bit = 0 → **Kernel Mode**
- Mode Bit = 1 → **User Mode**

Kernel Mode:

- OS runs in this mode
- Has **full access** to hardware & system instructions
- Can execute **privileged operations** like:
  I/O control
  Memory management
  Context switching
  Interrupt handling
- Kernal mode is non preemptive (Atomic)

User Mode:

- User applications run here
- Cannot perform direct hardware operations
- If such actions are needed, control is transferred to **kernel** via **system calls**
- User mode is preemptive (non Atomic)



Note:

As per Von Neumann architecture, all secondary storage devices are part of I/P | O/P devices.

Threads & Multithreading

A **thread** is a smaller, lightweight version of a process

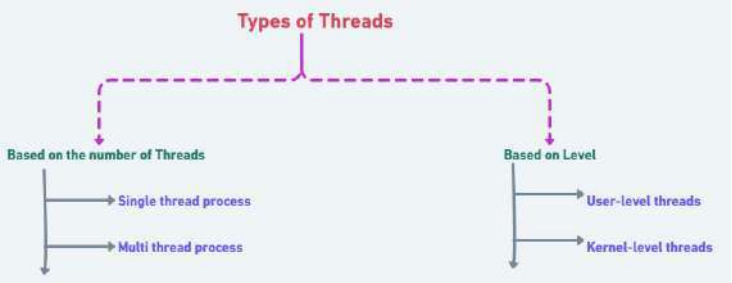
- In **multithreading**, all threads of the same process share:
  Memory (address space)
  Files
  Signals and their handlers

- But every thread has its own:
  Stack (used for function calls)
  Unique thread ID
  CPU information (like registers and stack pointer)
- Scheduling details (like thread state and priority)
  Like a process , thread is also a unit of cpu utilization .

  It is a Schedulable Unit .
  It is an Active Entity.

Benefits of Threads

- **Faster response** to users
- **Quick context switching** between threads
- **Better use of multiple processors** (parallel work)
- **Higher system throughput** (more work done in less time)
- **Cost-effective** (less overhead than processes)
- **Efficient sharing of resources** like memory and files



User-Level Threads	Kernel-Level Threads
These threads are managed at user level.	These threads are managed at kernel level.
These threads are not recognized by the kernel.	These threads are recognized by the kernel.
They are implemented as dependent threads.	They are implemented as independent threads.
All user-level threads of a process can run on one processor only, and one at a time.	Kernel-level threads can run on different processors at the same time (multiprocessing).
Blocking one user-level thread blocks the entire process.	Blocking one kernel-level thread doesn't affect other threads of the process.
These threads have less context.	These threads have more context.
Scheduling is done by the thread libraries.	Scheduling is done by the OS.
No hardware support required.	Hardware support is required.
Implementation is easy and simple.	Implementation is complicated and difficult.

Multithreading Models

1. **Many-to-One Model**
   - Many user-level threads (ULTs) are mapped to a **single kernel thread (KLT)**.
   - Drawback: If one thread makes a blocking system call → entire process is blocked.
   - Not suitable for multiprocessors (only one thread can access kernel at a time).
2. **One-to-One Model**
   - Each ULT maps to **one KLT**.
   - Allows better concurrency (multiple threads run in parallel on multiprocessors).
   - Drawback: Creating many threads incurs overhead → limited by OS.
3. **Many-to-Many Model**
   - Many ULTs are mapped to **many KLTs**.
   - OS creates required number of kernel threads dynamically.
   - Advantage: Flexibility + concurrency + avoids blocking problem.

Thread Libraries

- Provide **API to create and manage threads**.
- **Implementation Types:**
  - **User-level library** → Managed in user space (fast, but blocking issues).
  - **Kernel-level library** → Supported by OS (slower, but better concurrency).
- **Examples:**
  - **Pthreads (POSIX threads)** – C/C++ standard
  - **Java Threads** – built into JVM
  - **Green Threads** – User-level threads, scheduled by a runtime library

Disadvantages of Multithreading

1. **Blocking**
   - If one thread blocks, entire process may block (especially in many-to-one model).
   - CPU may stay idle.
2. **Security Issues**
   - Threads share same memory → higher risk of data corruption or unauthorized access.
3. **Overhead**
   - Maintaining **Thread Control Block (TCB)** for each thread increases overhead.

PROCESS CONCEPTS

Program Versus Process

Program

- A program is just a set of instructions written in some programming language.
- It is a passive entity stored on disk (file, executable).
- Example: notepad.exe, chrome.exe, or a C code file.
- It doesn't do anything until it is executed.

Process

- A process is a program in execution.
- It is an active entity with:
    - Program counter (which instruction to execute next)
    - Registers, Stack, Heap, and allocated memory
    - State (ready, running, waiting)
- **Example**: If you open Chrome 3 times, there are 3 processes running but only 1 program (chrome.exe).

Process as an Abstract Data Type (ADT)

A process can be thought of as an Abstract Data Type (ADT) because it has data (attributes) and operations defined on it.

Process (what makes a process in memory):

- Process attributes (PCB – Process Control Block)
- Run-time Stack (function calls, return addresses, local variables)
- Dynamic Data (heap, variables created at runtime)
- Static Data (global variables)
- Code Section (instructions to execute)

Process Attributes (stored in PCB):

1. Process Identification Information (PID, parent process ID)
2. Priority (for scheduling)
3. Process State Information (new, ready, running, waiting, terminated)
4. Program Counter (next instruction address)
5. Memory Limits (address space assigned to process)
6. List of Files (files used by process)
7. List of Open Devices (I/O devices allocated)
8. Protection Information (access rights, permissions)

Process Control Block (PCB)

- Each process has its own PCB.
- All PCBs are stored in main memory.
- Implemented using a doubly linked list.
- PCB acts as the "identity card" of a process for the Operating System.

Context of a Process

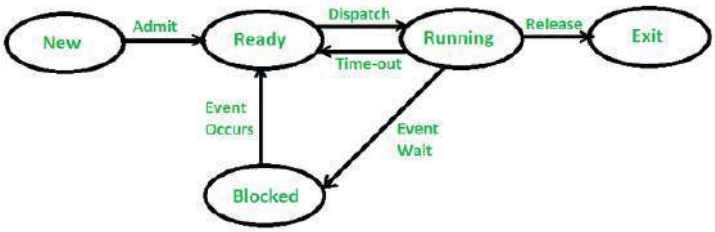
The context means the current status/info of a process.

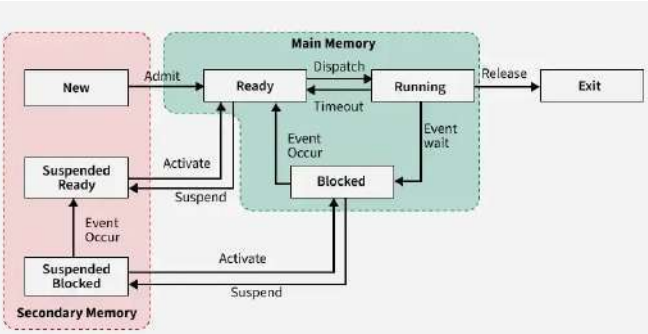
It includes:

- Process attributes (from PCB)

- Stack information (function calls, return values, etc.)

Process State Transition Diagram





Process States and Memory Location

1. Ready, Running, or Waiting (Blocked) State
    - The process is in main memory (RAM).
    - This allows the CPU or I/O devices to access it quickly.
2. Suspend Ready or Suspend Blocked (Wait) State
    - The process is moved to secondary memory (disk).
    - This happens when there isn't enough space in main memory or the OS decides to temporarily pause the process.

Number of Processes in Each State

- Ready State → Many processes can wait in main memory for CPU.
- Waiting (Blocked) State → Many processes can wait for I/O or events.
- Suspend Ready & Suspend Wait → Many processes can be kept in secondary storage.
- Running State → Only ONE process runs on the CPU at a time (per core).

Schedulers in Operating System

The OS uses different types of schedulers to manage processes efficiently.

1. Long-Term Scheduler (Job Scheduler)
    - Main Role → Decides which new processes should be admitted into the system.
    - Where it Works → From secondary storage (job pool) to main memory.
    - Controls → The degree of multiprogramming (how many processes are in memory at once).
    - Goal → Maintain a good balance between:
        - CPU-bound processes (need more CPU time)
        - I/O-bound processes (spend more time waiting for I/O)
    - State Transition → Responsible for New → Ready state.

2. Medium-Term Scheduler (Swapper)

- Main Role → Performs swap-out (move process from main memory to secondary memory) and swap-in (bring process back to main memory).
- Why Needed → To reduce the degree of multiprogramming (when too many processes are loaded).
- State Transitions →
    - Ready ⇌ Suspend Ready
    - Block ⇌ Suspend Block

In short: It suspends and resumes processes to balance load and improve performance.

3. Short-Term Scheduler (CPU Scheduler)

- Main Role → Chooses which process from Ready Queue will get the CPU next.
- Execution → Runs very frequently (milliseconds).
- State Transition → Ready → Run

In short: It is responsible for CPU allocation and ensures fair and fast execution.

Dispatcher in OS

- The dispatcher is the component that actually gives CPU control to the process chosen by the short-term scheduler.
- Its main job is to perform a context switch.

Context Switching means:

1. Saving the current process state (its registers, program counter, etc.) into its PCB (Process Control Block).
2. Loading the new process state from its PCB into the CPU.
3. Starting execution of the new process.

In short:

- Scheduler → decides *which process* should run.
- Dispatcher → actually *switches CPU* to that process.

