Operating System — Exam Survival Pack

Subject Code BT-406 · TU-830(A) · B.Tech IV Sem

Built from Previous Year Question Papers

May 2026

Contents

Welcome — Read This First

Hi. This pack is your exam companion. Open it tonight, read it tomorrow morning, and walk in confident. The whole thing is structured so the **most likely topics come first**, the **memorize-cold stuff** is on one cheat sheet, and **every PYQ has a worked solution** in the back. Nothing is skipped.

> **How to use this in 8 hours:** Skim front matter (15 min) → memorize the master cheat sheet (60 min) → read Top 10 high-probability topics deeply (4 hours) → speed-read remaining topics (90 min) → re-read the cheat sheet and "Top Mistakes" page (30 min). Sleep.

Part 1 — Pattern Analysis & Game Plan

1.1 The Exam Structure (TU-830(A), BT-406)

Section	Description	Marks	Questions to Attempt	Word Limit
A	Very Short Answer	5 × 2 = **10**	All 5 (compulsory)	≤ 75 words
B	Short Answer	2 × 9 = **18**	Any 2 of 3	≤ 200 words
C	Detailed Answer	3 × 14 = **42**	Any 3 of 5	Detailed
	Total	**70**		3 hours

1.2 Topic-by-Topic Probability Table

Topics that show up in **both** papers are nearly guaranteed to repeat. Prepare those first.

Topic	Paper 1 (May 2025)	Paper 2	Total Marks	Probability
Disk Scheduling (SCAN, LOOK, SSTF, FCFS)	Q13(a) — 7	Q8 — 9	**16**	NEAR-CERTAIN
Dining Philosopher Problem	Q9(b) — 7	Q7 — 9	**16**	NEAR-CERTAIN
File System Protection & Security	Q6 — 9; Q9(a)(i)	Q11(b)(i)	**16+**	NEAR-CERTAIN
File Concept, Organization & Access Methods	Q10(a) — 7	Q13(a) — 7	**14**	NEAR-CERTAIN
Banker’s Algorithm / Safe & Unsafe State	Q5 — 2	Q11(a) — 7	**9**	NEAR-CERTAIN
Linked File Allocation	Q9(a)(ii)	Q11(b)(ii)	**~7**	NEAR-CERTAIN
Real Time Operating System	Q1 — 2	Q1 — 2	**4**	NEAR-CERTAIN
I/O Buffering	Q4 — 2	Q5 — 2	**4**	NEAR-CERTAIN
Scheduling — Need / Performance Criteria	Q3 — 2	Q3 — 2	**4**	NEAR-CERTAIN
Disk Storage Theory	Q7 — 9	(overlap)	**9**	HIGH
Page Replacement (FIFO, LRU, Optimal)	Q10(b) — 7	—	**7**	HIGH
IPC Models & Schemes	Q11(a) — 7	—	**7**	HIGH
Monolithic vs Microkernel	Q11(b) — 7	—	**7**	HIGH

Topic	Paper 1 (May 2025)	Paper 2	Total Marks	Probability
Producer–Consumer + Semaphore	Q12(a) — 7	—	**7**	HIGH
RAID levels	Q13(b) — 7	—	**7**	HIGH
Multiuser & Multithreaded Systems	Q8 — 9	—	**9**	HIGH
Deadlock System Model & Characterization	—	Q6 — 9	**9**	HIGH
External vs Internal Fragmentation + Paging	—	Q12(a) — 7	**7**	HIGH
Thrashing	—	Q12(b) — 7	**7**	HIGH
Fixed vs Variable Partitioning	Q12(b)(ii)	Q4 — 2	**~5**	HIGH
Multilevel Feedback Queue	Q12(b)(i)	—	**~3**	MEDIUM
Seek Time / Latency Time	Q2 — 2	—	**2**	MEDIUM
Concurrent Processes	—	Q2 — 2	**2**	MEDIUM
Multiprogramming with Fixed Partitions	—	Q4 — 2	**2**	MEDIUM

1.3 The Must-Prep List

If you read **nothing else**, master these eight. They cover ~70% of the paper:

1. **Disk Scheduling algorithms** (SCAN, LOOK, SSTF, FCFS, C-SCAN)
2. **Dining Philosopher problem** (statement + semaphore solution)
3. **Banker's Algorithm** (Need matrix + safety algorithm)
4. **File system protection & security** + **Linked allocation**
5. **File concept**, organization and access methods
6. **Page replacement** algorithms — be ready to compute FIFO, LRU, Optimal
7. **Producer–Consumer** with semaphores
8. **Deadlock**: Coffman conditions, model, prevention/avoidance

1.4 Target Score (~85% = 60/70)

Section	Aim For	Strategy
A (10 marks)	**9**	Write all 5 confidently. Each in 3–4 lines.
B (18 marks)	**16**	Choose 2 strongest. Disk Scheduling + Dining Philosopher are safest bets.

Section	Aim For	Strategy
C (42 marks)	**35**	Choose 3. Banker’s Algorithm, File concept, Producer-Consumer/IPC are dependable.

1.5 Time Plan (180 minutes)

Block	Time	What you do
First 5 min	5	Read paper top-to-bottom, **mark** the 2 + 3 you’ll attempt in B and C
Section A	20	5 × 4 min, ~50–70 words each
Section B	50	2 × 25 min, ~180 words + diagram
Section C	95	3 × ~32 min, detailed + at least one diagram per answer
Buffer	10	Review, fix numbering, finish unfinished sentences

Part 2 — Master Cheat Sheet (Memorize Cold)

2.1 One-line Definitions

Term	Crisp Definition
Operating System	Software that manages hardware resources and acts as an interface between user and hardware.
Process	A program in execution, with its own PCB, code, data, stack, and state.
Thread	Lightweight unit of execution within a process; shares code/data, has its own stack and registers.
PCB	Process Control Block — data structure holding PID, state, PC, registers, memory info, I/O info.
Scheduling	Deciding which ready process gets the CPU next.
Context switch	Saving one process's state and loading another's.
Deadlock	Two or more processes each waiting for a resource the other holds; none can proceed.
Starvation	A process is repeatedly denied resources/CPU.
Semaphore	Integer variable accessed only via `wait()` and `signal()` for synchronization.
Mutex	Binary semaphore — provides mutual exclusion on a critical section.
Page	Fixed-size logical block of a process's address space.
Frame	Fixed-size physical block in memory; pages are loaded into frames.
Paging	Mapping logical pages to physical frames using a page table.
Thrashing	System spends more time paging than executing useful work.
File	Named collection of related information stored on secondary storage.
Seek time	Time for disk arm to move the head to the desired track.
Rotational latency	Time for the desired sector to rotate under the head.
Real-time OS	OS that must process events within strict time deadlines.
Multiuser OS	Allows multiple users to access the system simultaneously.
Multithreaded OS	Allows multiple threads within a single process to run concurrently.

Term	Crisp Definition
RAID	Redundant Array of Independent Disks — combine multiple disks for performance/reliability.
Monolithic kernel	All OS services run in kernel space as one large program.
Microkernel	Only essentials (IPC, scheduling, memory) in kernel; rest run as user-space servers.

2.2 Formulas, Numbers, Tuples to Recall

What	Formula / Notation
Effective Access Time (paging, no TLB)	$EAT = 2 \times$ memory access time
EAT (with TLB, hit ratio h)	$EAT = h(t+m) + (1-h)(t+2m)$ where t = TLB time, m = mem time
Page fault service time (avg)	$T = (1-p) \cdot m + p \cdot$ page-fault-time
Disk access time	$T =$ seek + rotational latency + transfer
Throughput	$\#$processes completed$/$time
Turnaround time	$T_{turn} = T_{complete} - T_{arrival}$
Waiting time	$T_{wait} = T_{turn} - T_{burst}$
Response time	$T_{response} = T_{first\,CPU} - T_{arrival}$
Need (Banker's)	Need $=$ Max $-$ Allocation
Producer-Consumer semaphores	`empty = n`, `full = 0`, `mutex = 1`
Dining Philosopher semaphores	`fork[5] = 1` each; pickup-left, pickup-right; signal-both

2.3 The Four Coffman Conditions (memorize verbatim)

Deadlock arises if and only if all four hold simultaneously:

1. **Mutual Exclusion** — resource cannot be shared
2. **Hold and Wait** — process holding resources requests more
3. **No Preemption** — resources cannot be forcibly taken
4. **Circular Wait** — a cycle of waiting processes exists

2.4 CPU Scheduling Criteria (5 of them)

Criterion	Goal
CPU utilization	Maximize (keep CPU busy)
Throughput	Maximize (jobs per unit time)
Turnaround time	Minimize
Waiting time	Minimize
Response time	Minimize (for interactive systems)

2.5 File Operations (the basic six)

`create · write · read · reposition (seek) · delete · truncate`

Part 3 — Topic-by-Topic Teaching

This section teaches every topic from the PYQs, **ordered by probability**. Read top-to-bottom.

3.1 Disk Scheduling — SCAN, LOOK, SSTF, FCFS, C-SCAN

Probability: NEAR-CERTAIN · Marks weight: 16+

The Idea

Imagine a librarian fetching books from a long shelf for a queue of impatient readers. She can't tele-port; she has to walk. The question is: in what order should she fetch them so she walks the least?

Disk requests are exactly the same problem. The "shelf" is the platter; the "librarian" is the read/write head. Each request asks the head to go to a specific track. The OS picks the order using a **disk scheduling algorithm**.

Why care? Because **seek time dominates** disk access time. Smart scheduling means more I/O done per second.

The Five Algorithms in One Table

Algorithm	One-line rule	Pros	Cons
FCFS	Serve in arrival order	Fair, no starvation	High seek time
SSTF (Shortest Seek Time First)	Always go to the closest request	Low seek time	Can **starve** far requests
SCAN (Elevator)	Move in one direction servicing everything, hit the end of disk, reverse	Uniform wait	Pointless trip to end of disk
LOOK	Like SCAN, but reverse at the **last request**, not end of disk	Less wasted movement	Slightly more complex
C-SCAN	One-way SCAN: at end, jump back to start, sweep again	Even more uniform wait	Big jump at end

Worked example (uses the May-2025 paper data)

Setup: disk has 100 tracks (0–99). Requests: 45, 20, 90, 10, 50, 60, 80, 25, 70. Head starts at track **49**. Assume current direction is towards higher tracks.

SSTF (shortest seek time first) Always pick the closest pending request.

Step	Head at	Distances to remaining	Pick	Move
1	49	45(4), 20(29), 90(41), 10(39), 50(1), 60(11), 80(31), 25(24), 70(21)	**50**	1
2	50	45(5), 20(30), 90(40), 10(40), 60(10), 80(30), 25(25), 70(20)	**45**	5
3	45	20(25), 90(45), 10(35), 60(15), 80(35), 25(20), 70(25)	**60**	15
4	60	20(40), 90(30), 10(50), 80(20), 25(35), 70(10)	**70**	10
5	70	20(50), 90(20), 10(60), 80(10), 25(45)	**80**	10
6	80	20(60), 90(10), 10(70), 25(55)	**90**	10
7	90	20(70), 10(80), 25(65)	**25**	65
8	25	20(5), 10(15)	**20**	5
9	20	10(10)	**10**	10

Total seek = 1+5+15+10+10+10+65+5+10 = 131 tracks

SCAN (moving towards 99 first) Sort the requests on either side. Above 49: {50, 60, 70, 80, 90}. Below 49: {45, 25, 20, 10}.

Head goes 49 → 50 → 60 → 70 → 80 → 90 → **99 (end)** → 45 → 25 → 20 → 10.

- Up: $99 - 49 = 50$
- Down: $99 - 10 = 89$
- **Total = 50 + 89 = 139 tracks**

LOOK (moving towards higher tracks first) Same as SCAN but we reverse at the **last request** instead of the disk end.

Head: 49 → 50 → 60 → 70 → 80 → 90 → 45 → 25 → 20 → 10.

- Up: $90 - 49 = 41$
- Down: $90 - 10 = 80$
- **Total = 41 + 80 = 121 tracks**

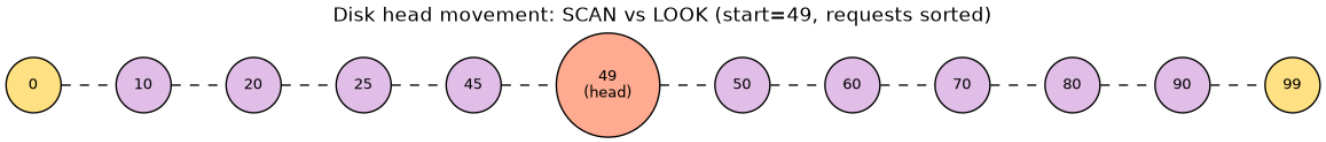

Figure 1: Track layout reference for SCAN and LOOK on the May-2025 request set.

Common traps

- **Direction matters.** If the question doesn't state direction, assume "towards higher track numbers" and **state your assumption**.
- For **SCAN**, the head touches the disk end (track 99 or 199 here) even if no request lives there. For **LOOK**, it does not.
- For **C-SCAN**, after reaching the end, the head jumps directly to track 0 (this jump is sometimes counted, sometimes not — note your assumption).
- Don't forget the **head's starting position** in the total.

Linked PYQs

- **Paper 1, Q13(a)** — exactly the worked example above (7 marks).
- **Paper 2, Q8** — same algorithms on a 200-cylinder disk, head at 143 (9 marks).

3.2 Dining Philosopher Problem

Probability: NEAR-CERTAIN · Marks weight: 16

The Idea

Five professors sit around a circular table. There's one bowl of noodles in front of each, but only **five chopsticks total**, one between each pair. To eat, a philosopher needs **both** the chopstick on their left and the one on their right. They alternate between thinking and eating.