CPU scheduling

Cpu scheduling is needed when the CPU must decide which process should run next.
It occurs in these cases:

1. **New process enters Ready state**
   - A new job comes into the Ready queue → scheduler must decide if it should get CPU.
2. **Wait → Ready transition**
   - A process waiting for I/O finishes its I/O → moves to Ready → scheduler decides.
3. **Run → Wait transition**
   - A running process requests I/O → CPU becomes free → another process must be scheduled.
4. **Run → Ready transition (time slice over)**
   - In **Round Robin**, after every *q seconds* (time quantum), the running process is stopped and moved back to Ready → scheduler picks another.
5. **Priority scheduling**
   - If a Ready process has **higher priority** than the one currently running, the CPU is given to the higher-priority process (**preemption**).

Goals of CPU Scheduling

Maximize CPU utilization.
Minimize the response time and waiting time of the processes.

1. Arrival Time (AT)

- The time when a process **enters the Ready Queue**.
  ☞ Example: If P1 comes at time 0, its AT = 0.

2. Burst Time (BT)

- The **total CPU time required** by a process for its execution.
  ☞ Example: If P1 needs 5 units, BT = 5.

3. Completion Time (CT)

- The time when a process **finishes execution**.
  ☞ Example: If P1 finishes at time 7, CT = 7.

4. Turnaround Time (TAT)

- The **total time taken** from arrival to completion.
- Formula: TAT= CT−AT

Example: If CT = 7 and AT = 0 → TAT = 7.

5. Waiting Time (WT)

- The **total time spent in Ready Queue**, waiting for CPU.
- Formula: WT=TAT−BT

Example: If TAT = 7, BT = 5 → WT = 2.

6. Response Time (RT)

- The time from **arrival to first CPU execution**.
- Formula: RT=(FirsttimeCPUgiven)−AT

Example: If process arrived at 0 but got CPU at 3 → RT = 3.

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

Scheduling Algorithms

CPU Scheduling Algorithms

1. First Come First Serve (FCFS)

- Type: Non-preemptive
- Rule: Process assigned to CPU in order of arrival time.
- Tie-breaker: If same arrival time → lower process ID first.
- No starvation (every process eventually executes).
- Convoy Effect → a long process delays all the short ones.

2. Shortest Job First (SJF)

- Type: Non-preemptive
- Rule: Process with shortest burst time first.
- Tie-breaker: If burst time equal → arrival time decides order.
- If all burst times equal → behaves like FCFS.
- Minimizes average response time.
- Can cause starvation (long jobs may wait if short ones keep coming).
- Needs knowledge of burst times in advance.

3. Shortest Remaining Time First (SRTF)

- Type: Preemptive
- Rule: At any moment, CPU is given to the process with the shortest remaining burst time.
- If all arrival times are same → behaves like SJF.
- Minimizes average turnaround time.
- Starvation possible (if many short processes keep arriving, long ones wait forever).

4. Longest Remaining Time First (LRTF)

- Type: Preemptive
- Rule: At any moment, CPU is given to the process with the longest remaining burst time.
- Minimizes average response time.
- Free from starvation (long processes always get preference).
- Favors CPU-bound processes → not fair to short ones.

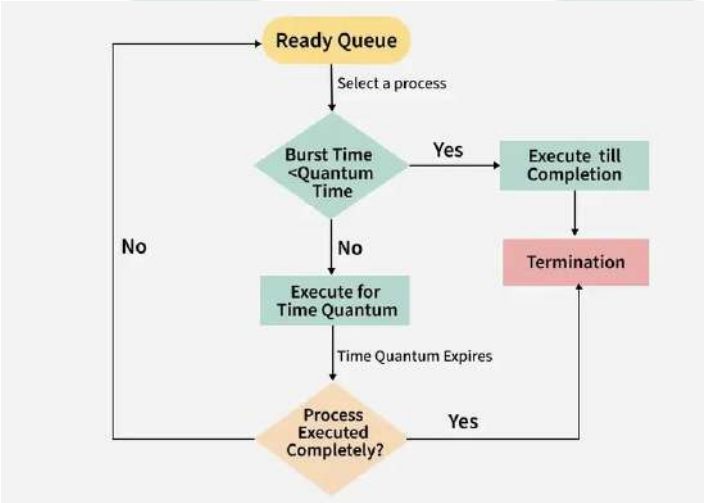
Priority Scheduling

- Type: Can be Preemptive or Non-preemptive
- Rule: CPU is assigned to the process with the highest priority.
- Preemptive Case: If a higher-priority process arrives, it preempts the current one.
- Non-preemptive Case: Once a process starts, it completes; scheduler always picks the highest priority next.
- If all processes have the same priority → behaves like FCFS.
- Advantage: Flexible (supports both preemptive & non-preemptive).
- Disadvantage: Starvation possible (low-priority processes may never execute).

- Solution: Use Aging technique (gradually increase priority of waiting processes).

6. Round Robin (RR) Scheduling

- Type: Preemptive (FCFS + Time Quantum)
- Rule: Each process gets CPU for a time slice (quantum q) in cyclic order.
- **Effect of q (time quantum):**
  - If q is too small → too many context switches (high overhead) but good response time.
  - If q is too large → behaves like FCFS (bad response time, but low overhead).
- Advantage: Fair (every process gets CPU time), good for time-sharing systems.
- Disadvantage: Average turnaround time can be large if q not chosen properly.



7. Highest Response Ratio Scheduling (HRRN / HRSN)

Type: Non-preemptive
Rule: Process with the highest response ratio is selected.
:
HRR=WT+BT

WT = Waiting Time

- **BT = Burst Time**

Advantages:

- Free from starvation (since waiting time increases HRR, even long jobs eventually get CPU).
- Balances between short jobs (high HRR early) and long jobs (HRR grows with wait).

Disadvantages:

- Requires recalculation of HRR each time CPU becomes free (overhead).

8. Multilevel Queue Scheduling (MLQ)

Type: Can be Preemptive or Non-preemptive (depends on policy).
Rule:

- Ready queue is split into multiple queues, each with its own scheduling algorithm.
- Each queue has a fixed priority → higher priority queues are served first.

Example:

- System processes → Highest priority (FCFS).
- Interactive processes → Round Robin.
- Batch jobs → FCFS / SJF.

Advantages:

- Separates processes by category (system vs user) → good workload management.
- Easy to implement.

Disadvantages:

- Rigid: once a process is assigned to a queue, it cannot move to another.
- May cause starvation of lower-priority queues.

7. Highest Response Ratio Scheduling (HRRN / HRSN)

Type: Non-preemptive
Rule: Process with the highest response ratio is selected.
Formula:

HRR=WT+BT
WT = Waiting Time

- BT = Burst Time

Advantages:

- Free from starvation (since waiting time increases HRR, even long jobs eventually get CPU).
- Balances between short jobs (high HRR early) and long jobs (HRR grows with wait).

Disadvantages:

- Requires recalculation of HRR each time CPU becomes free (overhead).

8. Multilevel Queue Scheduling (MLQ)

Type: Can be Preemptive or Non-preemptive (depends on policy).
Rule:

- Ready queue is split into multiple queues, each with its own scheduling algorithm.
- Each queue has a fixed priority → higher priority queues are served first.

Example:

- System processes → Highest priority (FCFS).
- Interactive processes → Round Robin.
- Batch jobs → FCFS / SJF.

Advantages:

- Separates processes by category (system vs user) → good workload management.
- Easy to implement.

Disadvantages:

- Rigid: once a process is assigned to a queue, it cannot move to another.
- May cause starvation of lower-priority queues.

Solution → Aging:

- A technique where the priority of a process increases automatically the longer it waits.
- Prevents starvation by gradually moving low-priority processes into higher-priority queues.

Process Synchronization

Inter-Process Communication (IPC)

- **Definition**: IPC is a mechanism that allows processes to communicate and synchronize their actions.
- **Need for Synchronization**: Every communication must be coordinated.
- **Lack of Synchronization leads to**:
    1. Inconsistency (incorrect results)
    2. Data loss
    3. Deadlock

2. Synchronization

- **Definition**: An agreed protocol in IPC to ensure correct execution without inconsistency, data loss, or deadlock.
- **Concept**: Involves orderly sharing of system resources among processes.
- Shared resource → critical section accessed by multiple processes.

3. IPC Environment

- Processes interact through **shared resources**.
- Synchronization ensures **safe and consistent access**.

Types of Processes

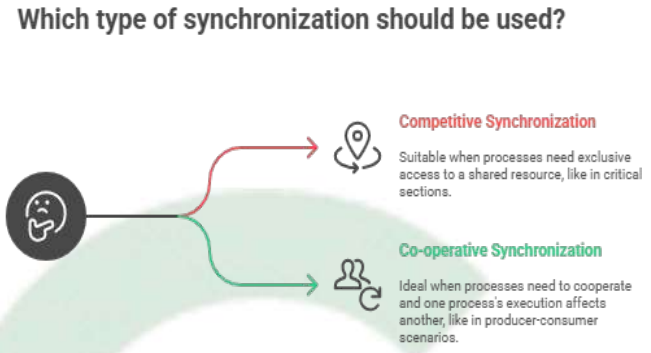
Processes are classified as:

- **Cooperative Process** – Execution of one process affects or is affected by another process.
- **Independent Process** – Execution of one process does not influence or depend on any other process.

Only cooperative processes require synchronization.

Understanding Synchronization

Synchronization is required to avoid problems such as race conditions, data inconsistency, deadlocks, and data loss.



To achieve synchronization, certain conditions must be satisfied: **mutual exclusion, progress, and bounded waiting**.
Effective solutions include semaphores, monitors, mutex locks, and other synchronization mechanisms.

Problems Due to Lack of Synchronization

Lack of synchronization leads to incorrect results, corrupted or lost data, deadlocks, or indefinite waiting (starvation).
Example: The *Producer–Consumer Problem* demonstrates the need for proper synchronization.