The catch: if every philosopher simultaneously picks up their **left** chopstick, nobody can ever pick up their right one. Frozen. That's deadlock. The problem is a parable about **shared resources, deadlock, and starvation**.

Formal statement

Given N philosophers $P_0, \dots, P_{N-1}$ around a circular table with N forks $f_0, \dots, f_{N-1}$, where philosopher P_i needs forks f_i (left) and $f_{(i+1) \bmod N}$ (right) to eat, design a synchronization protocol that:

- prevents **deadlock**,
- prevents **starvation**,
- allows maximum concurrency.

The naive semaphore solution (and why it deadlocks)

Each fork is a binary semaphore initialized to 1.

```
semaphore fork[5] = {1, 1, 1, 1, 1};

philosopher_i():
    while (true) {
        think();
        wait(fork[i]);              // pick up left
        wait(fork[(i+1) % 5]);      // pick up right
        eat();
        signal(fork[i]);
        signal(fork[(i+1) % 5]);
    }
```

Problem: if every philosopher executes `wait(fork[i])` simultaneously, they all hold their left fork and wait forever for the right. Classic deadlock — all four Coffman conditions hold.

Standard fixes (you should know at least one)

Fix	Idea	Trade-off
Allow only N-1 at table	Use an extra semaphore `room = N-1`. Block one philosopher from sitting.	Simple; provably deadlock-free.

Fix	Idea	Trade-off
Asymmetric pickup	Odd philosophers pick up left first, even pick up right first. Breaks circular wait.	Elegant.
Pick up both forks atomically	A monitor or extra semaphore ensures both forks are taken in one critical section.	Reduces concurrency.

Exam-ready solution (asymmetric)

> "We solve the dining philosopher problem using semaphores with an asymmetric pickup rule. Each fork is a binary semaphore initialized to 1. Odd-numbered philosophers pick up the **right** fork first, then the **left**; even-numbered philosophers do the opposite. This breaks the circular-wait condition since at least one philosopher reaches for forks in the opposite order, so a complete cycle of waiting cannot form. Hence the system is deadlock-free."

Common traps

- Don't forget to **signal both forks** after eating; otherwise other philosophers starve.
- A `wait(fork[i])` order chosen identically by all philosophers is the **buggy** version. The exam often expects you to identify why it fails.
- "Starvation-free" is **stronger** than "deadlock-free". The asymmetric fix prevents deadlock but doesn't guarantee fairness.

Linked PYQs

- **Paper 1, Q9(b)** — "Explain in detail about the Dining Philosopher Problem" (7 marks).
- **Paper 2, Q7** — same question (9 marks). Use almost the same answer; the 9-mark version adds an extra paragraph about starvation and the room-semaphore fix.

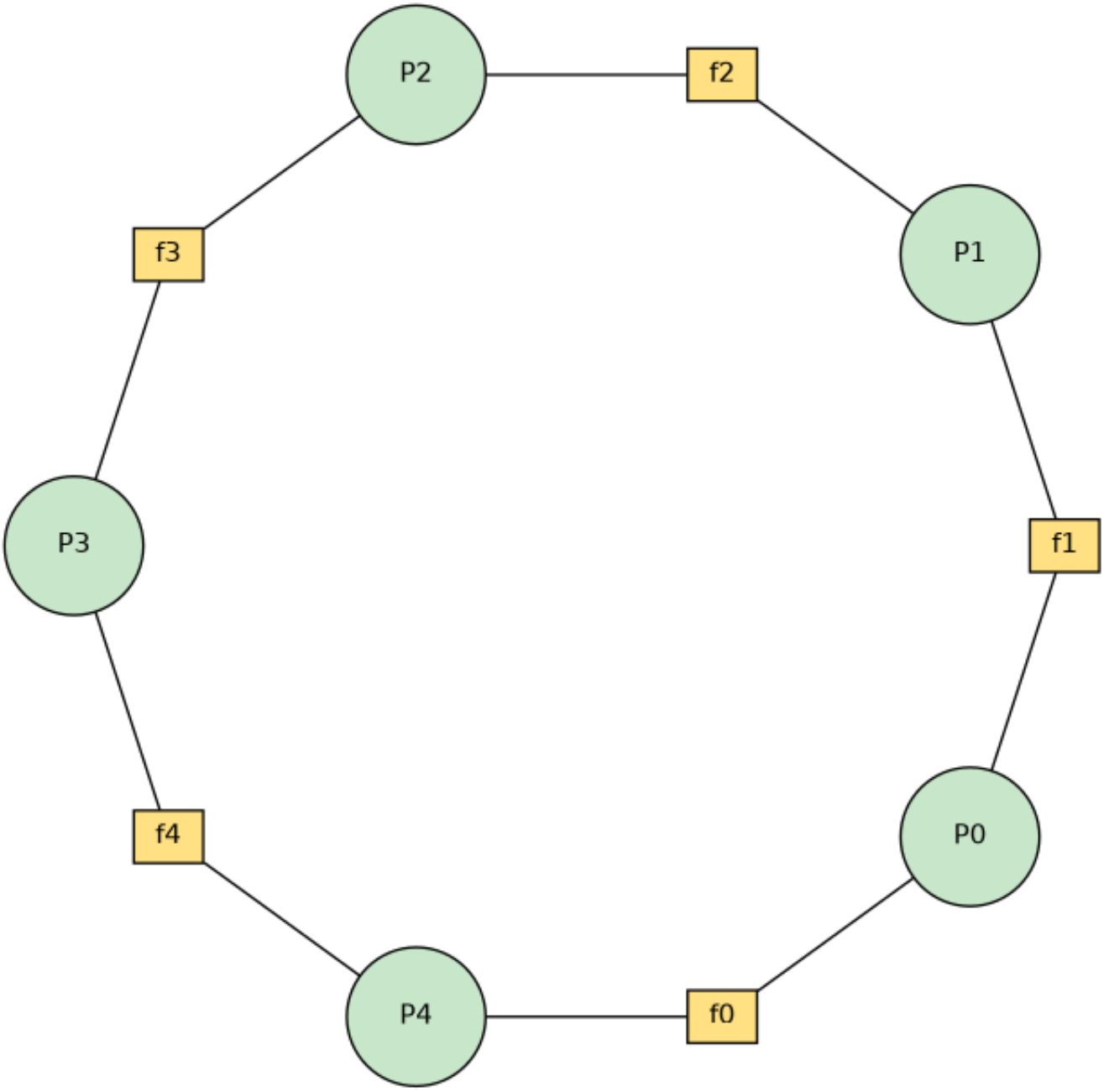

Figure 2: Dining philosopher table layout. Five processes (philosophers), five shared resources (forks/chopsticks).

3.3 Banker's Algorithm (Safe / Unsafe State)

Probability: NEAR-CERTAIN · Marks weight: 9

The Idea

You're a banker with a fixed amount of cash. Each customer has told you the maximum they'll ever need to borrow. They borrow in chunks. Your job: **never grant a loan that could leave you unable to satisfy at least one customer later.**

That's literally it. The OS plays the banker. Processes are the customers. Resource instances are the cash. The "Banker's Algorithm" decides whether a new request keeps the system **safe** — meaning there exists at least one order in which every process can finish.

The Three Matrices

Matrix	Meaning
Allocation	What each process currently holds
Max	The most each process will ever need
Need	What each still might ask for. **Need = Max − Allocation**

Plus a vector: **Available** = total resources − sum of Allocation.

Safety Algorithm (the procedure)

1. Let `Work = Available`, `Finish[i] = false` for all i.
2. Find some i such that `Finish[i] == false` **and** `Need[i] ≤ Work`.
3. If found: `Work = Work + Allocation[i]`, `Finish[i] = true`. Repeat step 2.
4. If no such i exists and some Finish[i] is still false → **UNSAFE**.
5. If every Finish[i] = true → **SAFE**. The order in which you marked them finished is a **safe sequence**.

Worked example

Total resources: A=10, B=5, C=7. Three processes.

Process	Allocation (A,B,C)	Max (A,B,C)
P0	0,1,0	7,5,3
P1	2,0,0	3,2,2
P2	3,0,2	9,0,2
P3	2,1,1	2,2,2
P4	0,0,2	4,3,3

Step 1: compute Need = Max − Allocation

Process	Need (A,B,C)
P0	7,4,3
P1	1,2,2
P2	6,0,0
P3	0,1,1
P4	4,3,1

Step 2: compute Available

Sum of Allocation: A = 0+2+3+2+0 = 7, B = 1+0+0+1+0 = 2, C = 0+0+2+1+2 = 5. **Available = (10−7, 5−2, 7−5) = (3, 3, 2).**

Step 3: run safety algorithm

Try	Work	P needing ≤ Work	Pick
1	(3,3,2)	P1 needs (1,2,2) ✓; P3 needs (0,1,1) ✓	**P1** → Work = (3+2, 3+0, 2+0) = (5,3,2)
2	(5,3,2)	P3 needs (0,1,1) ✓	**P3** → Work = (5+2, 3+1, 2+1) = (7,4,3)
3	(7,4,3)	P0 needs (7,4,3) ✓; P4 needs (4,3,1) ✓	**P4** → Work = (7+0, 4+0, 3+2) = (7,4,5)
4	(7,4,5)	P0 needs (7,4,3) ✓; P2 needs (6,0,0) ✓	**P0** → Work = (7+0, 4+1, 5+0) = (7,5,5)
5	(7,5,5)	P2 needs (6,0,0) ✓	**P2** → Work = (7+3, 5+0, 5+2) = (10,5,7)

All processes finished. **Safe sequence: ⟨ P1, P3, P4, P0, P2 ⟩.** System is in a **safe state**.

Resource-Request Algorithm (when a process actually asks)

When P_i requests `Request[i]`:

1. If `Request[i] > Need[i]` → error (process exceeded its claim).
2. If `Request[i] > Available` → wait (resources unavailable).
3. **Tentatively allocate**: `Available -= Request[i]`, `Allocation[i] += Request[i]`, `Need[i] -= Request[i]`.
4. Run safety algorithm. If SAFE → grant. If UNSAFE → **roll back** and make P_i wait.

Safe vs Unsafe vs Deadlock — the three states

> A **safe state** is one in which the OS can satisfy every process's maximum future demand in **some** order without deadlock. An **unsafe state** is one where no such order is guaranteed — deadlock is **possible** but not certain. A **deadlocked state** is unsafe and all involved processes are blocked.

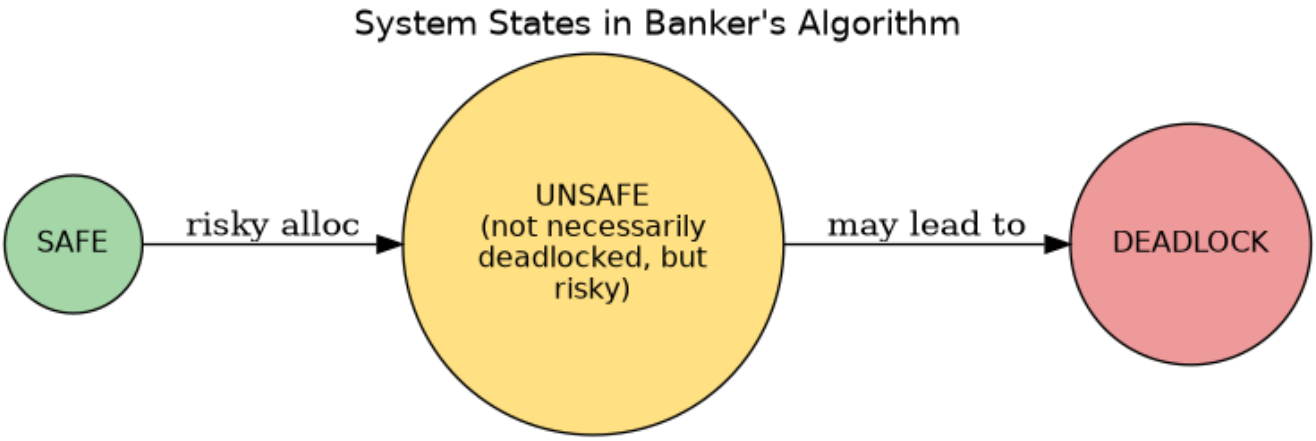

Figure 3: Safe ⊃ Unsafe ⊃ Deadlock. Unsafe doesn't mean deadlocked, just "could deadlock if unlucky."

Common traps

- Always compute **Need** first; many students confuse Max and Need.
- "Safe" ≠ "deadlock-free now" — it means **a safe sequence exists**.
- The Banker's algorithm requires processes to declare **Max in advance** (a strong assumption rarely met in real systems; that's why it's avoidance, not real-world standard).
- Resources of multiple types need vector comparison: ≤ means **every component** is ≤.

Linked PYQs

- **Paper 1, Q5** — "Safe state and unsafe state" (2 marks). Two sentences each.
- **Paper 2, Q11(a)** — full banker's algorithm with Need matrix and safe-state check (7 marks).

3.4 File Concept, File Organization & Access Methods

Probability: NEAR-CERTAIN · Marks weight: 14

The Idea

A **file** is just a named bucket for related bytes — your photo, a Word document, an executable. The OS hides the messy detail (disk blocks, sectors) and gives you a clean abstraction: open, read, write, close.

Three layers you should remember:

1. **File concept** — what a file is, its attributes and operations.
2. **File organization** — how the bytes are arranged on disk (sequential, indexed, hashed).
3. **Access methods** — how a program reads/writes (sequential, direct, indexed).

File Attributes (a file has all of these)

Attribute	What it is
Name	Human-readable label
Identifier	Unique number used inside the file system
Type	Determined by extension or magic number
Location	Pointer to file's location on the device
Size	Current size in bytes
Protection	Access permissions (read/write/execute)
Time, date, user-id	Metadata used for protection, security, monitoring

File Operations

`create · open · read · write · seek (reposition) · delete · close · truncate · append`

File Organization Methods

Organization	How records are stored	Pros	Cons
Sequential	Records in order, one after the other	Simple, fast bulk read	Slow random access
Indexed	Index block holds pointers to records	Fast random access	Index uses extra space
Indexed-Sequential	Sequential file with sparse index	Compromise: decent for both	More complex
Hashed	Hash function maps key → disk block	Constant-time access	Collisions; hard to grow

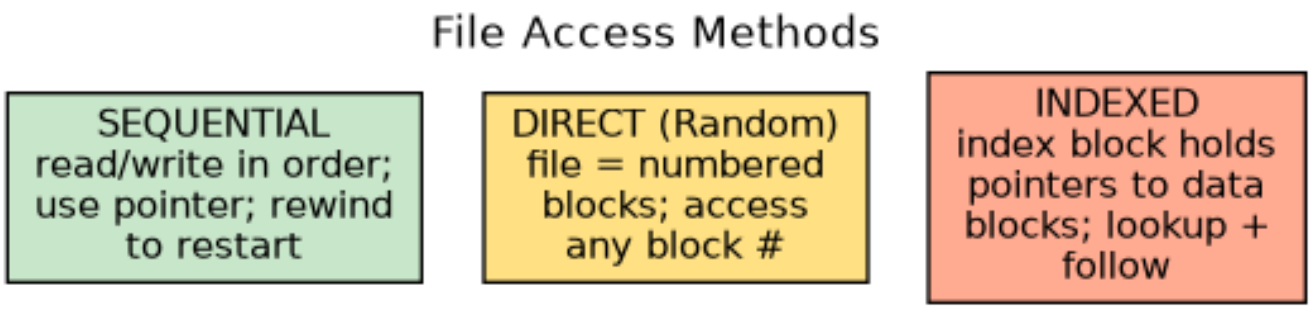

Figure 4: Three classic ways a program can read a file.

Access Methods

Sequential access. Read or write byte by byte, in order. There's a current-position pointer that auto-advances. Like reading a book front-to-back. Used by editors, compilers, most everyday programs.

Direct (random) access. File treated as numbered blocks; you can jump to block n in $O(1)$. Used by databases, indexed files. Operations are `read n`, `write n`, `position n`.

Indexed access. The file has an index block whose entries are pointers to data blocks. To read record r, you look up the index, then go to the data block. Used in databases, large indexed files.

File System Layers

The OS implements files through stacked modules. Going top to bottom:

Layer	Job
Application Programs	The user-facing programs
Logical File System	Manages metadata, directory structure, file-level protection
File Organization Module	Translates logical block addresses to physical block addresses; manages free-space list
Basic File System	Issues generic commands to drivers ("read physical block X")
I/O Control	Device drivers and interrupt handlers
Hardware	Disk

Common traps

- Don't confuse **organization** (how bytes sit on disk) with **access method** (how you read them). A file can be sequentially organized yet accessed directly via byte offset.
- The "indexed" access method needs an index block — note its overhead.
- File operations include the often-forgotten **truncate** (shrink to zero without deleting).

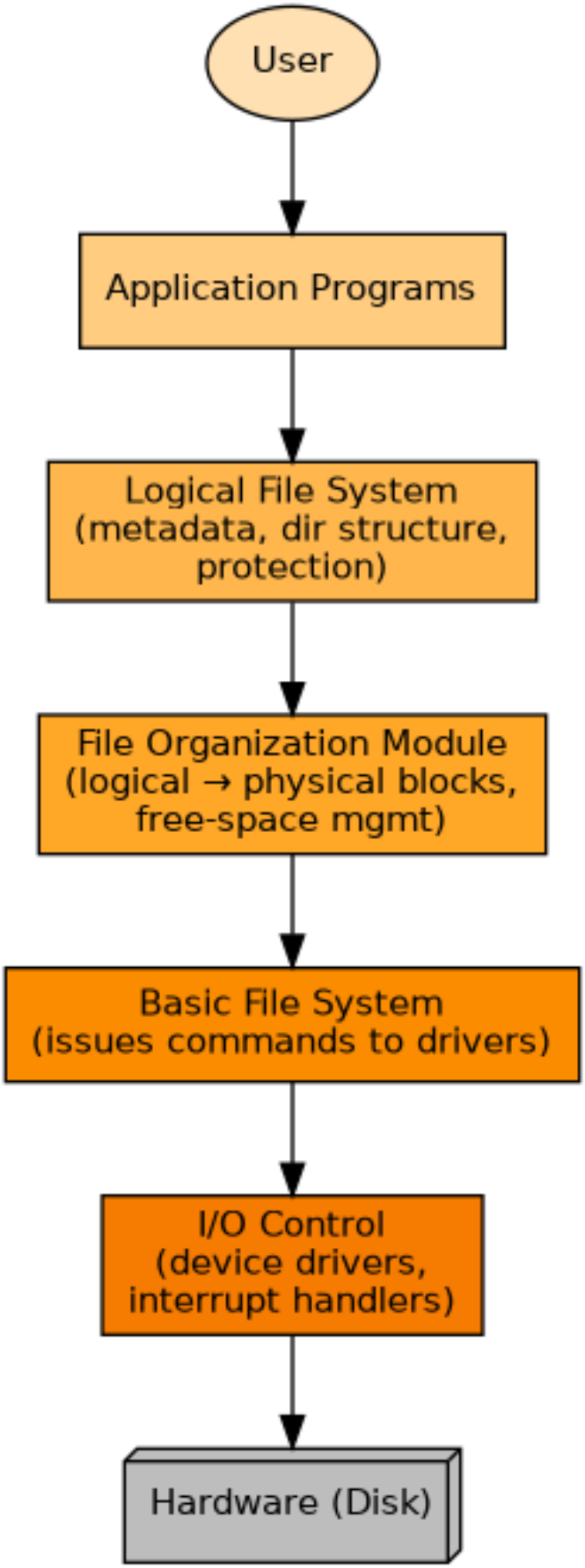

Figure 5: File system layered architecture.

Linked PYQs

- **Paper 1, Q10(a)** — “Explain about File concept. Define File organization and access mechanism” (7 marks).
- **Paper 2, Q13(a)** — “What are files and explain the access method for files” (7 marks).

3.5 File System Protection & Security

Probability: NEAR-CERTAIN · Marks weight: 16+

The Idea

Protection is the *mechanism* — how the OS prevents one user from messing with another's files. **Security** is the *policy* — defending against external threats like viruses, network attackers, and unauthorized logins. Both work together: protection enforces what security policy says.

Protection Concepts

Each file has an associated **access control** describing who can do what.

Operation	Meaning
Read	Read from the file
Write	Write or rewrite the file
Execute	Load and run as a program
Append	Write new data at the end only
Delete	Remove the file
List	Determine attributes (name, size) of the file

Common Protection Mechanisms

Mechanism	Idea	Pros	Cons
Access Control List (ACL)	Each file stores a list of (user, permissions) pairs	Fine-grained, per-user control	List can grow large
Owner / Group / World (Unix)	Three classes only; each gets r/w/x bits	Compact (9 bits)	Coarse-grained
Capabilities	Each user holds a list of (object, rights); presenting it grants access	Strong principle of least privilege	Capability management complex
Password-per-file	Each file requires a password	Simple	Users have to remember many; passwords leak

Security: threats to defend against

Threat	What it is	Defence
Trojan Horse	Hidden malicious code inside trusted program	Code review, sandboxing, signed binaries
Trap Door	Hidden backdoor left by developer	Audit, careful code review

Threat	What it is	Defence
Virus	Code that attaches itself to legitimate programs and spreads	Antivirus, signed binaries
Worm	Self-replicating program that uses network to spread	Patching, firewalls
Denial of Service	Resource exhaustion to halt service	Rate limits, redundancy
Phishing / Social	Tricking users into giving credentials	Training, MFA
Stack/Buffer Overflow	Overwriting memory to inject code	Bounds checking, ASLR, DEP/NX bit

Authentication, the first line of defence

Method	Example
Something you know	Password, PIN
Something you have	Smart card, OTP device
Something you are	Biometric (fingerprint, iris)

Strong systems combine two of these (**multi-factor**).

Exam-ready paragraph

> "File system protection is the OS mechanism that controls who can perform which operations on a file. The typical operations are read, write, execute, append, delete, and list. Protection is enforced through access control lists — which store per-user permissions — or through coarser owner/group/world bits as in Unix. Security extends this idea against external attackers: it defends against viruses, trojans, worms, denial-of-service, and unauthorized access. Authentication (passwords, tokens, biometrics) ensures only legitimate users obtain identities, while encryption protects file contents at rest and in transit."

Common traps

- "Protection" and "security" are **not synonyms** in OS exams. Protection = internal mechanism, security = external policy.
- Access bits in Unix are **rwx** for **owner**, **group**, **others** — total nine bits.
- ACLs are more expressive than the Unix 9-bit scheme but require more storage.

Linked PYQs

- **Paper 1, Q6** — full 9-mark "File system protection and security" answer.
- **Paper 1, Q9(a)(i)** — short notes on the same (3.5 marks within a larger Q).
- **Paper 2, Q11(b)(i)** — short notes on the same (3.5 marks).

3.6 Linked File Allocation

Probability: NEAR-CERTAIN · Marks weight: 7

The Idea

When the OS needs to store a file on disk, it can't always find one big contiguous region of free space. So it stores the file as a **chain of disk blocks scattered anywhere**, with each block holding a pointer to the next one. Like a treasure hunt: each clue tells you where the next clue is.

How it works

- The directory entry stores **start block** and (optionally) **end block** pointers.
- Each disk block holds **data + a pointer to the next block**.
- To read sequentially, follow the chain.
- The last block points to a special "null" or end-of-file marker.

Comparison of the three allocation strategies

Method	How blocks are arranged	Direct access?	External fragmentation?
Contiguous	One continuous run of blocks	Yes (block n = start + n)	Yes (huge problem)
Linked	Scattered blocks chained by pointers	No (must traverse)	No
Indexed	One index block holds all block pointers	Yes	No

Pros & cons

Advantages	**Disadvantages**
No external fragmentation	Slow **random access** — must follow the chain
File can grow easily	Each block loses some space to the next-pointer
Easy to implement	Single corrupted pointer breaks the file
	Reliability: pointers must be backed up (FAT, doubly linked)

Variant: File Allocation Table (FAT)

Instead of storing the "next pointer" inside each data block, put all pointers in a **single table** at the start of the disk. Each entry FAT[i] tells you the next block after block i, or end-of-file. This is what MS-DOS and many flash drives use. Fast traversal because the FAT is cached in memory.

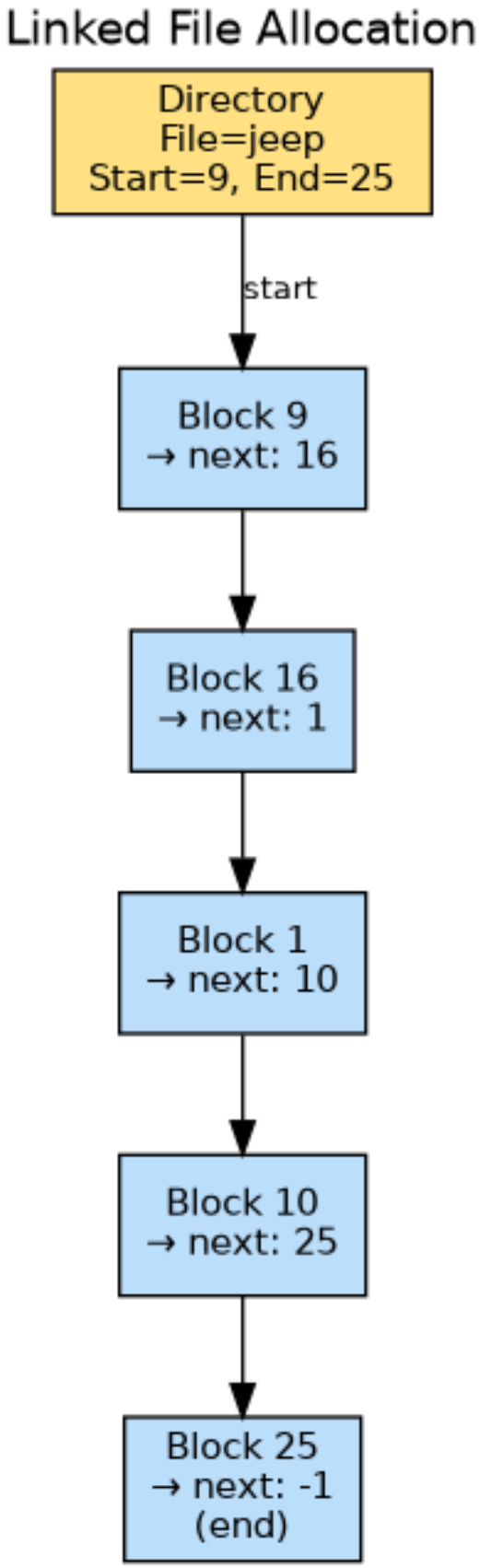

Figure 6: Linked allocation: the directory entry only knows the start (and optionally end) block. Each block stores a pointer to the next.

Common traps

- Random access in pure linked allocation is **O(n)**; FAT improves this but doesn't make it O(1).
- A "lost pointer" is catastrophic — exam answer should mention reliability.
- Linked allocation uses some bytes of every block for the next-pointer; effective data capacity per block is reduced.

Linked PYQs

- **Paper 1, Q9(a)(ii)** — short notes on linked file allocation (3.5 marks).
- **Paper 2, Q11(b)(ii)** — short notes on linked file allocation methods (3.5 marks).