Producer–Consumer Problem (Bounded Buffer)

Concept

- **The producer** generates items and places them into a finite buffer.
- **The consumer** removes items from the buffer and uses them.
- **IN**: Index used by the producer to place the next item.
- **OUT**: Index used by the consumer to remove the next item.
- **COUNT**: Tracks number of items currently in the buffer.
  Problem: Must ensure producer does not add to a full buffer, and consumer does not remove from an empty buffer.

Without synchronization:

```
#define N 8              // Buffer size
int count = 0;           // Initially buffer is empty
int in = 0, out = 0;     // Indices for producer and consumer
int buffer[N];           // Shared buffer

void producer(void) {
    int itemp;
    while (true) {
        itemp = produce_item();          // Create new item
        while (count == N);              // Wait if buffer is full (busy waiting)
        buffer[in] = itemp;              // Insert item
        in = (in + 1) % N;               // Update index circularly
        count = count + 1;               // Increase item count
    }
}

void consumer(void) {
    int itemc;
    while (true) {
        while (count == 0);              // Wait if buffer is empty (busy waiting)
        itemc = buffer[out];             // Remove item
        out = (out + 1) % N;             // Update index circularly
        count = count - 1;               // Decrease item count
        process_item(itemc);             // Consume item
    }
}
```

Drawbacks of This Solution

- Uses **busy waiting** → wastes CPU cycles.
- No **mutual exclusion** → race conditions possible on shared variables (count, in, out).
- Not efficient for multiprocessor systems.

Correct Solution (Using Synchronization Tools)

Replace busy waiting with **semaphores** or **mutex + condition variables**.
Ensure:

- Producer waits if buffer is full.
- Consumer waits if buffer is empty.
- Mutual exclusion while accessing buffer.

Producer–Consumer Problem (Bounded Buffer)

- **COUNT**: Shared variable used by producer and consumer to keep track of items in the buffer.
- **IN**: Index where the producer places the next item.
- **OUT**: Index where the consumer removes the next item.
- **Buffer size = N** (finite).

Conditions to be satisfied

- If buffer is **full**, producer must wait.
- If buffer is **empty**, consumer must wait.

Uncontrolled Execution (Problem)

If producer and consumer both access COUNT simultaneously without synchronization, **race conditions** occur.
Example (Producer executes):
T0: Load R1 ← COUNT
T1: Increment R1
T2: Store COUNT ← R1

Example (Consumer executes at same time):
T0: Load R2 ← COUNT
T1: Decrement R2
T2: Store COUNT ← R2

⚠ Final value of COUNT may become inconsistent, since both processes update it concurrently.
This leads to **incorrect buffer status**, possible data loss or overwrite.

Daemon Example – Printer Spooler

- Multiple processes may send jobs to the printer at the same time.
- Without synchronization, one process might overwrite another's request in the **spooler directory**, leading to lost print jobs.
- This illustrates the need for synchronization in shared resource management.

Critical Section Problem

- A **critical section** is a part of the program where a shared resource (variable, file, or hardware) is accessed.
- Only **one process at a time** should execute in its critical section.
- Every process has the following structure:

```
do {
  Entry Section   // Request to enter critical section
  Critical Section
  Exit Section    // Signal that critical section is over
  Remainder Section
} while(true);
```

Race Condition

- A race condition occurs when multiple processes access and update shared data concurrently, and the final outcome depends on the order of execution.
- Example: Producer–Consumer using shared COUNT.

Requirements for a Correct Synchronization Solution

1. **Mutual Exclusion**
   - Only one process can be in the critical section at a time.
2. **Progress**
   - If no process is in its critical section, the selection of the next process to enter must not be postponed indefinitely.
3. **Bounded Waiting**
   - There must be a limit on how many times other processes can enter before a waiting process gets its turn
4. No assumptions related to hardware and the processor speed: Number of processes.

I. Software-based Solutions

Use only software logic, no hardware support.

Examples:

1. **Lock Variables**
   - Use a shared boolean variable lock.
   - If lock = 0 → section is free.
   - If lock = 1 → section is busy.

- Problem → Not atomic, leads to busy waiting + race conditions.

2. Strict Alternation (Dekker's Algorithm)

- Processes take turns in strict sequence.
- Problem → Causes unnecessary blocking (even if section is free).

3. Peterson's Algorithm

- **Uses two variables:**
  - **flag[i]** → interest of process **i.**
  - **turn** → whose turn it is.
- Ensures mutual exclusion + progress + bounded waiting.
- Correct software solution, but may fail on modern CPUs (out-of-order execution).

II. Hardware-based Solutions

Leverage atomic machine instructions.

1. TSL (Test-and-Set Lock) Instruction

- Atomic instruction: reads and sets a lock variable in one step.
- Prevents race conditions.
- Problem → Busy waiting.

2. Test-and-Set Lock (similar idea)

- Continuously checks (test) and then sets lock atomically.
- Provides mutual exclusion.
- Still suffers from busy waiting.

III. Operating System-based Solutions

Implemented inside OS for process synchronization.

1. Counting Semaphore

- Integer value, can be >1.
- Controls access to multiple instances of a resource.
- Uses P (wait) and V (signal) operations.

2. Binary Semaphore (Mutex)

- Value = 0 or 1.
- Equivalent to a lock.
- Ensures only one process enters critical section at a time.

IV. Programming Language / Compiler Support

Synchronization provided at language level.

1. Monitors

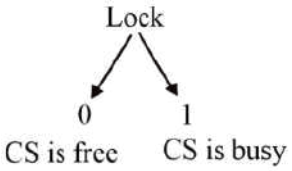
- High-level abstraction for process synchronization.
- Has shared variables + procedures + condition variables.
- Compiler ensures mutual exclusion automatically.
- Examples: Java synchronized, C# lock.

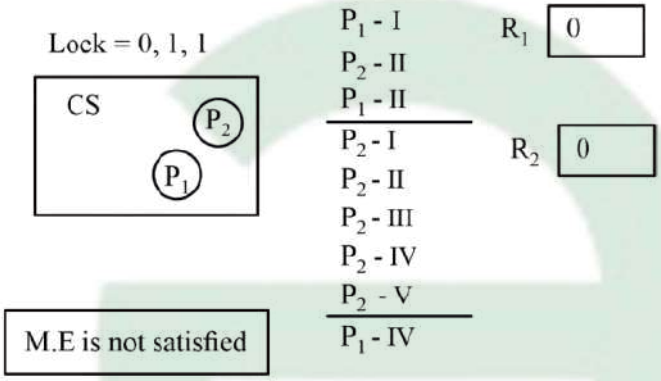
I. Software Types:

(a) Lock Variables:

Entry Section:

I. Load Ri, m[Lock] (Ri Respective Process Register)
II. Cmp Ri, #0
III. JNZ to Step (I)
IV. Store m[Lock], #1 V. C.S VI. Store m[Lock], #0





We have proved that both the processes P1 and P2 are entering into the critical section at the same time, Hence mutual exclusion is not satisfied and the solution is bound to be incorrect.

(b) Strict Alteration or Decker's Algorithm:

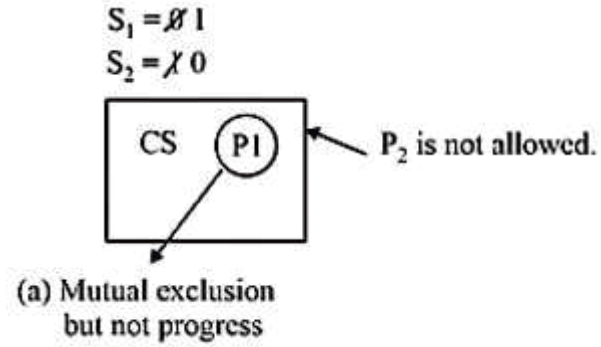
(Process takes 'Turn' to enter into C.S)

Process 'P_0' code	Process 'P_1' code

while (true)
{
non_cs ();
while (turn! = 0);
c.s
turn = 1;
}
	while (true)
	{
	non_cs();
	while(turn! = 1);
	c.s
	turn = 0;
	}

Important Points:

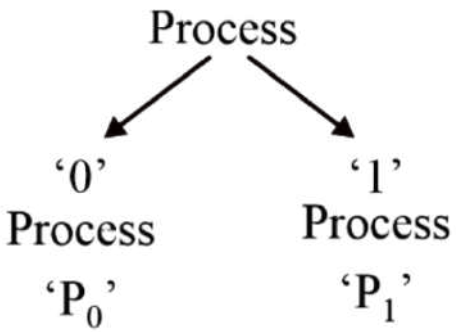
The pre-emption is just a temporary stop and the process will come back and continue the remaining execution. If there is any possibility of solution becoming wrong by taking the pre-emption then consider the pre-emption. If any solution is having deadlock the progress is not satisfied.



(c) Peterson's Algorithm:

(Two Process Solution)

```
#define N 2
 #define TRUE 1
#define FALSE 0
 int turn; int interested [N];
void enter_Region (int process)
 {
1. int other
2. other = 1 – process;
 3. interested [process] = TRUE;
 4. turn = process;
5. while (turn = = process && interested [other] = = TRUE);
}
C.S
void leave_Region(int process)
{
 interested [process] = FALSE;
 }
 initially interested [0] = FALSE;
interested [1] = FALSE
```



Hardware Type

(a) TSL Instruction Set:
(Test and Set Lock):

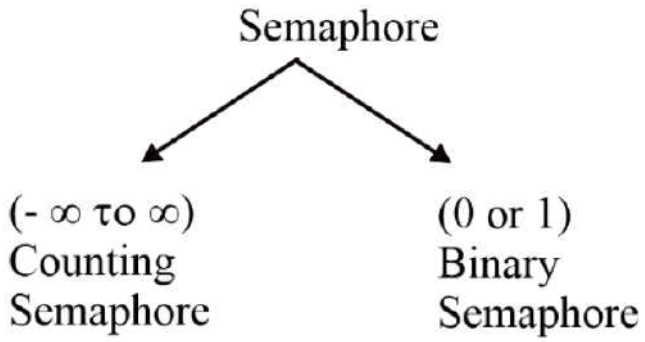
TSL Register Flag:
Copies the current value of flag into register and stores the value of '1' into flag in a single atomic cycle without any pre-emption.