3.7 Page Replacement Algorithms — FIFO, LRU, Optimal

Probability: HIGH · Marks weight: 7

The Idea

Your computer has limited RAM (frames). Your programs use more memory (pages) than fits. When a page is needed but isn't in RAM, a **page fault** happens — the OS picks one currently loaded page to **evict** and brings the new one in. Which page to evict? That's the **replacement algorithm**.

The cost is huge: a page fault means a slow disk read. Fewer faults = faster system. So we want the **smartest** eviction policy.

The Three Algorithms

Algorithm	Rule for who to evict
FIFO	The page that arrived **earliest**
LRU (Least Recently Used)	The page that has been **unused for the longest**
Optimal (OPT/Belady)	The page that **will not be used for the longest** in the future (oracle — unimplementable, but a theoretical lower bound)

Worked example (from May-2025 Q10(b))

Reference string: `0, 9, 0, 1, 8, 1, 8, 7, 8, 7, 1, 2, 8, 2, 7, 8, 2, 3, 8, 3`

Frames available: **3**.

FIFO — evict the oldest-loaded frame.

#	Ref	Frame state	Fault?
1	0	[0]	F
2	9	[0,9]	F
3	0	[0,9]	hit
4	1	[0,9,1]	F
5	8	[9,1,8] (evict 0)	F
6	1	[9,1,8]	hit
7	8	[9,1,8]	hit
8	7	[1,8,7] (evict 9)	F
9	8	[1,8,7]	hit
10	7	[1,8,7]	hit
11	1	[1,8,7]	hit
12	2	[8,7,2] (evict 1)	F
13	8	[8,7,2]	hit
14	2	[8,7,2]	hit
15	7	[8,7,2]	hit
16	8	[8,7,2]	hit
17	2	[8,7,2]	hit

#	Ref	Frame state	Fault?
18	3	[7,2,3] (evict 8)	F
19	8	[2,3,8] (evict 7)	F
20	3	[2,3,8]	hit

FIFO page faults = 8

LRU — evict the page used longest ago.

#	Ref	Frame state	Fault?
1	0	[0]	F
2	9	[0,9]	F
3	0	[9,0]	hit
4	1	[9,0,1]	F
5	8	[0,1,8] (evict 9 — least recently used)	F
6	1	[0,8,1]	hit
7	8	[0,1,8]	hit
8	7	[1,8,7] (evict 0)	F
9	8	[1,7,8]	hit
10	7	[1,8,7]	hit
11	1	[8,7,1]	hit
12	2	[7,1,2] (evict 8)	F
13	8	[1,2,8] (evict 7)	F
14	2	[1,8,2]	hit
15	7	[8,2,7] (evict 1)	F
16	8	[2,7,8]	hit
17	2	[7,8,2]	hit
18	3	[8,2,3] (evict 7)	F
19	8	[2,3,8]	hit
20	3	[2,8,3]	hit

LRU page faults = 9

Optimal — evict the page whose next use is furthest in the future.

#	Ref	Frame state	Notes / Fault?
1	0	[0]	F
2	9	[0,9]	F
3	0	[0,9]	hit
4	1	[0,9,1]	F
5	8	[8,9,1] (evict 0 — no more uses)	F
6	1	hit	
7	8	hit	
8	7	[8,7,1] (evict 9 — never used again)	F
9	8	hit	
10	7	hit	
11	1	hit	

#	Ref	Frame state	Notes / Fault?
12	2	[8,2,1] (evict 7 — used later but 1 used sooner) — careful! Look ahead: future = 8,2,7,8,2,3,8,3. Page 1 not used again, so evict 1. State [8,2,7]	F
13	8	hit	
14	2	hit	
15	7	hit	
16	8	hit	
17	2	hit	
18	3	future = 8,3. Page 7 not used again. Evict 7. [8,2,3]	F
19	8	hit	
20	3	hit	

Optimal page faults = 7

Summary table

Algorithm	Page faults	Notes
FIFO	**8**	Vulnerable to Belady’s anomaly
LRU	**9**	Slightly worse here; typically among best in practice
Optimal	**7**	Theoretical lower bound

Note: small differences in tracking can change LRU/FIFO counts by 1. Always **show your table** in the exam; partial marks are saved by the working.

Common traps

- **Belady’s Anomaly** — for FIFO, increasing the number of frames can *increase* page faults. Mention it.
- Optimal needs future knowledge — explicitly state it’s a theoretical baseline.
- Track the order carefully for LRU: every access updates a recency timestamp.

Linked PYQs

- **Paper 1, Q10(b)** — exact problem above; 7 marks.

3.8 Inter-Process Communication (IPC) — Models & Schemes

Probability: HIGH · Marks weight: 7

The Idea

Processes are isolated by design — one process can't peek into another's memory. But often they need to **talk**: a printer daemon receives jobs, a web server receives requests, two threads coordinate work. So the OS provides **inter-process communication (IPC)** primitives.

There are two big families of models. Memorize both.

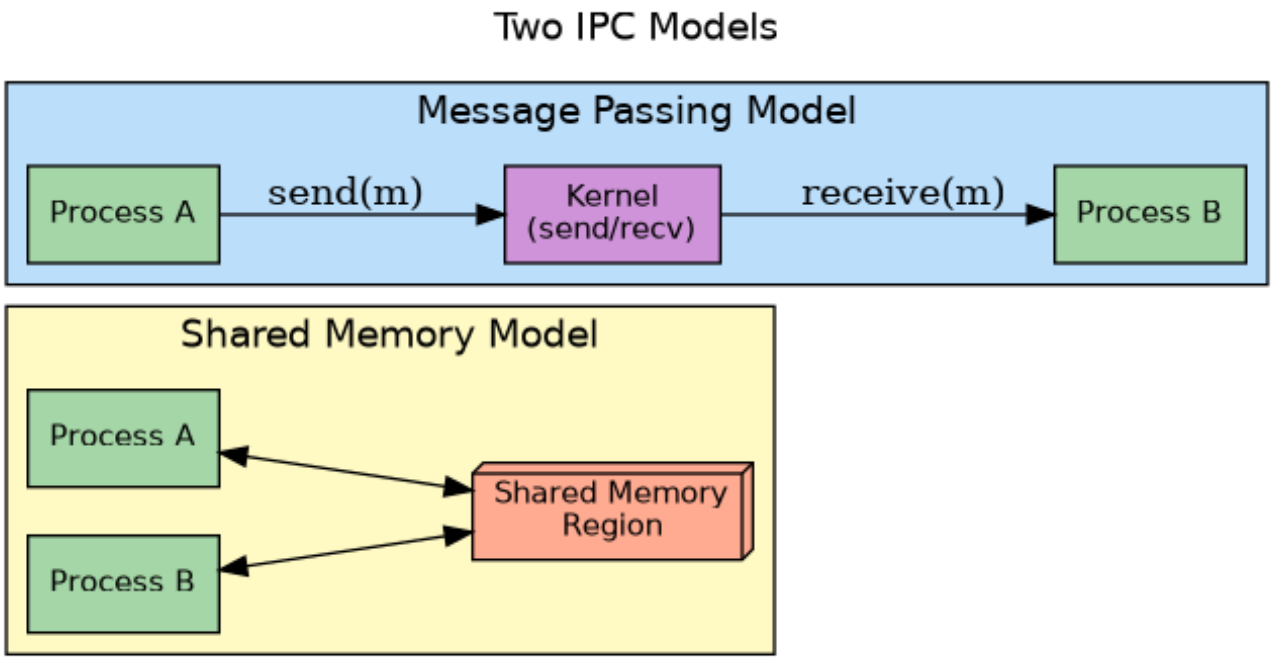

Figure 7: Shared memory vs message passing — the two IPC models.

The Two Models

Aspect	Shared Memory	Message Passing
How	Both processes map a common region of memory	Send/receive messages via the kernel
Speed	Faster (no kernel for each access)	Slower (each msg crosses user/kernel boundary)
Synchronization	Required explicitly (semaphores, mutexes)	Built into send/receive
Best for	High-throughput data sharing	Distributed/networked systems, simple coordination
Example	POSIX shm, mmap	pipes, sockets, message queues

Concrete IPC Schemes (mention these for a 7-marker)

Scheme	What it is
Pipes	One-way byte stream between related processes (parent/child); `read` end + `write` end
Named pipes (FIFOs)	Pipes with a name in the file system; usable by unrelated processes
Message queues	Kernel-managed queue of discrete messages; processes can prioritize and tag
Shared memory	Memory region attached to multiple processes' address spaces
Sockets	Endpoint for network or local IPC; TCP/UDP/UNIX-domain
Signals	Async notifications (`SIGINT`, `SIGTERM`, etc.)
Semaphores	Synchronization primitive — pure signalling, not data transfer

Send & Receive — direct vs indirect

Method	How processes identify each other
Direct	`send(P, msg)` / `receive(Q, msg)` — name the partner
Indirect	`send(mailbox, msg)` / `receive(mailbox, msg)` — name a mailbox/port

Blocking vs non-blocking

- **Blocking send** — sender waits till message received.
- **Non-blocking send** — sender continues immediately.
- **Blocking receive** — receiver waits till message arrives.
- **Non-blocking receive** — receiver gets msg or null immediately.

Common traps

- **Don't say message passing has no synchronization needed.** It does — just hidden inside send/receive.
- Shared memory is **faster** but **needs explicit sync**.
- Pipes are **one-directional**. For two-way, use two pipes or a socket.

Linked PYQs

- **Paper 1, Q11(a)** — "Explain in detail about the Inter Process Communication Models and Schemes" (7 marks).

3.9 Monolithic vs Microkernel

Probability: HIGH · Marks weight: 7

The Idea

How you build the OS itself is an architectural choice. Two famous styles:

- **Monolithic** — one big binary in privileged mode containing everything.
- **Microkernel** — a tiny privileged core (only IPC, scheduling, basic memory) plus all other services running as **user-space servers**.

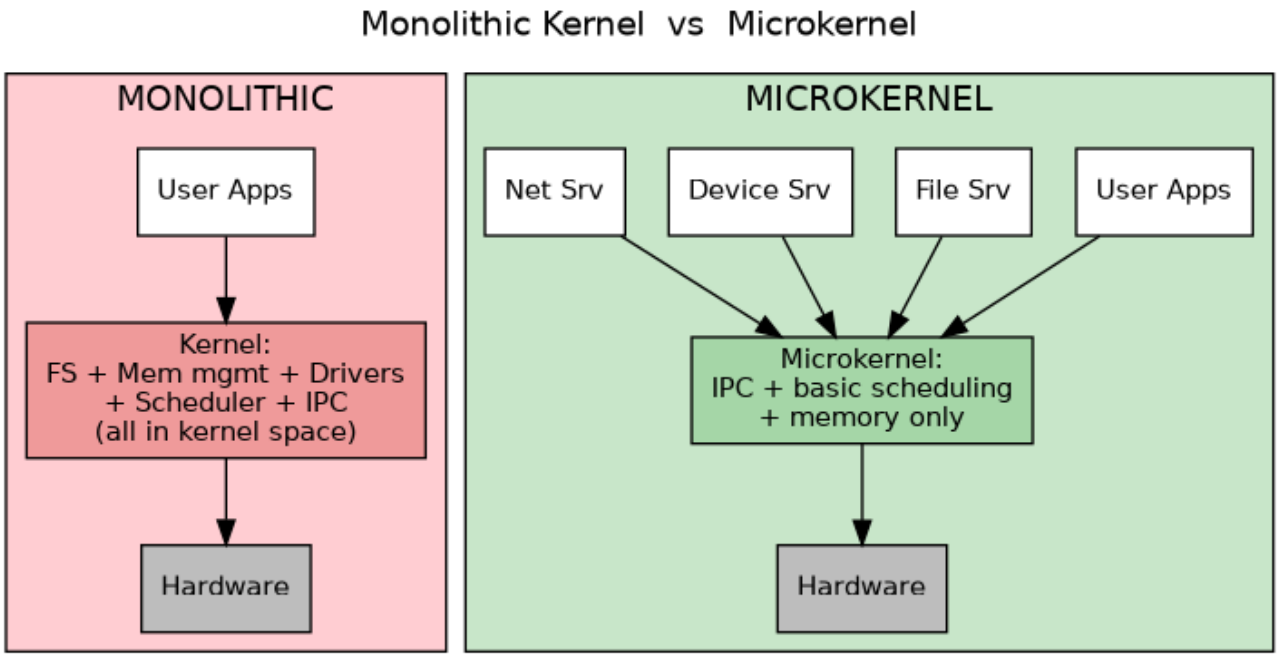

Figure 8: Both have user apps and hardware; the difference is what runs in kernel space.

Side-by-side

Aspect	Monolithic	Microkernel
Where services run	All in kernel space	Only IPC + minimal stuff in kernel; everything else in user space
Performance	Fast (no IPC for kernel ops)	Slower (lots of IPC between user-space servers)
Reliability	One driver crash → kernel panic	One driver crash → only that service dies; kernel survives
Extensibility	Recompile / reload kernel modules	Just start a new server in user space
Size	Large (often millions of lines)	Tiny core (sometimes a few thousand lines)

Aspect	Monolithic	Microkernel
Examples	Linux, traditional Unix, MS-DOS	Mach, MINIX 3, QNX, L4, GNU Hurd
Security	Larger attack surface	Smaller TCB (trusted computing base)

Exam-ready paragraph

> "A **monolithic kernel** runs the entire operating system — file systems, drivers, network stack, scheduler — in a single privileged address space. This gives high performance because there's no IPC between kernel services, but a fault in any module can crash the whole system. Linux is monolithic but uses **loadable modules** to add functionality at runtime. A **microkernel** keeps only the bare essentials — IPC, scheduling, basic memory management — in kernel space, while every other service runs as a user-space server. This improves modularity, reliability, and security at the cost of extra IPC overhead. QNX and MINIX 3 are well-known microkernels."