Entry Selection:
1. TSL Ri, m[flag]
2. Cmp Ri, 0
3. JMP to step (1)
4. c.s
5. Store m[flag], 0

Algorithm	M.E.	Progress	Bounded Waiting
1. Lock Variable	×	✓	×
2. Strictalteration or Decker's Algorithm	✓	×	✓
3. Peterson's Algorithm	✓	✓	✓
4. TSL Instruction	✓	✓	×

OS Type

Semaphore is of two types:



1. Counting Semaphore

- Can take **non-negative integer values**.
- Used to control access to a resource that has **multiple instances**.
- Example: If a printer pool has 5 printers, the counting semaphore is initialized to 5. Each process requesting a printer decreases the value, and releasing increases it.

2. Binary Semaphore (Mutex)

- Can take only **0 or 1** as its value.
- Used to implement **mutual exclusion (M.E.)**, ensuring that only one process accesses the critical section.

(a) Counting Semaphore:

```
Down (Semaphore s)
 {
s.value = s.value – 1;
 if (s.value < 0)
{
Block the process and place its PCB
suspended list ();
}
        }
Up (Semaphore s)
        {
        s.value = s.value + 1; if (s.value  0)
        {
                Select a process from suspended list
        and wakeup ();
```

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

```
        }
}
```

Binary Semaphore:

```
Down (Semaphore S)
{
    if (S.value == 1)
       S.value = 0;
    else
    {
        block the process
        and place its PCB
        in the suspended list ()
    }
}
```

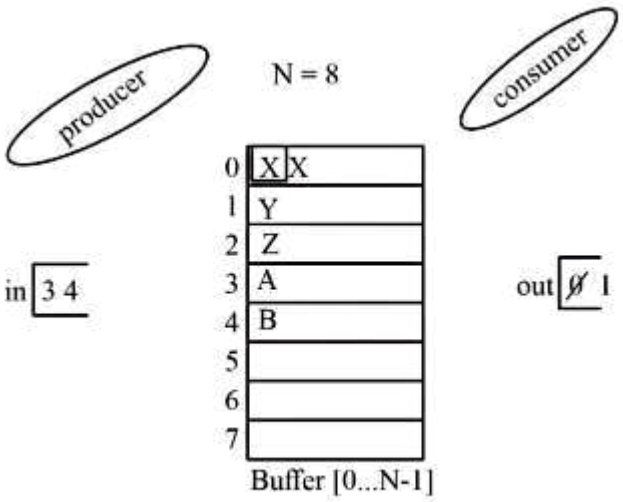
```
Up (Semaphore S)
{
    if (Suspended list is empty)
    S.value = 1;
    else
    {
        select a process
        from suspended list
        and wakeup ();
    }
}
```

Notes on Semaphores (Counting & Binary)

- Each semaphore has its **own suspended (waiting) list**.
- **Down (P) and Up (V) operations are atomic** → OS ensures no interrupts.
- If multiple processes are waiting, **one process wakes up per Up operation** (chosen in **FIFO order** → ensures bounded waiting).
- If processes remain in the suspended list with no chance to wake up, they may enter **deadlock**.

Classical Problems of IPC

Producer consumer with semaphore:




```
semaphore mutex = 1;
semaphore empty = N;
 semaphore full = 0;
void producer(void)
{
int item p;
while(true)

{
 produce_item (item p);
down(empty);
down(mutex);
buffer [in] = item p;
 in = (in + 1) mod N up(mutex); up(full);
 }
 }
void consumer (void)
 {
 int item c;
while (true)

{
 down (full);
down (mutex);
item c = buffer [out];
out = (out + 1)mod N;
up (mutex);
 up (empty);
process_item (item c);
}
}
```

- **mutex** → Binary semaphore.
    - Ensures **mutual exclusion** when producer/consumer accesses buffer.
- **empty** → Counting semaphore.
    - Shows **number of empty slots** available in buffer.
- **full** → Counting semaphore.
    - Shows **number of filled slots** in buffer.

READERS WRITERS PROBLEM

```
int rc = 0;
 semaphore mutex = 1;
semaphore db = 1;
 void reader (void)
 {
while (true)
 {
 down (mutex);
 rc = rc + 1;
 if (rc = = 1)
 down (db);
 up (mutex);
 }

}
void writer (void)
 {
while (true)
{
down (db);
D.B
up (db);
 }
 }
```

conditions to be followed
1) R - W
2) R – R
3) W – R
4) W – W

Semaphores in Readers–Writers Problem

- **rc (Readers Count)** → Integer variable.
    - Tracks **number of readers** currently accessing the database.
- **mutex** → Binary semaphore.
    - Ensures **mutual exclusion** while updating rc (readers count).
- **db** → Binary semaphore.
    - Ensures **mutual exclusion** for database access (between readers & writers).

```
# define N                5                    /* number of philosophers */
# define LEFT             (i+N-1)%N            /* number of I's left neighbours */
# define RIGHT            (i+1)%N              /* number of I's right neighbours */
# define THINKING         0                    /* philosopher is thinking */
# define HUNGRY           1                    /* philosopher is trying to get forks */
# define EATING           2                    /* philosopher is eating */
typedef int semaphore                          /*semaphores are a special kind of int */
int state [N];                                 /* array to keep track of everyone's state */
semaphore mutex = 1;                           /* mutual exclusion for critical regions */
semaphore s[N];                                /* one semaphore per philosopher */

void philosopher (int i)                       /* i: philosopher number, from 0 to N -1 */
{       while (TRUE) {                         /* repeat forever */
                Think();                       /* philosopher is thinking */
                Take_forks(i);                 /* acquire two forks or block */
                Eat();                         /* yum-yum spaghetti */
                Put_forks(i);                  /* put both forks back on table */
        }
}
void take_forks(int i)                         /* i: philosopher number, from 0 to N -1 */
{
        down(&mutex);                          /* enter critical region */
        state [i] = HUNGRY;                    /* record fact that philosopher is hungry */
        test(i);                               /* try to acquire two forks */
        up(&mutex);                            /* exit critical region */
        down(&s[i]);                           /* block if forks were not acquired */
}
void put_forks(i)                              /* i: philosopher number, from 0 to N -1 */
{
        down(&mutex);                          /* enter critical region */
        state [i] = THINKING;                  /* philosopher has finished eating */
        test(LEFT);                            /* see if left neighbour can now eat */
        test(RIGHT);                           /* see if left neighbour can now eat */
        up(&mutex);                            /* exit critical region */
}
void test(i)                                   /* i: philosopher number, from 0 to N -1 */
{
      if (state[i] == HUNGRY && state [LEFT] = EATING && state [RIGHT] = EATING) {
              state [i] = EATING;
              up(&s[i]);
        }
}
```

Made picking up left and right chopsticks an atomic operation ξ or, N – 1 philosophers but N chopsticks ξ …both changes prevent deadlock.

Monitors (Synchronization)

- **Monitor** → Language-level construct for synchronization (compiler-supported).
- A monitor = **variables + procedures** combined in a special module.
- External processes **cannot access variables directly**, but can **call monitor procedures**.
- **Key property** → At any time, only **one process** can be active inside the monitor → ensures **mutual exclusion**.

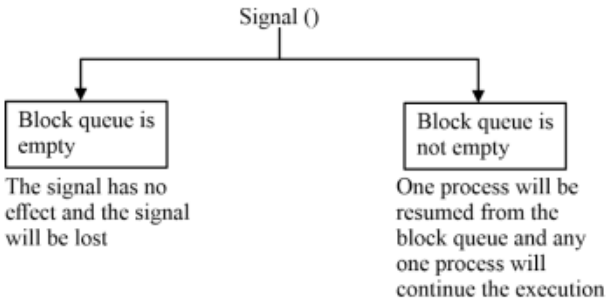
Syntax

```
Monitor example
 {
 Variables;
Condition variables;
Procedure P1
 {
 }
Procedure P2
        {
        }
}
```

Condition Variables in Monitors

- Declared as: **Condition x, y;**
- Used for synchronization inside monitors.
- Two main operations:

1. **Wait()**
   - Example: x.wait(); or wait(x);
   - Process is **suspended** and placed in the **block queue** of that condition variable.
2. **Signal()**
   - Example: x.signal(); or signal(x);
   - Wakes up **one process** waiting on that condition variable (if any).



Concurrent Programming

S1: a = b + c ;
S2: d = e * f ;
S3: g = a / d ;
S4: h = g * i ;
Read set = {b, c, e, f, a, d, g, i}
Write set = {a, d, g, h}

Precedence graph

S1, S2 can execute concurrently

Any two statements Si and Sj can be executed concurrently or parallel if they are following the conditions.

(1) R(Si) ∩ W (Sj) = φ
(2) W(Si) ∩ R (Sj) = φ
(3) W(Si) ∩ W (Sj) = φ

The real concurrent programming is possible only on the multiprocessor system.

Concurrent has 3 different meanings

o They can execute concurrently or parallel
o They don't have any dependency Anyone can start first [for single processor this will be applicable]

Deadlock – Concept

- **Definition:**
  A **deadlock** is a situation in which a set of processes are **blocked** because each process is holding a resource and waiting for another resource that is already held by some other process in the set.

- **Example:**
  - System has **2 disk drives**.
  - Process **P1** holds **one disk** and needs the **second disk**.
  - Process **P2** holds the **second disk** and needs the **first disk**.
  - Both are waiting for each other → deadlock occurs.

System Model

- **Resources:**
  1. Types of resources → R1, R2, ..., Rm
  2. Examples: CPU cycles, memory space, I/O devices
  3. Each resource type Ri has **Wi instances**

- **Resource Utilization by a Process:**
  1. **Request** → Process requests a resource
  2. **Use** → Process uses the resource
  3. **Release** → Process releases the resource

Deadlock Characteristics (Coffman's Conditions)

Deadlock can occur in a system **only if all the following 4 conditions hold simultaneously**:

1. Mutual Exclusion

- Only **one process at a time** can use a resource.
- If another process requests the same resource, it must wait until the resource is released.

Example:

- A printer can be used by only one process at a time.

2. Hold and Wait

- A process is **holding at least one resource** and **waiting to acquire additional resources** that are currently held by other processes.

Example:

- Process **P1** holds a printer and requests a disk.
- Process **P2** holds the disk and requests the printer.

3. No Preemption

- A resource **cannot be forcibly taken** from a process.
- It can be released **only voluntarily** by the process holding it, after completing its task.

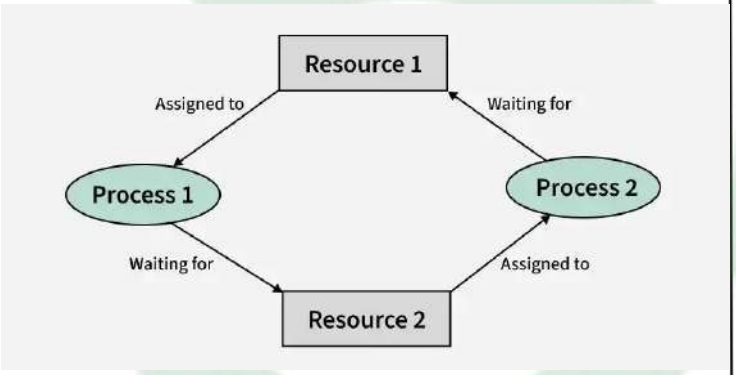
Example:

- CPU registers or I/O devices cannot be forcibly taken back.

4. Circular Wait

- A set of processes {P1, P2, ..., Pn} are **waiting for resources** in such a way that:
    - P1 → waiting for a resource held by P2
    - P2 → waiting for a resource held by P3
    - ...
    - Pn → waiting for a resource held by P1

This forms a **circular chain of waiting**, leading to deadlock.



Note: If all the deadlock characteristics simultaneously exist in the system, then the system is in deadlock.

Deadlock Prevention

☞ *Deadlock prevention ensures that the system never enters a deadlock state.*

☞ This is done by denying at least one of the four necessary conditions (Coffman's conditions).

1. Mutual Exclusion

- **Cannot be prevented because:**
    - Some resources are inherently non-sharable (e.g., printer, tape drive).
    - Only sharable resources (like read-only files) allow multiple processes simultaneously.

2. Hold and Wait

Deadlock can be prevented by ensuring processes do not hold resources while waiting for others.

Strategies:

1. All-at-once allocation → A process must request all resources at the beginning of execution.
    - Prevents hold and wait.
    - Leads to low resource utilization and possible starvation.
2. **Release-before-request** → A process must release all currently held resources before requesting new ones.
    - Prevents deadlock.
    - Can cause starvation and extra overhead due to repeated release/reacquire.

3. No Preemption

Allow resources to be forcibly taken away (preempted) if needed.

- If process P1 requests a resource held by P2:
  - If P2 is executing, then P1 must wait.
  - If P2 is waiting, then its resources can be preempted and given to P1.

Works well for CPU and memory.
Not suitable for non-preemptible resources (e.g., printer).

4. Circular Wait

Prevented by ordering resources and forcing processes to request them in a fixed order.

Algorithm:

1. Assign a unique number to each resource type.
2. A process must request resources in increasing (or decreasing) order of numbering.

Breaks circular wait.

Not always flexible for real systems.

Memory

Introduction

- In a multiprogramming system, the task of dividing memory among various processes is called memory management.
- The responsibility of the Memory Management Unit (MMU) is to utilize memory efficiently and minimize both internal and external fragmentation.

Logical vs Physical Address

- The address generated by CPU is called the logical address.
- The address perceived by the memory unit is called physical address.

Memory Management Unit

- A hardware device that maps virtual addresses to physical addresses is called the Memory Management Unit (MMU).
- In the MMU scheme, the value stored in the relocation register is added to every address generated by a user process before it is sent to memory.
- The user program works with logical (virtual) addresses; it never directly accesses or sees the actual physical addresses.

Loading

It is defined as bringing the program from the secondary to the main memory.
It is classified into three types:

(i) Absolute Loading

A given program is always loaded into the same memory location whenever it is loaded for execution.

(ii) Relocatable Loading

- A given program can be loaded into any desired memory location each time it is loaded for execution
- The compiler must generate relative (logical) addresses for the program.

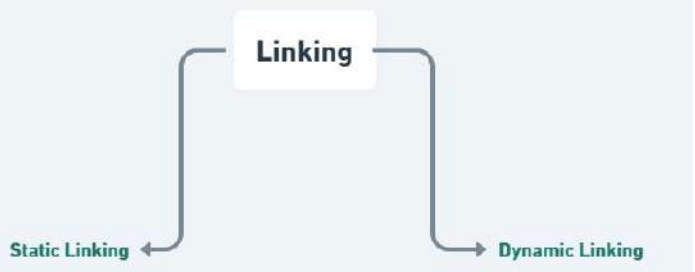
(iii) Dynamic Loading

- A routine is not loaded into memory until it is called, which allows for better memory-space utilization (unused routines are never loaded and their loading is postponed until execution time).
- It is useful when large amounts of code are required for handling infrequently occurring cases.
- No special support from the operating system is needed; it is implemented through the program's design.
- Address translation is handled by the hardware.

Linking

Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed. Linking can be performed at **compile time**, **load time**, or **run time**.

Linking is classified into two types:



1. Static Linking

Static linkers take as input a collection of **relocatable object files** along with **command-line arguments**, and generate a **fully linked executable file**. This executable file can then be loaded into memory and run directly.

2. Dynamic Linking

Dynamic linking involves **shared libraries**, which are object modules that can be **loaded at run time** at arbitrary memory addresses and linked with the program in memory. This allows for **efficient memory usage** and **code reuse** across multiple programs.

Address Binding

Address binding is the process of associating program instructions and data with actual physical memory locations.
Address binding can occur at three different stages:

1. Compile-Time Binding

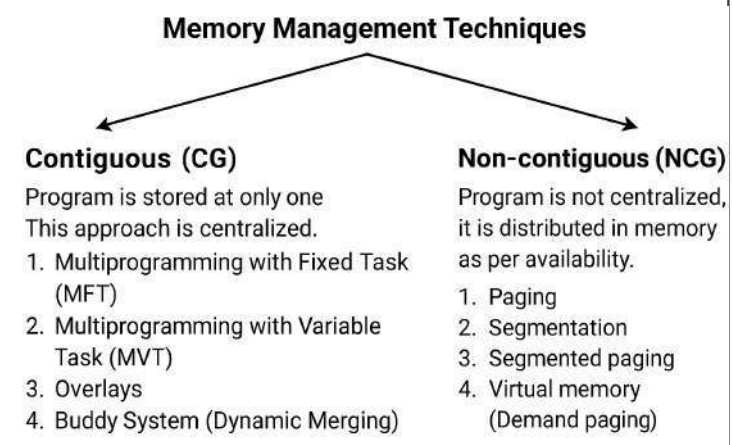
- If the memory location is known in advance, **absolute code** can be generated.
- However, if the starting memory location changes, the code must be **recompiled**.

2. Load-Time Binding

- If the memory location is **not known at compile time**, the compiler must generate **relocatable code**.
- The actual addresses are then determined when the program is **loaded into memory**.

3. Execution-Time Binding

- Address binding is delayed until **run time**, which allows a process to be **moved in memory** during its execution (e.g., for swapping or dynamic relocation).
- This requires **hardware support** for address mapping, typically using **base and limit registers**.



Contiguous Memory Allocation

Two schemes:

1. **Fixed Partitioning (Static)**
2. **Variable Partitioning (Dynamic)**

Fixed Partitioning (Static)

- Memory is divided into a fixed number of partitions (equal/unequal sizes).
- Each partition is associated with a limit register.
- Degree of multiprogramming is limited to the number of partitions.
- A process may not fit if partition size is too small.
- Problem: **Internal Fragmentation** occurs.

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

Variable Partitioning (Dynamic)

- Initially, memory is a single large continuous free block.
- When a process arrives, a hole of exact required size is allocated.
- No **Internal Fragmentation**.
- Problem: **External Fragmentation** occurs.
- Requires **Compaction** (overhead).

Dynamic Partition Allocation Methods

When more than one partition can accommodate a process, the following strategies are used:

- **First Fit**:
  Allocate the *first* free block (from the beginning of memory) that is large enough.
  - *Fast, but may lead to external fragmentation at the beginning.*
- **Next Fit**:
  Similar to First Fit, but scanning begins from the **last allocated position**, not from the start.
  - *Reduces search time compared to First Fit.*
- **Best Fit**:
  Scans the **entire memory** to find the **smallest free block** that can accommodate the process.
  - *Minimizes leftover space but increases search time; can lead to many small fragments.*
- **Worst Fit**:
  Scans the **entire memory** to find the **largest free block**.
  - *Leaves large leftover space, may help reduce external fragmentation.*

Non-Contiguous Memory Allocation

Paging

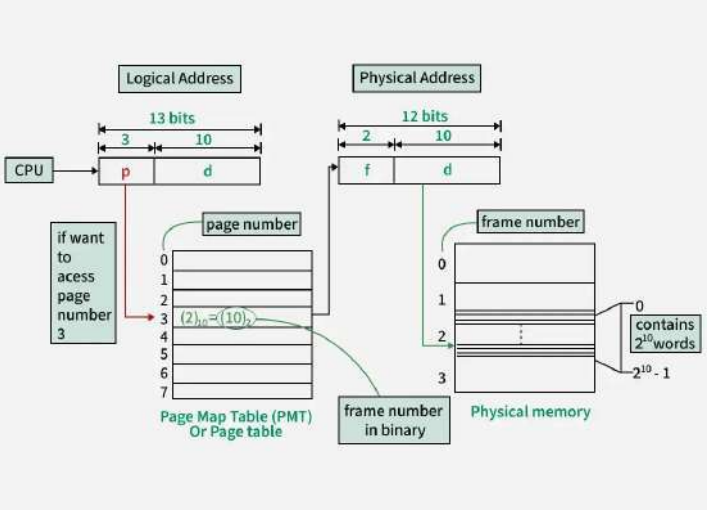
- **Definition:**
  The technique of mapping CPU-generated **logical addresses** to **physical addresses** using a **page table**.