Common traps

- Modern Linux is **monolithic**, not microkernel — it uses modules but kernel-space.
- Windows NT is **hybrid** — mostly monolithic with some microkernel ideas.
- Mention that microkernels trade **performance** for **reliability and modularity**.

Linked PYQs

- **Paper 1, Q11(b)** — "Monolithic and Microkernel Systems" (7 marks).

3.10 Producer–Consumer Problem & Semaphore Solution

Probability: HIGH · Marks weight: 7

The Idea

A factory worker (the **producer**) makes widgets and drops them on a conveyor (the **buffer**). A packer (the **consumer**) picks widgets off the conveyor. The conveyor has a fixed length. Two rules:

- The producer must wait if the conveyor is **full**.
- The consumer must wait if the conveyor is **empty**.

This is the classic **producer-consumer problem** (a.k.a. bounded buffer). It teaches us how semaphores enforce both **mutual exclusion** and **synchronization**.

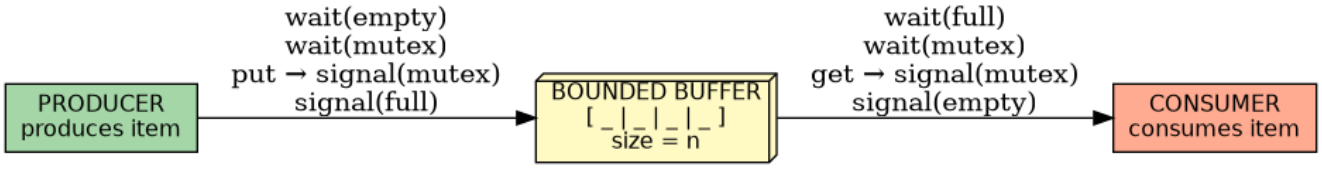

Figure 9: Bounded buffer with semaphore-controlled producer and consumer.

The three semaphores

Semaphore	Init	Purpose
`empty`	n	Counts empty slots in the buffer
`full`	0	Counts filled slots
`mutex`	1	Guards the buffer (mutual exclusion)

The code

```
semaphore empty = n, full = 0, mutex = 1;
item buffer[n];

PRODUCER:
while (true) {
   produce_item(item);
   wait(empty);
   wait(mutex);
   buffer[in] = item;
   in = (in + 1) % n;
   signal(mutex);
   signal(full);
}
```

```
CONSUMER:
while (true) {
   wait(full);
   wait(mutex);
   item = buffer[out];
   out = (out + 1) % n;
   signal(mutex);
   signal(empty);
   consume_item(item);
}
```

Why this works

- `wait(empty)` makes the producer block when buffer is full.

- `wait(full)` makes the consumer block when buffer is empty.
- `mutex` guarantees that producer and consumer never modify the buffer at the same time.

The bug to avoid

Swapping the order to `wait(mutex)` before `wait(empty)` deadlocks: if the buffer is full, the producer holds mutex while waiting for empty, and the consumer can't enter to free space.

Common traps

- **Order matters.** Always `wait(empty)` then `wait(mutex)` (not the reverse). Same logic for the consumer.
- The semaphore `mutex` is a **binary semaphore** (acts as a mutex). The other two are **counting semaphores**.
- Some texts call this the "bounded buffer problem". Same thing.

Linked PYQs

- **Paper 1, Q12(a)** — "Producer–Consumer Problem and its solution using semaphore" (7 marks).

3.11 Deadlock System Model & Characterization

Probability: HIGH · Marks weight: 9

The Idea

A deadlock happens when a set of processes are stuck waiting for resources that other deadlocked processes hold. Nobody moves. Real-world analogy: two cars meet on a one-lane bridge from opposite ends, both refuse to back up. Frozen.

The **system model** is the way we represent processes and resources so we can reason about deadlock. The **characterization** identifies the conditions under which it occurs.

The System Model

The OS has n processes $\{P_1, P_2, \ldots, P_n\}$ and m resource types $\{R_1, \ldots, R_m\}$. Each resource type has some number of identical instances. A process must follow this sequence to use a resource:

1. **Request** — ask the OS for the resource. If unavailable, wait.
2. **Use** — operate on the resource.
3. **Release** — return the resource to the OS.

The Four Coffman Conditions

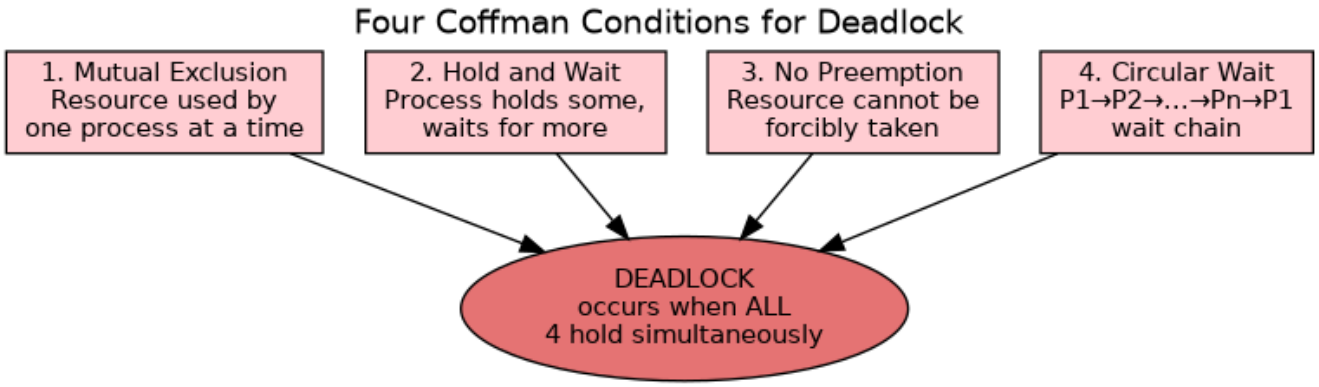

Figure 10: All four must hold for deadlock to occur. Break one → no deadlock.

#	Condition	Meaning
1	**Mutual Exclusion**	At least one resource must be held in non-shareable mode
2	**Hold and Wait**	A process holding some resources can request additional ones
3	**No Preemption**	Resources cannot be forcibly taken; only released voluntarily

#	Condition	Meaning
4	**Circular Wait**	A circular chain $P_1 \to P_2 \to \ldots \to P_n \to P_1$ exists, where each P_i waits for a resource held by the next

Resource Allocation Graph (RAG)

A directed graph: round nodes = processes, square nodes = resources (with dots for instances). An edge $P_i \to R_j$ means **request**; $R_j \to P_i$ means **assignment**.

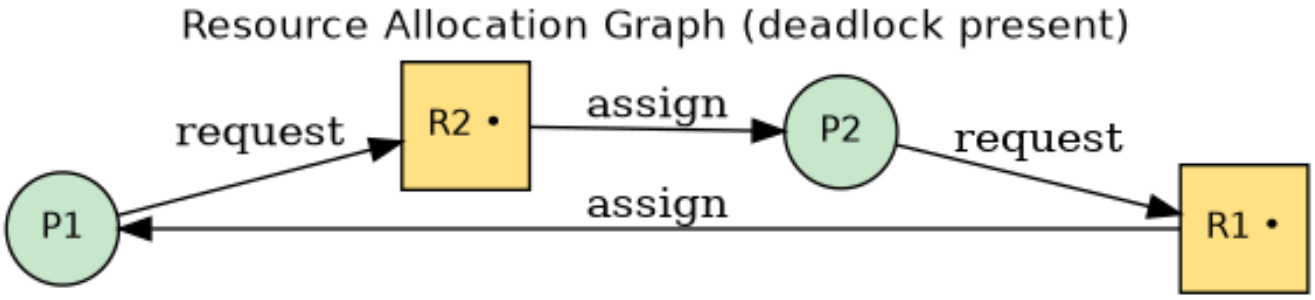

Figure 11: A simple deadlocked RAG: each process holds one resource and requests the other.

Rule: - If there's no cycle → no deadlock. - If a cycle exists **and each resource type has exactly one instance** → deadlock. - If a cycle exists with multiple instances → deadlock *may or may not* exist; further analysis (banker's algorithm) needed.

Four ways to handle deadlock

Approach	What it does	Cost
Prevention	Ensure at least one of the four conditions never holds	Restrictive; reduces throughput
Avoidance	Use Banker's algorithm to grant only safe requests	Needs Max declaration in advance
Detection & Recovery	Allow deadlock, periodically detect, then kill or rollback	Periodic overhead
Ignore (Ostrich)	Pretend it doesn't happen	Cheap; used by UNIX/Windows in practice

Methods for Deadlock Prevention (per condition)

Condition Broken	How
Mutual exclusion	Make resources shareable (not always possible)

Condition Broken	How
Hold and wait	Require process to request **all** needed resources upfront, OR release current resources before requesting more
No preemption	Allow OS to take resources from waiting processes
Circular wait	Impose a total ordering on resources; processes request only in increasing order

Common traps

- **All four conditions must hold simultaneously** — break any one and there's no deadlock.
- A cycle in RAG with **multiple instances per resource type** is necessary but not sufficient.
- Deadlock ≠ starvation. Starvation = indefinite waiting due to scheduling; deadlock = waiting due to held resources.

Linked PYQs

- **Paper 2, Q6** — "Deadlock system model and Deadlock characterization" (9 marks).

3.12 External vs Internal Fragmentation, and Paging

Probability: HIGH · Marks weight: 7

The Idea

When the OS allocates memory, two kinds of "wasted space" arise:

- **Internal fragmentation** — wasted space **inside** a partition that was allocated. Example: you ask for 1.5 KB, OS gives you a 2 KB block. The extra 0.5 KB is wasted (you got it but don't use it).
- **External fragmentation** — wasted space **outside** any allocation, scattered as small unusable holes between allocated regions.

Comparison

	Internal	External
Where wasted	Inside an allocated partition	Between allocated partitions (holes)
Cause	Fixed-size partitions / power-of-two allocators	Variable-size allocations + deallocations over time
Fix	Smaller partitions (more overhead)	Compaction (expensive) or paging
Example	4 KB page used for 3 KB process → 1 KB wasted	Free holes 100B + 200B + 50B totaling 350B but request for 300B fails

How paging fixes external fragmentation

Paging divides physical memory into fixed-size **frames** and a process's logical memory into equally sized **pages**. Any page can go in any frame. There's no requirement for contiguity, so external fragmentation **vanishes entirely**.

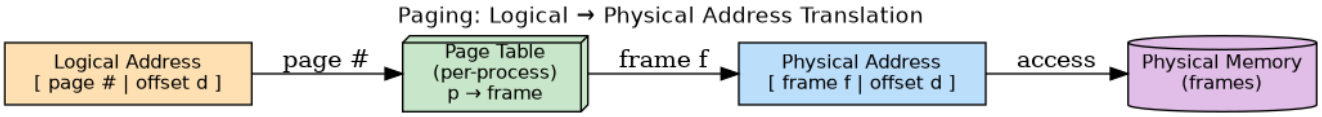

Figure 12: Logical address split into page # and offset; page table maps page # → frame #.

Address translation (formally)

A logical address has two parts: p (page number) and d (offset within the page). The page table gives the frame number f for p. The physical address is $f \cdot \text{page\_size} + d$.

If page size = 2^n bytes and logical address is m bits, then:

- Lowest n bits = offset d
- Upper $m - n$ bits = page number p

Paging still has internal fragmentation

If a process is 7.5 KB and page size is 4 KB, it needs 2 pages = 8 KB; 0.5 KB wasted in the last page. **Average internal fragmentation = page_size / 2.**

Exam-ready paragraph

> "**Internal fragmentation** is wasted memory inside an allocated block — for example, when a process needs 9 KB but is given a 16 KB partition. **External fragmentation** is wasted memory scattered between allocated regions as small unusable holes. Paging eliminates external fragmentation by dividing memory into equal-sized frames so any logical page can be placed in any free frame — contiguity is no longer required. However, paging still suffers from internal fragmentation when the last page of a process is not full. The trade-off is acceptable because the saved external fragmentation is far larger and the page table provides flexible mapping."

Common traps

- Paging fixes external, not internal.
- A common confusion is calling internal frag "fragmentation inside a page" — clarify: it's inside any *allocation unit*, page or partition.

Linked PYQs

- **Paper 2, Q12(a)** — "Difference between External and Internal fragmentation. How to solve fragmentation problem using paging?" (7 marks).

3.13 Thrashing

Probability: HIGH · Marks weight: 7

The Idea

Imagine you're juggling 10 plates with only two hands. You keep dropping one to catch another. The harder you try to "keep all in the air," the worse you do. That's thrashing.

In an OS: too many processes are loaded, none has enough frames to hold its working set, so every process keeps page-faulting, the CPU sits idle waiting for disk, the OS interprets that as "CPU is idle, must need more processes," loads even more processes — and the system grinds to a halt **doing only paging**.

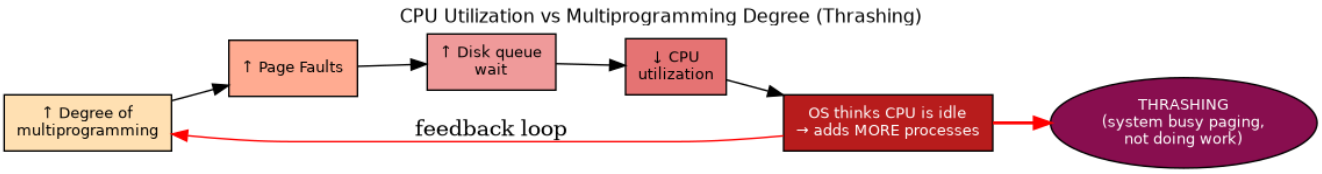

Figure 13: The vicious feedback loop that produces thrashing.

Causes

- Multiprogramming degree too high.
- Sum of working sets > total frames.
- The OS scheduler keeps loading more processes when CPU utilization drops.

How the system detects it

- **CPU utilization drops sharply** while paging activity skyrockets.
- **Page fault frequency (PFF)** exceeds a threshold.
- Average disk queue length increases dramatically.

Two ways to eliminate it

1. Working Set Model. The working set of a process at time t is the set of pages referenced in the last Δ references. The OS sums working-set sizes; if total > available frames, **suspend** some processes. The suspended ones are swapped out to disk until enough memory frees up.

2. Page Fault Frequency (PFF) Control. Monitor PFF for each process. If PFF > upper bound → process needs more frames; give it some. If PFF < lower bound → process has more than enough; take frames away. If no frames available globally → suspend a process.

Exam-ready paragraph

> "Thrashing occurs when a system spends more time swapping pages than executing useful work. The root cause is over-commitment: the multiprogramming degree is so high that no process has enough frames for its working set, leading to constant page faults. The CPU drops to low utilization; the OS misinterprets

this as needing more processes and loads more, worsening the problem. Detection is via CPU-utilization monitoring or page-fault-frequency tracking. The fix is either the **working set model** — which decides how many frames each process needs based on recent references and suspends some processes if total demand exceeds memory — or **page-fault-frequency control**, which adjusts each process's frame allocation up or down to keep PFF within an acceptable band."

Common traps

- Thrashing is **not** caused by slow disks alone; it's the combination of demand + frames.
- Working set's Δ window is a tunable parameter, not fixed.

Linked PYQs

- **Paper 2, Q12(b)** — "What is Thrashing? Cause? How does the system detect it? How to eliminate?" (7 marks).

3.14 Multiuser System and Multithreaded System

Probability: HIGH · Marks weight: 9

The Idea

Two often-confused concepts:

- **Multiuser system** — the OS allows **multiple human users** to use the same computer at the same time, each isolated from the others (think Unix server with many SSH sessions).
- **Multithreaded system** — a single **process** can have multiple **threads** of execution running concurrently within it, sharing its memory.

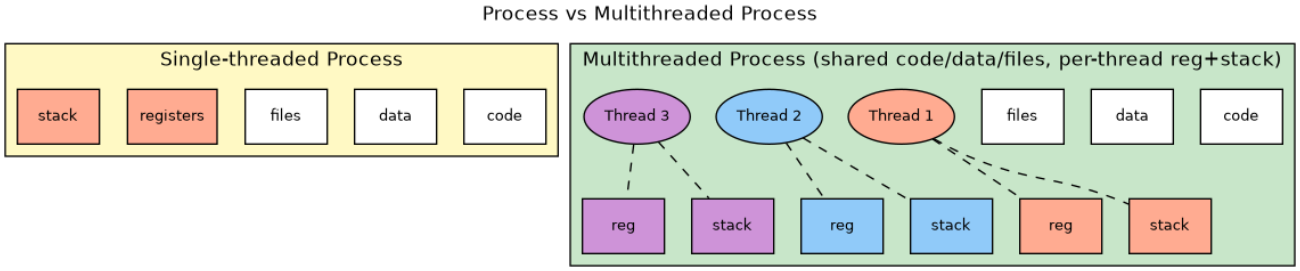

Figure 14: Single-thread vs multithreaded process — threads share code/data/files but have their own stack and registers.

Multiuser System

Feature	Description
User isolation	Each user has separate account, files, processes
Concurrent sessions	Many users active at once; OS time-shares
Resource sharing	CPU, memory, disk shared but accounted per user
Authentication	Each user logs in with credentials
Protection	File ownership, permissions, quotas
Examples	UNIX/Linux, mainframes, multi-session Windows Server

Multithreaded System

Feature	Description
Threads share	Code section, data section, open files
Threads have their own	Program counter, register set, stack
Benefit	Responsiveness, resource sharing, cheaper than processes, scales on multicore
Cost	Race conditions, synchronization complexity

Types of threads

Type	Where managed	Pros	Cons
User-level threads	In user space by a thread library	Fast switch, portable	Blocking syscall blocks whole process
Kernel-level threads	Directly by the OS kernel	True parallelism on multicore	More expensive to create/switch

Multithreading models

- **Many-to-One** — many user threads mapped to one kernel thread (older systems).
- **One-to-One** — each user thread mapped to one kernel thread (Windows, Linux pthreads).
- **Many-to-Many** — flexible: many user threads multiplexed onto a smaller pool of kernel threads.

Benefits of multithreading

1. **Responsiveness** — UI stays alive while background thread loads data.
2. **Resource sharing** — cheaper than full IPC between processes.
3. **Economy** — creating a thread is far cheaper than creating a process.
4. **Scalability** — different threads run on different cores in parallel.

Common traps

- A **multiuser** system **may or may not** support multithreading; the two are orthogonal.
- Threads share memory → race conditions; processes don't.
- "Multitasking" is a third related word — it just means running multiple processes; not the same as multiuser or multithreaded.

Linked PYQs

- **Paper 1, Q8** — "Multiuser system and Multithreaded System" (9 marks).

3.15 RAID Levels

Probability: HIGH · Marks weight: 7

The Idea

RAID = Redundant Array of Independent Disks. Combine several physical disks to act as one logical drive, gaining either **performance** (parallelism via striping), **reliability** (redundancy via mirroring or parity), or both.

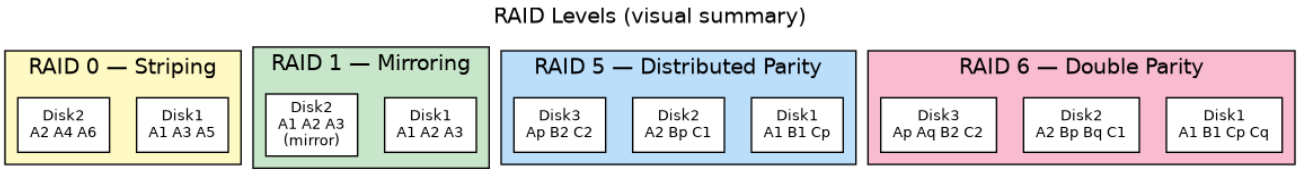

Figure 15: Visual layout of RAID 0, 1, 5, 6.

Key Characteristics

Characteristic	Meaning
Striping	Split data across disks for parallel access
Mirroring	Copy the same data on two or more disks
Parity	Compute a checksum (XOR) on a stripe; can reconstruct any one lost disk
Fault tolerance	How many disks can fail without data loss

The Major Levels

Level	Technique	Min disks	Fault tolerance	Use case
RAID 0	Striping only, no redundancy	2	None	Pure speed, throwaway data
RAID 1	Mirroring	2	One disk	Critical systems, small storage
RAID 2	Bit-level striping with Hamming code parity	3+	One bit	Rarely used today
RAID 3	Byte-level striping + dedicated parity disk	3+	One disk	Large sequential files
RAID 4	Block-level striping + dedicated parity disk	3+	One disk	Parity disk is a bottleneck

Level	Technique	Min disks	Fault tolerance	Use case
RAID 5	Block-level striping + **distributed parity**	3+	One disk	Best balance; most popular
RAID 6	Striping + **two** parity blocks	4+	Two disks	Higher reliability than RAID 5
RAID 10 (1+0)	Mirror first, then stripe across mirrors	4+	One per mirror pair	High performance + reliability

Trade-offs at a glance

Level	Read speed	Write speed	Capacity efficiency	Reliability
0	Very fast	Very fast	100%	Worse than single disk
1	Fast (parallel reads)	Slow (write to both)	50%	High
5	Fast	Slower (parity calc)	$(n-1)/n$	Survives 1 failure
6	Fast	Slowest of the parity ones	$(n-2)/n$	Survives 2 failures
10	Very fast	Fast	50%	Very high

Exam-ready paragraph

> "RAID combines multiple disks into a logical unit to improve performance, reliability, or both. **Striping** spreads data across drives so reads and writes happen in parallel; **mirroring** duplicates data on a second drive for redundancy; **parity** computes a checksum that allows reconstruction of any lost block. RAID 0 uses pure striping with no redundancy — fast but fragile. RAID 1 uses mirroring — reliable but only 50% capacity. RAID 5 stripes blocks across all disks and distributes a parity block across them; it tolerates one disk failure while keeping high capacity efficiency. RAID 6 adds a second independent parity block, tolerating two simultaneous failures. RAID 10 combines mirroring and striping for performance and reliability at the cost of 50% capacity."

Common traps

- RAID is **not a backup**. It protects against disk failure, not against deletion, corruption, or disaster.
- RAID 5 needs at least **3** disks. RAID 6 at least **4**.
- "Hot-swap" — the ability to replace a failed disk without downtime — is a feature of the array, not of any single RAID level.

Linked PYQs

- **Paper 1, Q13(b)** — “RAID and its characteristics. Various RAID levels” (7 marks).

3.16 Disk Storage & Disk Scheduling (theory)

Probability: HIGH · Marks weight: 9

The Idea

A hard disk is a stack of spinning platters. Each platter has two surfaces; each surface has many concentric **tracks**; each track is divided into **sectors**. A **cylinder** is the set of tracks at the same radius across all platters. The read/write head is positioned by an actuator arm.

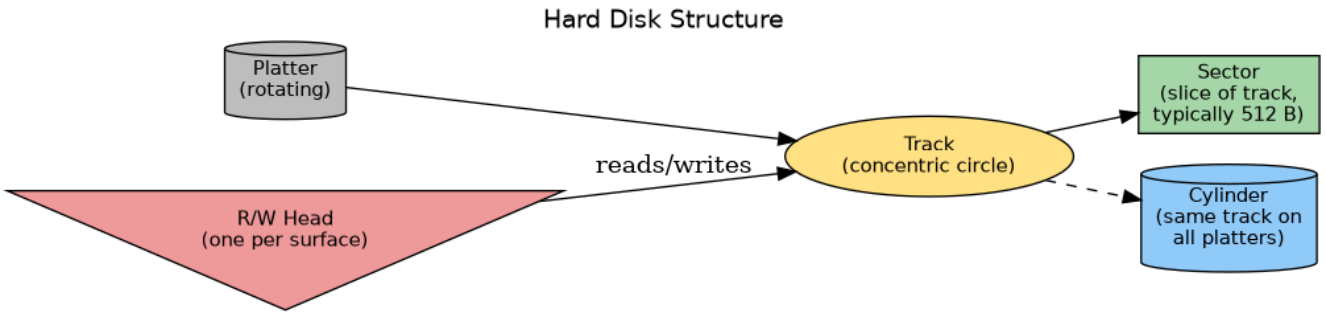

Figure 16: Anatomy of a hard disk: platter, track, sector, cylinder, head.

The four components of disk access time

Component	What it is	Typical
Seek time	Time for the arm to move to the right track	~5–10 ms
Rotational latency	Time for the right sector to rotate under the head	~half a rotation = ~4 ms at 7200 rpm
Transfer time	Time to actually move the bytes once positioned	depends on rate and size
Controller overhead	Bus and controller delay	tiny

$$\text{Disk access time} \approx \text{seek} + \text{rotational latency} + \text{transfer time}$$

Disk scheduling = minimize seek + latency

Since seek time dominates, the OS reorders pending requests to minimize total head movement. See §3.1 for the algorithms and worked examples.

Disk performance metrics

- **Bandwidth** — bytes transferred per unit time.
- **Throughput** — number of I/O operations per second (IOPS).
- **Mean response time** — average wait + service time per request.

Common traps

- “Latency” in OS exams almost always means **rotational latency** specifically, not network latency.
- Cylinder = set of tracks at the same radius (one per platter); a cylinder can be accessed without moving the head.

Linked PYQs

- **Paper 1, Q7** — “Disk storage and Disk scheduling” (9 marks). For 9 marks, combine the structure (this section) with one full worked SCAN/SSTF example.

3.17 Fixed vs Variable Partitioning

Probability: HIGH · Marks weight: ~5

The Idea

Old-school OSes divided memory into **partitions**. Two main schemes:

- **Fixed partitioning** — divide memory into a number of fixed-size partitions at boot time. Each holds one process. Simple but rigid.
- **Variable partitioning** — allocate exactly the amount each process needs when it starts. Smarter but harder to manage.

Comparison

Aspect	Fixed Partitioning	Variable Partitioning
Partition size	Fixed in advance	Dynamic; based on process need
Number of partitions	Fixed	Varies over time
Internal fragmentation	Yes (big partition + small process)	No (partition matches need)
External fragmentation	No	Yes (holes between processes)
Implementation	Simple	Needs free-list management
Memory utilization	Lower	Higher
Example	OS/360 MFT	OS/360 MVT, modern allocators with compaction

Placement strategies (used in variable partitioning)

Strategy	Rule
First Fit	Use the first hole big enough
Best Fit	Use the smallest hole big enough
Worst Fit	Use the largest hole available

First Fit and Best Fit beat Worst Fit in practice.

Linked PYQs

- **Paper 1, Q12(b)(ii)** — “Fixed portioning vs variable partitioning” (3.5 marks).
- **Paper 2, Q4** — “Multiprogramming with fixed partitions” (2 marks).

3.18 Multilevel Feedback Queue Scheduling

Probability: MEDIUM · Marks weight: ~3

The Idea

A multilevel queue scheduler classifies processes into different queues (e.g., interactive, batch) with different scheduling policies for each. **Multilevel feedback queue** (MLFQ) adds one extra ingredient: **processes can move between queues** based on their CPU usage pattern.