Key Concepts

1. **Division of Memory:**
   - Logical address space → divided into **equal-size pages**.
   - Physical address space → divided into **equal-size frames**.
   - **Page size = Frame size**.
2. **Process Mapping:**
   - When a process is created, it is divided into pages.
   - A **Page Table** is maintained in main memory for each process.
   - Base address of the page table is stored in the **PCB (Process Control Block)**.

Page Table:

- Number of entries = number of pages in logical address space.
- Each entry stores the **frame number** where the corresponding page is placed.
- Page table is also called the **Address Translation Table**.
  Fragmentation in Paging
- **External Fragmentation:** *Does not occur.*
- **Internal Fragmentation:** *May occur in the last page only.*
- Maximum Internal Fragmentation = **[p / 2]**, where **p = page size**.



Paging with TLB (Translation Lookaside Buffer)

- **Translation Lookaside Buffer (TLB):**
  - A **hardware cache** implemented using **associative registers**.
  - Stores **recently/frequently used (Page No. → Frame No.)** mappings.
  - Access time of TLB (**c**) ≪ main memory access time.
  - Purpose: **Reduces effective memory access time (EAT)**.

Working

1. CPU generates logical address → TLB is checked.
2. **TLB Hit:** Frame number is found in TLB → Only **1 memory access** required.
3. **TLB Miss:** Page table in main memory is accessed → **2 memory accesses** required (one for page table, one for data).

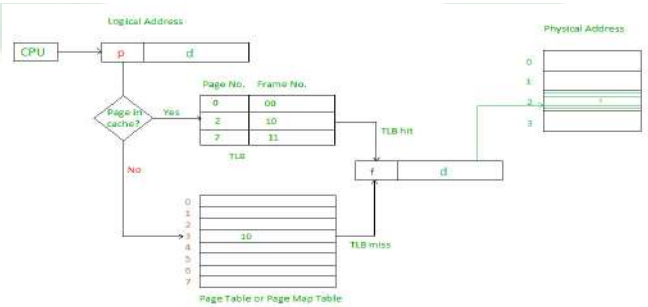
Formula for Effective Memory Access Time (EAT)

Let:

- **c** = TLB access time
- **m** = Main memory access time
- **x** = TLB hit ratio ($0 \le x \le 1$)

$$E.M.A.T = x(c+m) + (1-x)\cdot 1\cdot(c+2m)$$

1 TLB access | 1 MR for actual page | 1 MR for PT + 1 MR for actual page

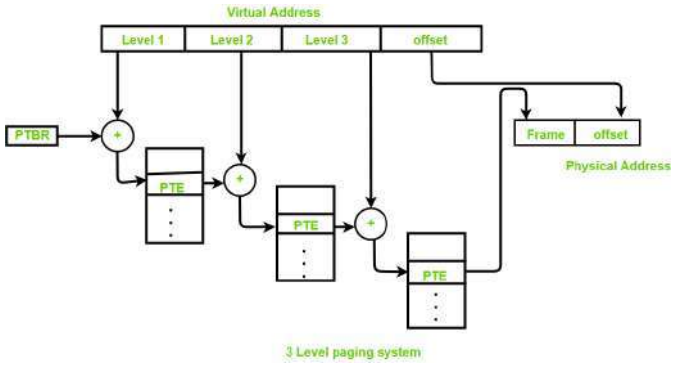


Multilevel Paging

- A single-level page table may be very large → requires huge contiguous memory.
- To reduce this overhead, multilevel paging is used.

Concept

- The page table itself is divided into pages.
- Page tables of all levels are kept in memory.
- Address Translation:
  - Level-1 Page Table entries → pointers to Level-2 Page Table.
  - Level-2 Page Table entries → pointers to Level-3 Page Table.
  - ... and so on.
  - Last-level Page Table entries → actual frame numbers of data pages.



Every Page Table Entry (PTE) contains a frame number.

Performance with TLB

- Let:
  - p = TLB hit ratio
  - c = TLB access time
  - m = Main memory access time
  - n = number of levels in paging

$EMAT = p(c+m) + (1-p)(c+(n+1)m)$

Explanation of Formula

- On TLB Hit → 1 memory access (to get data) + TLB access time.
- On TLB Miss → Need to access all n page tables + 1 data access = (n+1) memory accesses + TLB access time.

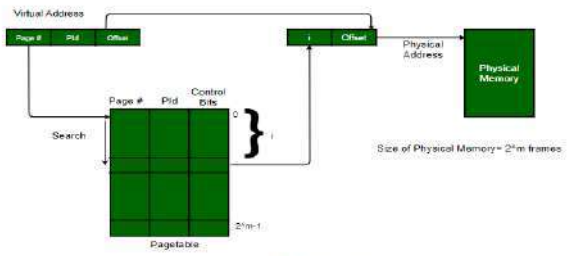
Inverted Paging

Concept

- Unlike normal paging (which maps **pages → frames**), **inverted paging** maps **frames → pages**.
- Only **one global page table** (inverted page table) is maintained instead of separate page tables for each process.

Structure

- Number of entries in the inverted page table = **number of frames in physical memory**.
- Each entry contains:
  - **Process Identifier (PID)** → identifies which process owns the page.
  - **Page Number** → which logical page of that process is stored.
- This tells: *"Which page of which process is currently stored in which frame."*



Inverted PageTable

Advantages

- Saves memory space (since only one page table is maintained, size = number of frames).

Disadvantages

- **Lookup time increases** (must search the whole inverted table).
- More complex to implement (extra mapping required).

Address Translation

1. CPU generates **<Process ID, Page Number, Offset>**.
2. The system searches the **Inverted Page Table** for a matching entry.
3. If found → frame number is obtained → physical address = <Frame Number, Offset>.
4. If not found → **Page Fault** occurs.

To reduce lookup time, a **hash table** is usually used with the inverted page table.

Segmentation

Concept

- **Paging** divides a process into *equal-sized pages*, which does **not match the user's logical view** of a program.
- **Segmentation** divides a process into **variable-length segments**, based on logical units such as:
  - functions, arrays, stack, code, data, etc.

Segment Table

- A **segment table** is maintained per process.
- Number of entries = number of segments in the process.
- Each entry contains:

- **Base** → starting physical address of the segment.
- **Limit** → length of the segment.

Fragmentation

- **Internal Fragmentation** → does not occur (since segments are variable length).
- **External Fragmentation** → usually avoided using compaction, but in general segmentation **does suffer external fragmentation** (unlike paging).
  (Some books note "no internal fragmentation, but external fragmentation exists.")

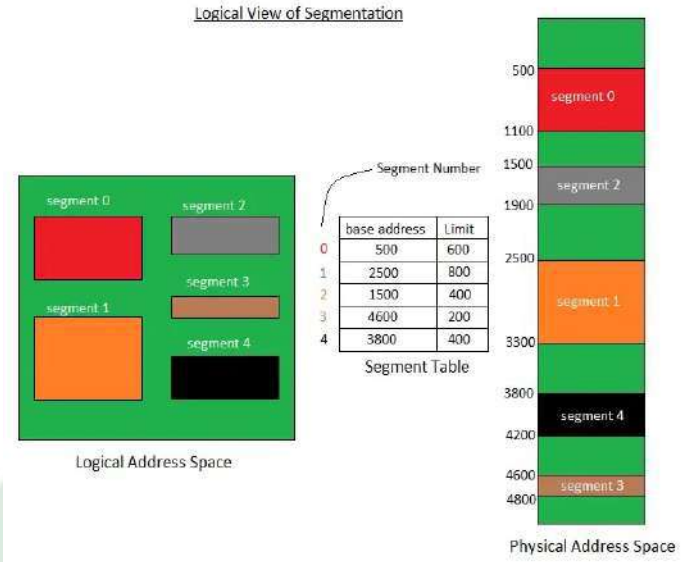
Address Translation

- Logical address = <segment number, offset>
- Using the **segment table**:
  - Check if offset < limit → if yes, valid.
  - Physical address = base + offset.
  - If offset ≥ limit → segmentation fault.

Segmented Paging

Concept

- In **pure segmentation**, entire large segments must be loaded into memory → overhead.
- To reduce this, **paging is applied inside each segment**.
- Thus, each **segment is divided into pages**, and these pages are loaded into memory frames.



Structure

- For each segment, a **Page Table** is maintained.
- Number of entries in a segment's page table = number of pages in that segment.
- Logical address structure:
  ⟨Segment No.,Page No.,Offset⟩\langle \text{Segment No.}, \text{Page No.}, \text{Offset} \rangle⟨Segment No.,Page No.,Offset⟩

Address Translation

1. **Segment Number** → used to locate the **segment's Page Table**.
2. **Page Number** → used as index into that Page Table to find the **Frame Number**.
3. **Offset** → combined with Frame Number to form the **Physical Address**.