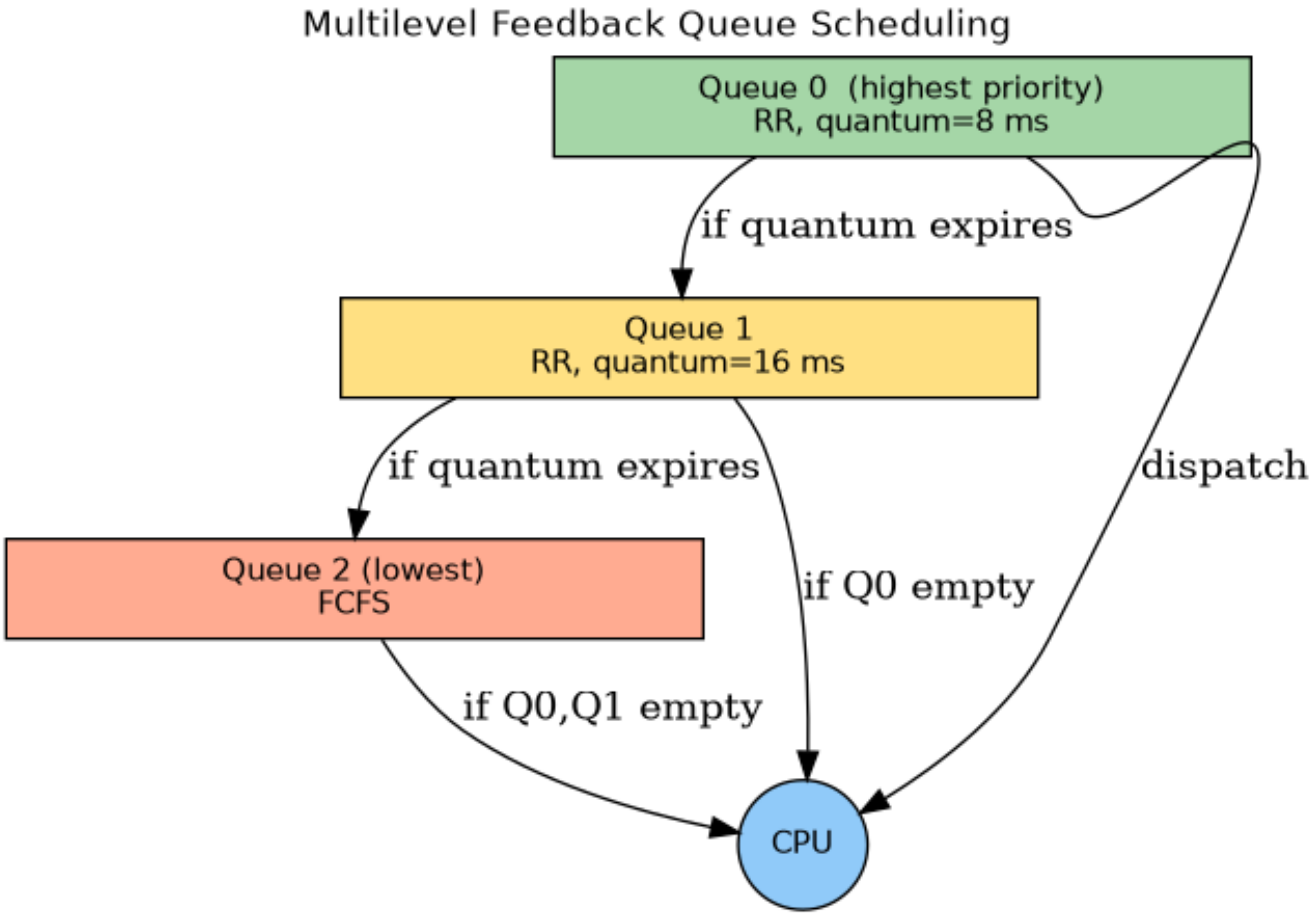

Figure 17: Three-level feedback queue: top is highest priority, bottom is lowest.

How it works

- A new process enters the **top queue** (highest priority).
- Top queues use small time quanta (e.g., RR with 8 ms).
- If a process uses its full quantum, it gets demoted to the next queue (longer quantum).
- If a process gives up CPU early (likely I/O bound), it stays or moves up.
- The bottom queue often uses FCFS.

Tunable parameters

1. Number of queues.
2. Scheduling policy per queue.
3. Rules for promotion (moving up).
4. Rules for demotion (moving down).
5. Rule for which queue a new process joins.

Why it’s clever

It **automatically learns** which processes are CPU-bound (sink to lower queues, longer quanta) and which are I/O-bound or interactive (stay on top, get quick responses).

Linked PYQs

- **Paper 1, Q12(b)(i)** — “Multilevel Feedback Queue Scheduling” (3.5 marks).

3.19 Short Section A Topics

These are the 2-mark warm-up questions. Memorize a tight 50-word answer for each.

3.19.1 Real-Time Operating System

A **real-time operating system** is one that must produce a response within a **strict, predictable deadline**. The correctness of the system depends not only on the logical result but also on the **time** at which it is produced. Two types: **hard real-time** (missing a deadline causes catastrophic failure, e.g., aircraft autopilot, pacemakers) and **soft real-time** (occasional misses are acceptable; quality degrades, e.g., video streaming). Used in embedded systems, robotics, industrial control.

3.19.2 Seek Time and Latency Time

Seek time is the time taken by the disk's read/write head to move from its current track to the requested track. **Latency time** (rotational latency) is the time required for the desired sector to rotate beneath the head once the head is on the correct track. Both contribute to the total disk access time, with seek time usually being dominant.

3.19.3 Why we need Scheduling

Multiple processes typically compete for a single CPU. Without **scheduling**, the system could not decide which process runs next, leading to unfair CPU usage, poor responsiveness, and possible starvation. Scheduling maximizes CPU utilization, balances throughput against turnaround and response time, and ensures fairness among processes.

3.19.4 I/O Buffering

I/O buffering is the use of a memory area (buffer) to temporarily hold data while it is being transferred between two devices (or between a device and an application). Buffering smooths out the speed mismatch between fast CPU/memory and slow I/O devices, allows overlapping of I/O and computation, and supports block-oriented operations. Three forms: **single buffer**, **double buffer**, and **circular buffer**.

3.19.5 Safe State and Unsafe State (already covered in §3.3)

A **safe state** is one in which the OS can satisfy every process's maximum resource demand in some order without deadlock. An **unsafe state** is one in which deadlock is **possible** (no guaranteed safe sequence exists), though not certain. Safe states avoid deadlock; unsafe states may or may not result in deadlock depending on subsequent allocations.

3.19.6 Concurrent Processes

Concurrent processes are processes whose executions overlap in time. They may run in true parallel on multiple CPUs, or interleaved on a single CPU via time-sharing. Concurrency introduces challenges of **synchronization** (race conditions on shared data), **mutual exclusion** (critical sections), **deadlock**, and **starvation**. Tools like semaphores, mutexes, and monitors are used to coordinate them.

3.19.7 Performance Criteria in CPU Scheduling

CPU schedulers are evaluated on five primary criteria: **CPU utilization** (keep the CPU busy — maximize), **throughput** (jobs completed per unit time — maximize), **turnaround time** (total time from submission to completion — minimize), **waiting time** (time spent in ready queue — minimize), and **response time** (time from submission to first CPU allocation — minimize for interactive systems). Different algorithms trade these off.

3.19.8 Multiprogramming with Fixed Partitions

In **multiprogramming with fixed partitions**, main memory is divided into a fixed number of partitions of (possibly equal or unequal) fixed size at boot time. Each partition can hold exactly one process. The OS maintains a queue of waiting processes per partition (or a single queue). The scheme is simple but suffers from internal fragmentation, since a small process in a large partition wastes the leftover space.

Part 4 — Complete PYQ Solutions

Below: every question from both attached papers, fully solved in the **exact style and depth you should reproduce in the answer book**.

4.1 Paper 1 — TU-830(A), May 2025

Section A (2 marks each, ≤ 75 words)

Q1. Briefly define the term Real Time O.S.

A **Real Time Operating System (RTOS)** is an OS designed to process events and produce responses within strict, predictable time deadlines. Correctness depends on both the logical result **and** the time at which it is delivered. RTOSes are categorized as **hard real-time** — where missing a deadline causes system failure (avionics, pacemakers) — and **soft real-time** — where occasional misses degrade quality but don't fail (multimedia, gaming). Examples: VxWorks, QNX, FreeRTOS, RTLinux.

Takeaway: "Hard vs soft real-time" is what earns the second mark.

Q2. Define the Seek Time and Latency Time.

Seek time is the time required for the disk's read/write head to move from its current track to the requested track. It is typically the largest component of disk access time (~5–10 ms on traditional drives).

Latency time — also called **rotational latency** — is the time for the desired sector to rotate beneath the read/write head once it has reached the correct track. On average, this is half a full rotation; at 7200 rpm, ~4.2 ms.

Total disk access time ≈ seek + latency + transfer time.

Takeaway: Give both definitions, mention typical magnitudes for the second mark.

Q3. What do we need Scheduling?

Multiple processes typically need the CPU simultaneously, but only one process can use it at a time. **Scheduling** is required to decide which process gets the CPU next, ensuring high CPU utilization, fair allocation, low waiting time, and good response time for interactive processes. Without scheduling, fairness, throughput, and responsiveness would all collapse. CPU scheduling occurs at three levels — long-term, mid-term, and short-term — each with different goals.

Takeaway: Mention purpose (fairness, utilization, throughput) and the three levels.

Q4. What do you mean by the I/O Buffering?

I/O buffering is the use of a memory region (a buffer) to temporarily hold data being transferred between two devices, or between a device and an application. Buffering smooths out the speed difference between fast CPU/memory and slow I/O devices, allows overlap of computation and I/O, and supports block-oriented

operations. Three common schemes are **single buffering** (one buffer per device), **double buffering** (two buffers used alternately), and **circular buffering** (a ring of buffers for streaming).

Takeaway: Reason + three types = full 2 marks.

Q5. What do you mean by Safe State and Unsafe State?

A **safe state** is a system state in which the OS can satisfy every process's maximum resource demand in **some order** without producing deadlock — that is, a safe sequence $\langle P_1, P_2, \ldots, P_n \rangle$ exists.

An **unsafe state** is one where no such safe sequence is guaranteed. Note that an unsafe state is **not** necessarily a deadlocked state — deadlock is **possible** but not certain. The Banker's Algorithm uses safety checking to keep the system in a safe state.

Takeaway: Make clear that unsafe ≠ deadlocked.

Section B (9 marks each, attempt any 2 of 3)

Q6. Explain in detail about File System Protection and Security. (9)

File system protection is the operating system mechanism that controls who is allowed to perform which operations on each file. The basic protected operations are **read, write, execute, append, delete, and list**. The OS enforces protection through one of several schemes.

The most common is the **access control list (ACL)** — each file stores a list of (user, allowed-operations) entries. ACLs are expressive but can grow long. UNIX uses a more compact scheme: each file has three classes — **owner, group, others** — and three bits per class — **read, write, execute** — totalling **nine permission bits** (rwxrwxrwx). The **capability list** model is the dual: each user holds a list of (object, rights) tokens.

File system security extends protection against external threats. Major threats include **viruses** (code that attaches itself to programs and spreads), **worms** (network-replicating programs), **trojan horses** (malicious code hidden inside trusted programs), **trap doors** (developer backdoors), **denial-of-service** attacks, and **buffer overflow** exploits.

Defences combine multiple layers. **Authentication** — verifying identity via passwords, smart cards, or biometrics — is the first line; **multi-factor authentication** combines two of these. **Encryption** protects file contents at rest (full-disk encryption) and in transit (TLS). **Auditing** logs all sensitive operations. **Sandboxing** confines untrusted code. **Address space layout randomization (ASLR)** and the **NX bit** harden the kernel against exploitation. Finally, the **principle of least privilege** ensures each subject receives only the rights it strictly needs.

Together, protection (internal mechanism) and security (defence against external threats) safeguard file system data, integrity, and confidentiality.

Takeaway: ACL + Unix bits + threat list + defences = full marks.

Q7. Explain in detail about Disk Storage and Disk Scheduling. (9)

A hard disk consists of one or more **platters** that rotate at high speed (5400–15000 rpm). Each platter surface is divided into concentric circles called **tracks**; each track is split into **sectors** (usually 512 bytes). The set of all tracks at the same radius across every platter is a **cylinder**. A read/write head is mounted on an arm; together the heads are positioned by an actuator that moves them all in unison across the disk.

Disk access time has three main components:

1. **Seek time** — moving the head to the right track (largest).
2. **Rotational latency** — waiting for the right sector to spin under the head.
3. **Transfer time** — actually moving bytes through the head.

Because seek time dominates, the OS uses **disk scheduling** to reorder the queue of pending requests and minimize total head movement.

Common algorithms:

- **FCFS (First Come First Serve)** — serve requests in arrival order. Simple, fair, but slow.
- **SSTF (Shortest Seek Time First)** — always serve the closest pending request. Low seek time but can starve far requests.
- **SCAN (Elevator)** — the head moves in one direction servicing all requests until it reaches the disk end, then reverses and services in the other direction.
- **C-SCAN (Circular SCAN)** — like SCAN, but after reaching the end, jumps back to the start without serving requests on the return; gives more uniform wait times.
- **LOOK** — like SCAN but reverses at the **last request** in that direction, not the disk end. Avoids unnecessary travel.
- **C-LOOK** — the LOOK variant of C-SCAN.

Example (head at 49, requests {45, 20, 90, 10, 50, 60, 80, 25, 70}):

Algorithm	Order	Total head movement
FCFS	49→45→20→90→10→50→60→80→25→70	280
SSTF	49→50→45→60→70→80→90→25→20→10	131
SCAN (up first)	49→50→60→70→80→90→99→45→25→20→10	139
LOOK (up first)	49→50→60→70→80→90→45→25→20→10	121

Disk scheduling choice depends on workload: SSTF minimizes average seek for random workloads; SCAN/LOOK give fairer, more uniform wait times.