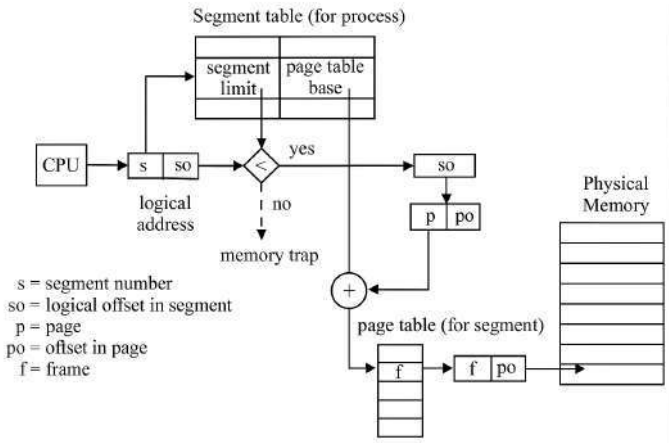
Physical Address=⟨Frame No.,Offset⟩

Performance

- **Lookup time increases** (need to access: Segment Table + Page Table + Memory).
- TLB can be used to speed up access.

Fragmentation

- Suffers only from **Internal Fragmentation** (since pages are fixed size).
- **No External Fragmentation** (same as paging).



Virtual Memory (VM) Concept

- Virtual memory provides an **illusion to the programmer** that a program larger than the available physical memory can be executed.
- It allows **address space sharing** by multiple processes, improving system utilization and flexibility.

Virtual Memory Implementation

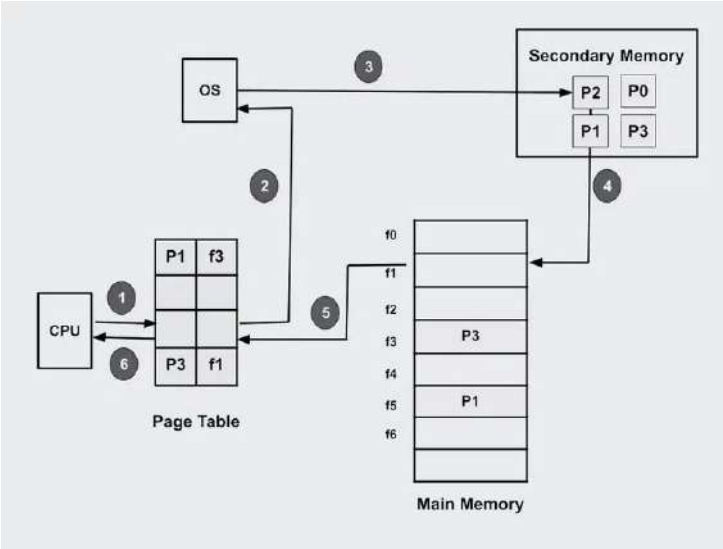
Virtual memory can be implemented using:

1. **Demand Paging** – Only the required pages of a process are loaded into memory on demand.
2. **Demand Segmentation** – Only the required segments of a process are loaded into memory on demand.

Demand Paging

- **Definition**: Loading the required pages from **secondary memory** (disk) into **main memory** only when they are demanded by the CPU is called *Demand Paging*.
- **Page Fault**:
  - If the CPU refers to a page **not present in main memory**, a **page fault** occurs.
  - The OS is then signaled about the fault, locates the required page in secondary storage, and loads it into a free memory frame.
  - If **all frames are full**, a **page replacement algorithm** (like FIFO, LRU, Optimal) is used to decide which page to replace.
  - The **page table** entries are updated accordingly.

- **Page Fault Service Time**:
  - The total time required to handle a page fault (detect, fetch the page, and update tables) is called the **Page Fault Service Time**.



Performance of Virtual Memory

Let:

- S = Page Fault Service Time
- M = Main Memory Access Time
- P = Page Fault Rate, where 0≤P≤10

Effective Memory Access Time (EMAT)
Since every memory access can either be a hit (page is in main memory) or a miss (page fault occurs):
EMAT=P×S+(1−P)M

Demand Paging with TLB

Assume, TLB hit ratio = 'h'
TLB access time = 'c'
Page fault service time = 'S'
Main memory access time = 'M'
Page fault rate = 'P' where 0 ≤ P ≤ 1
The effective memory access time is formulated as
EMAT = h(c + m) + (1 – h)(c + (P × S + (1– P) × M))

Page Replacement Algorithms

1. FIFO (First-In, First-Out)

- When a page fault occurs and all the memory frames are full, FIFO algorithm replaces the oldest page to allocate the page referred by the CPU.
- It is implemented using a queue or time-stamp on pages.
- Sometimes, even after increasing the number of frames, the page fault rate increases.
  - This situation is called Belady's Anomaly.

2. LRU (Least Recently Used)

- LRU algorithm replaces the page that has not been used for the longest period (least recently used page).
- It is implemented using a stack or counter.

3. Optimal

- Optimal algorithm replaces the page that will not be used for the longest period in future.
- For a fixed number of frames, Optimal algorithm gives the least page fault rate.
- It cannot be implemented in real-time as it requires knowledge of future references.

Least Recently Used (LRU)

- Replace the page that has not been used for the longest period of time.
- Performance of LRU is closer to Optimal as it is practical approximation.
- Criteria: Time of Reference.

Most Recently Used (MRU)

- Replace the page that is used most recently.
- Criteria: Time of Reference.

Counting Algorithms

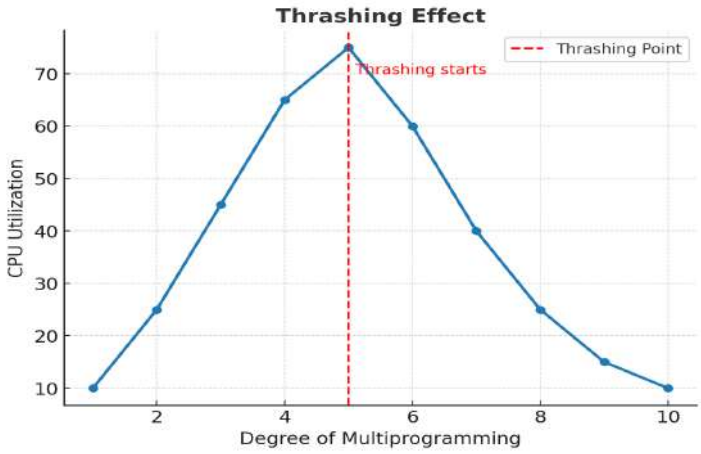
- Least Frequently Used (LFU): Replace the page with the smallest count.
- Most Frequently Used (MFU): Replace the page with the largest count.
- Implementation is expensive.

Variants of LRU (LRU Approximation)

- Approximation and not exactly LRU, they approximate to the behavior of LRU (i.e., they work like LRU).
- **Reference bit (R):**
  - 1 → Page is referred at least once during the current epoch (a period of time).
  - 0 → Page is not referred so far during the current epoch.
- **Second Chance / Clock Algorithm:**
  - It degenerates to FIFO.
  - Criteria is Reference bit.
- **Enhanced Second Chance / Not Recently Used:**
  - Criteria is Reference Bit and Modify/Dirty Bit.
  - **Modify Bit:**
    - 1 → Page is modified.
    - 0 → Page is clean.

Thrashing

- **Definition:**
  Thrashing is a phenomenon in which a process spends more time in page faults and page replacement than in actual execution.
- **Cause:**
  - When CPU utilization is low, the OS increases the degree of multiprogramming (adds more processes).
  - After a certain point, there aren't enough frames to handle all processes → page faults increase heavily.
  - CPU remains idle as it keeps waiting for pages to be swapped in/out.
- **Effect:**
  - System throughput decreases drastically.
  - High paging activity dominates, reducing actual work done.
- **Graphical Behavior:**
  - Initially, as multiprogramming increases → CPU utilization increases.
  - After a threshold, more multiprogramming = less CPU utilization due to thrashing.



Working Set Model

- **Definition:**
  The working set is the set of unique pages that a process has referred to in the last Δ memory references (where Δ is the window size).
  Think of it as a sliding window: within that window, all the distinct pages form the process's working set.

- **Conditions:**
  - If Δ < total number of frames →
    The OS can bring in more processes into memory (higher degree of multiprogramming is possible).
  - If Δ > total number of frames →
    The process does not have enough frames to hold its working set.
    In this case, the OS should use the mid-term scheduler to suspend some processes to avoid thrashing.

- **Purpose / Use:**
  - Helps the OS determine the minimum number of frames a process needs to execute efficiently.
  - Controls CPU utilization and prevents thrashing.
  - Balances the degree of multiprogramming by monitoring working set sizes.

FILE SYSTEM

File

- A file is a **collection of logically related information/records**, stored on secondary storage (like hard disk).

File Attributes

Every file has some **properties** called attributes:

• File Name • File Type • File Size • Location

• Creation Date • Last Modified Date • Permissions • Owner/Author

• Password

File Context

- File data is stored in **File Control Block (FCB)**, which keeps file-related information (metadata).

- Files can be of different **types/formats**, e.g.:
  .doc, .txt, .pdf, .exe
  .obj, .png, .apk, .xls/.xlsx
  .jpg, .mp3, .mp4, .avi/.flv
  .mkv/.3gp, .c/.cpp, .java, .xml/.html

Operations on File

Common operations that can be performed on files:

Create, open, write, read, delete, save save as, close, copy, paste, move, rename send/share, print

Access Methods (ways to access data in Files):

- **Sequential Access:**
  Data is accessed in a **fixed order or linear order,** one record after another, from beginning to end (e.g., magnetic tape).

- **Random Access (Direct Access):**
  Any part of the file can be accessed directly without following a sequence (e.g., hard disk).

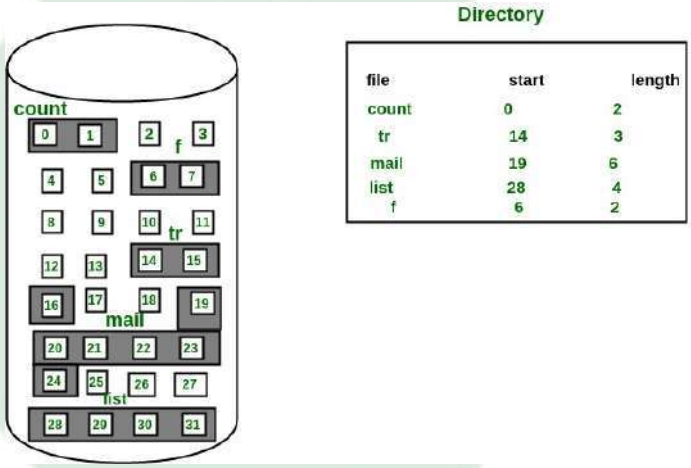
Note: To organize files properly, they are stored inside directories (folders).

Disk Space Allocations Methods:

1. Contiguous Allocation

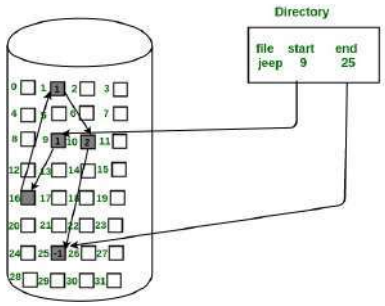
- File blocks are stored **together in a continuous manner** on disk.
- File is described using:
  - Starting block address
  - Size (number of blocks)

- Supports **both sequential and random access**.
- Suffers from **external fragmentation** and **difficulty in expanding size**
- Random location can be calculated using: Starting Block Address + Offset



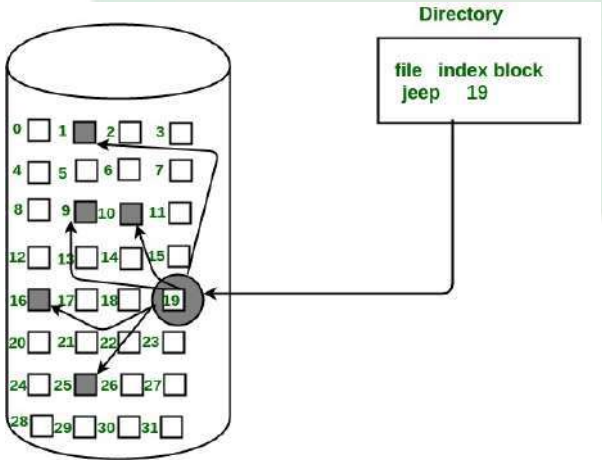
2. Linked (Non–Contiguous) Allocation

- File blocks are stored **anywhere on disk**, each block contains a **pointer to next block**.
- File is described using:
  - Starting block address
  - Ending block address
- No external fragmentation, can easily grow file size.
- Supports **only sequential access**, random access is slow.

- 

3. Indexed Allocation

- Each file has a separate **index block (i-node)**.
- Index block stores **addresses of all disk blocks** used by the file.
- No external fragmentation and supports **direct/random access**.
- If a file is very large, one index block may be insufficient.
- If a file is very small, using a whole index block may waste space.



UNIX I-NODE IMPLEMENTATION

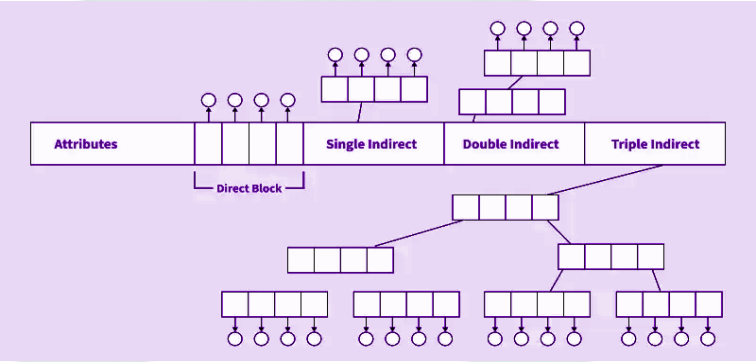
Every file is represented by an **i-node (index node)**.

- i-node stores **metadata** of file:
  - File size, location, owner, date & time, permissions
- It contains multiple pointers to data blocks:

Pointers in i-node:

- **Direct Pointers** → directly point to data blocks of the file.
- **Single Indirect Pointer** → points to a block which contains more pointers to data blocks.
- **Double Indirect Pointer** → points to a block, which points to another block of pointers.
- **Triple Indirect Pointer** → 3-level pointer chain for very large files.

Note : UNIX i-node allows efficient storage for **small files using direct pointers** and supports **very large files using multiple levels of indirect pointers** without external fragmentation.



Maximum possible size of a file (using UNIX i-node):

$$(n_0 + n_1 \times \frac{DB_{Size}}{DBA} + n_2 \times (\frac{DB_{Size}}{DBA})^2 + n_3 \times (\frac{DB_{Size}}{DBA})^3) \times DB_{Size}$$

Where:

- n_0 = number of **direct pointers**
- n_1 = number of **single-indirect pointers**
- n_2 = number of **double-indirect pointers**
- n_3 = number of **triple-indirect pointers**
- `DBSize` = **size of disk block**
- `DBA` = **bits required to store a block address**

Each pointer is assumed to point to a valid data block.

Total size of file system = (number of blocks) × (block size)

- Disk Block Address = `DBA` bits
- If `DBA` bits are used to identify all blocks ⇒ total number of blocks = `2^DBA`

$$\textbf{Total size of file system} = (\text{Disk Block Size}) \times 2^{(\text{DBA in bits})}$$

Disk Free Space Management

When a file is deleted, its disk blocks become free. The OS must keep a **record of all such free blocks**, so that it can **re-use** them when new files are created.
To manage this free space efficiently, two main techniques are used:

1. Free List Approach (Linked List of Free Blocks)

- Stores addresses of all free disk blocks in a **linked list**.
- Each free block contains a pointer to the **next free block**.
- Easy to allocate a block (just take the first address from the list).
- Useful when free space is scattered randomly.
- But needs extra pointers stored *on disk* → takes up space.

2. Bit Map (Bit Vector) Approach

- Uses a **bitmap array** where each block is represented by **1 bit**:
  - 0 → free block
  - 1 → occupied block
- Compact in memory (uses very little space).
- Fast to find the next free block by scanning bits.
- Requires a continuous region in memory to store the bitmap.

Disk Scheduling

- When several processes want to access the hard disk at the same time, their read/write requests are placed in a waiting queue. **Disk scheduling** is the process by which the operating system **chooses the order** in which these waiting requests are sent to the disk.
- **Disk scheduling** decides *which disk request should be processed next* to make disk access **faster and efficient**.

Goal of Disk Scheduling

- Increase throughput (number of I/O requests served).
- Reduce average seek time of the disk.
- Give fair chances to all I/O requests.

Disk Scheduling Algorithms

1. FCFS (First Come First Serve)

→ The requests are served in the **same order they arrive**.
→ Simple but can cause **long head movements**.

2. SSTF (Shortest Seek Time First)

→ The request **closest to the current head position** is served first.
→ Faster than FCFS, but **some requests may wait too long** (starvation).

3. SCAN (Elevator Algorithm)

→ The head moves in **one direction**, serving all requests on the way.
→ When it reaches the end, it **reverses direction** and continues.
→ Like a **lift that goes up and down**.

4. LOOK

→ Same as SCAN, but the head **only goes till the last request** in that direction.
→ Does not go to the physical end of the disk.

5. C-SCAN (Circular SCAN)

→ The head moves in **one direction**, serves requests, and when it reaches the end,

it **jumps to the start** without serving on the way back.
→ Like a **lift that only goes up**, then comes down empty.

6. C-LOOK

→ Same as C-SCAN, but head **only goes till the last request**,

not to the physical end of the disk.

Algorithm	Working Style	Advantages	Disadvantages	Hint
FCFS	Serve requests in arrival order	Very simple, fair (no starvation)	High seek time, slow	Jo pehle aaya, wahi pehle serve hoga
SSTF	Pick request with shortest seek time (nearest)	Less arm movement, good throughput	Far requests may starve	Sobse paas waale ko serve karo
SCAN	Head goes in one direction then reverses (elevator)	No starvation, good for all loads	Needs direction bit, overhead	Lift ki tarah upar-neeche jaata hai
C-SCAN	Moves in one direction only, then jumps to start	Uniform wait time	Implementation complex	Ek hi direction mein ghoomta rahe
LOOK	Like SCAN but stops at last request only	Avoids extra movement, efficient	Slightly complex	Jahan last request ho, bas wahin tak jao

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

STAR MENTOR CS/DA

KHALEEL SIR
ALGORITHM & OS
29 YEARS OF TEACHING EXPERIENCE

CHANDAN SIR
DIGITAL LOGIC
GATE AIR 23 & 26 / EX-ISRO

SATISH SIR
DISCRETE MATHEMATICS
BE in IT from MUMBAI UNIVERSITY

MALLESHAM SIR
M.TECH FROM IIT BOMBAY
AIR – 114, 119, 210 in GATE
(CRACKED GATE 8 TIMES)
14+ YEARS EXPERIENCE

VIJAY SIR
DBMS & COA
M. TECH FROM NIT
14+ YEARS EXPERIENCE

PARTH SIR
DA
IIIT BANGALORE ALUMNUS
FORMER ASSISTANT PROFESSOR

SAKSHI MA'AM
ENGINEERING MATHEMATICS
IIT ROORKEE ALUMNUS

SHAILENDER SIR
C PROGRAMMING & DATA STRUCTURE
M.TECH in Computer Science
15+ YEARS EXPERIENCE

AVINASH SIR
APTITUDE
10+ YEARS OF TEACHING EXPERIENCE

AJAY SIR
PH.D. IN COMPUTER SCIENCE
12+ YEARS EXPERIENCE