Takeaway: Structure (platter/track/sector) + access components + ≥3 algorithms + worked numbers = full marks.

Q8. Explain in detail about Multiuser System and Multithreaded System. (9)

A **multiuser operating system** allows several human users to access the same physical computer concurrently, each in their own isolated environment. The system maintains a separate account, file space, and process tree per user. Authentication (login) verifies identity; the OS uses **time-sharing** to interleave each user's processes on the CPU; file ownership and permission bits enforce isolation. UNIX/Linux servers, mainframe systems, and Windows Server with Remote Desktop are classic examples. Benefits: shared hardware investment, centralized administration, ability to support many users simultaneously. Costs: scheduling complexity, security overhead, and need for fair resource allocation.

A **multithreaded operating system** allows a single process to contain multiple **threads** of execution that share the same address space. Each thread has its own **program counter, register set, and stack**, but shares the process's **code, data, open files, and signals**. Threads are sometimes called **lightweight processes**.

Benefits of multithreading:

1. **Responsiveness** — a UI thread stays alive even while a worker thread does I/O.

2. **Resource sharing** — threads share memory; communication is much cheaper than IPC between processes.
3. **Economy** — creating a thread costs ~100× less than creating a process.
4. **Scalability** — on multicore CPUs, different threads can truly run in parallel.

Thread implementations:

- **User-level threads** — managed by a user-space library; the kernel sees one process. Fast switching, but a blocking syscall blocks the whole process.
- **Kernel-level threads** — managed directly by the kernel. Slower to create but allow true parallelism and don't block on syscalls.

Multithreading models describe how user threads map to kernel threads: **many-to-one**, **one-to-one** (Linux, Windows), and **many-to-many**.

Multiuser and multithreading are **orthogonal concepts**: a system can be both (Linux), one (a single-user multithreaded laptop), the other (an old multiuser mainframe with no threads), or neither.

Takeaway: Define both, contrast, and list at least three benefits of multithreading.

Section C (14 marks each, attempt any 3 of 5)

Q9. (a) Write short notes on (7 marks total):

(i) File System Protection and Security

See full answer in Q6. The 3.5-mark version: define both terms; mention ACL + Unix-permission-bits for protection; mention viruses/worms/trojans and authentication/encryption for security.

(ii) Linked File Allocation Methods

In **linked allocation**, each file is stored as a linked list of disk blocks scattered across the disk; each block contains data plus a pointer to the next block. The directory entry stores the start block (and optionally the end block) of the file. No external fragmentation occurs because blocks are non-contiguous; files can grow easily. Disadvantages: **slow random access** — to read block n you must traverse $n-1$ blocks; some space in each block is consumed by the next-pointer; a single corrupted pointer can lose the rest of the file. The **File Allocation Table (FAT)** is a popular variant: all pointers are kept in a single in-memory table, making traversal much faster while preserving non-contiguity benefits. Used in MS-DOS and flash drives.

Q9. (b) Explain in detail about the Dining Philosopher Problem. (7)

The **Dining Philosopher Problem** is a classical synchronization problem proposed by Dijkstra. Five philosophers sit around a circular table; each alternates between thinking and eating. Between each pair of adjacent philosophers there is a single fork (or chopstick), giving five forks in total. To eat, a philosopher must acquire **both the left and right forks** simultaneously. Since each fork is shared between two philosophers, the problem captures the essence of **concurrent access to shared resources**.

The naive semaphore solution declares a binary semaphore `fork[5]`, all initialised to 1:

```
philosopher_i():
  while (true) {
    think();
    wait(fork[i]);              // pick up left
    wait(fork[(i+1) % 5]);      // pick up right
    eat();
    signal(fork[i]);
    signal(fork[(i+1) % 5]);
  }
```

This solution is **incorrect**: if all five philosophers simultaneously pick up their left fork, every right `wait` will block forever — a deadlock with all four Coffman conditions present.

Standard fixes:

1. **Limit the table to N − 1 philosophers** using a semaphore `room = 4`. With at most four seated, at least one fork pair must be free, breaking hold-and-wait.
2. **Asymmetric pickup** — odd-numbered philosophers pick up the right fork first, then the left; even-numbered philosophers do the opposite. This breaks the circular-wait condition.
3. **Atomic pickup of both forks** — wrap the two `wait` calls in another mutex so a philosopher either grabs both at once or grabs neither. This prevents deadlock but reduces concurrency.

The problem teaches three lessons: (a) **deadlock can arise even with simple, symmetric code**; (b) **breaking one of the four Coffman conditions** is sufficient to eliminate it; and (c) **starvation-freedom is harder than deadlock-freedom** — a complete solution should also ensure no philosopher waits indefinitely.

Takeaway: Statement + naive code + bug + fix + lesson — that's the 7-mark structure.

Q10. (a) Explain the concept of File concept. Define File organization and access mechanism. (7)

A **file** is an abstraction for a named collection of related information stored on secondary storage. To the OS, a file is the smallest unit of logical storage that users can create, delete, read, write, or share. The OS hides the underlying physical organization (sectors, blocks, cylinders) and gives applications a clean interface.

File attributes include name, identifier, type, location on storage, size, protection bits, owner, and timestamps.

File operations include `create`, `open`, `read`, `write`, `seek (reposition)`, `close`, `delete`, `truncate`, and append.

File organization describes how the records (or bytes) of a file are physically laid out on disk:

Organization	Description
Sequential	Records stored one after another in order; simple, fast bulk read, slow random access.
Indexed	An index block contains pointers to data records; fast random access at the cost of index overhead.
Indexed-Sequential	Sequential file augmented with a sparse index — compromise for both bulk and random access.
Hashed	A hash function maps key → block; constant-time access, but collisions to manage and harder to grow.

Access methods describe how a program reads or writes data:

1. **Sequential access** — bytes/records read in order, with an implicit position pointer that advances after each operation. Used by editors, compilers.
2. **Direct (random) access** — file viewed as numbered fixed-size blocks; `read n` retrieves block n directly. Used by databases.
3. **Indexed access** — an index file points to data blocks; to read a record, look up the index, then fetch the data block. Common in large database files.

Organization (storage layout) and access method (program interface) are related but distinct: a sequentially organized file may still be accessed randomly via byte offset, though efficiency may suffer.

Q10. (b) Page references: 0, 9, 0, 1, 8, 1, 8, 7, 8, 7, 1, 2, 8, 2, 7, 8, 2, 3, 8, 3 with three frames. Compute page faults under FIFO, LRU, Optimal. (7)

Reference string (length 20): 0, 9, 0, 1, 8, 1, 8, 7, 8, 7, 1, 2, 8, 2, 7, 8, 2, 3, 8, 3.

Three frames available.

(i) FIFO replacement. Evict the page that has been in memory the longest.

Ref	Frames (oldest → newest)	Hit/Fault
0	[0]	F
9	[0,9]	F
0	[0,9]	H
1	[0,9,1]	F
8	[9,1,8]	F (evict 0)
1	[9,1,8]	H
8	[9,1,8]	H
7	[1,8,7]	F (evict 9)
8	[1,8,7]	H
7	[1,8,7]	H
1	[1,8,7]	H
2	[8,7,2]	F (evict 1)
8	[8,7,2]	H
2	[8,7,2]	H
7	[8,7,2]	H
8	[8,7,2]	H
2	[8,7,2]	H
3	[7,2,3]	F (evict 8)
8	[2,3,8]	F (evict 7)
3	[2,3,8]	H

FIFO page faults = 8

(ii) LRU replacement. Evict the page that has been unused for the longest time.

Ref	Frames (LRU → MRU)	Hit/Fault
0	[0]	F
9	[0,9]	F
0	[9,0]	H
1	[9,0,1]	F
8	[0,1,8]	F (evict 9)
1	[0,8,1]	H
8	[0,1,8]	H
7	[1,8,7]	F (evict 0)
8	[1,7,8]	H
7	[1,8,7]	H
1	[8,7,1]	H
2	[7,1,2]	F (evict 8)
8	[1,2,8]	F (evict 7)
2	[1,8,2]	H
7	[8,2,7]	F (evict 1)
8	[2,7,8]	H
2	[7,8,2]	H
3	[8,2,3]	F (evict 7)
8	[2,3,8]	H
3	[2,8,3]	H

LRU page faults = 9

(iii) Optimal replacement. Evict the page whose next reference is furthest in the future (or absent).

Ref	Frames	Note	Hit/Fault
0	[0]		F
9	[0,9]		F
0	[0,9]		H
1	[0,9,1]		F
8	[8,9,1]	evict 0 (future: never)	F
1	[8,9,1]		H
8	[8,9,1]		H
7	[8,7,1]	evict 9 (future: never)	F
8	[8,7,1]		H
7	[8,7,1]		H
1	[8,7,1]		H
2	[8,7,2]	evict 1 (future: never again)	F
8	[8,7,2]		H
2	[8,7,2]		H
7	[8,7,2]		H
8	[8,7,2]		H
2	[8,7,2]		H
3	[8,2,3]	evict 7 (future: never)	F
8	[8,2,3]		H
3	[8,2,3]		H

Optimal page faults = 7

Summary:

Algorithm	Page Faults
FIFO	8
LRU	9
Optimal	7

Observation: Optimal is the theoretical minimum (it requires knowledge of the future). FIFO and LRU are achievable; either can win depending on the reference pattern.

Q11. (a) Explain in detail about the Inter Process Communication Models and Schemes. (7)

Inter-Process Communication (IPC) is the set of mechanisms an OS provides for processes to exchange information and synchronize their execution. Because each process runs in its own address space, special primitives are needed.

Two fundamental models:

1. **Shared Memory Model.** Two or more processes attach a common memory region to their address spaces. Once attached, reads and writes are ordinary memory accesses — very fast. But processes must add their own synchronization (semaphores, mutexes) to prevent race conditions. Suited to high-throughput data sharing.
2. **Message Passing Model.** Processes exchange discrete messages through the kernel using `send(message)` and `receive(message)` primitives. Slower (each call crosses the user/kernel boundary) but synchronization is built in. Suited to distributed and loosely-coupled systems.

IPC schemes (concrete implementations):

Scheme	Description
Pipes	A unidirectional byte stream between parent and child processes
Named pipes (FIFOs)	Pipes that have a name in the file system; usable by unrelated processes
Message queues	A kernel-managed queue of discrete messages with priority and type tags
Shared memory	A memory region attached to multiple processes
Sockets	Endpoints for network or local IPC; TCP, UDP, or UNIX-domain
Signals	Asynchronous, lightweight notifications (e.g., `SIGINT`)
Semaphores	Synchronization-only — pure signalling, no data transfer

Communication addressing:

- **Direct** — `send(P, msg)` / `receive(Q, msg)` — partner named explicitly.
- **Indirect** — `send(mailbox, msg)` — communicate through a named mailbox.

Synchronization options:

- **Blocking send** — sender waits till delivery.
- **Non-blocking send** — sender resumes immediately.
- **Blocking receive** — receiver waits till a message arrives.
- **Non-blocking receive** — receiver gets either a message or a null.

A robust system typically supports several IPC mechanisms so programmers can pick the right tool: shared memory for speed, message queues for structured exchange, sockets for networking.

Q11. (b) Explain in brief about Monolithic and Microkernel Systems. (7)

Operating system kernels are organized using two main architectures: **monolithic** and **microkernel**.

A **monolithic kernel** packs all OS services — file systems, device drivers, network stack, scheduler, memory manager — into a single large program running entirely in privileged kernel mode. Communication between modules is a normal function call, which is fast. The drawback: a bug or crash in any module can crash the entire kernel. Linux and traditional UNIX are monolithic; Linux mitigates rigidity with **loadable kernel modules**.

A **microkernel** keeps only the **minimum** in kernel mode — typically inter-process communication, basic scheduling, and low-level memory management. Everything else — file systems, device drivers, networking — runs in **user-space servers**. Inter-module communication uses message passing through the microkernel, which adds overhead but isolates failures: a driver crash kills only that server, not the kernel. MINIX 3, QNX, L4, and GNU Hurd are microkernels.

Comparison:

Aspect	Monolithic	Microkernel
Services in kernel space	All	Only essentials
Performance	High	Lower (IPC overhead)
Reliability	Low (one bug crashes all)	High (isolated servers)
Extensibility	Recompile or load module	Just start a new server
Code size	Large (millions of lines)	Tiny (thousands of lines)
Security	Larger attack surface	Smaller TCB
Examples	Linux, UNIX	MINIX 3, QNX, L4

Modern systems often adopt a **hybrid** approach (Windows NT, macOS Darwin): mostly monolithic for performance, but with microkernel-style separation for some services.

Q12. (a) Explain Producer–Consumer Problem and its solution using semaphore. (7)

The **Producer–Consumer problem** (a.k.a. bounded-buffer problem) models a common pattern: one process (the **producer**) generates items and places them in a shared, bounded buffer; another process (the **consumer**) removes items from the buffer to use them. The challenge is to keep them coordinated:

- The **producer must wait** when the buffer is full.
- The **consumer must wait** when the buffer is empty.
- Both must never modify the buffer at the same time.

Semaphore solution. Three semaphores are used:

Semaphore	Initial	Purpose
`empty`	n	Counts the number of empty slots
`full`	0	Counts the number of filled slots
`mutex`	1	Binary semaphore for mutual exclusion on the buffer

Producer code:

```
while (true) {
    produce item;
    wait(empty);          // block if buffer full
    wait(mutex);          // enter critical section
    buffer[in] = item;
    in = (in + 1) % n;
    signal(mutex);
    signal(full);         // one more filled slot
}
```

Consumer code:

```
while (true) {
    wait(full);           // block if buffer empty
    wait(mutex);
    item = buffer[out];
    out = (out + 1) % n;
    signal(mutex);
    signal(empty);        // one more empty slot
    consume item;
}
```

Why this is correct. `empty` and `full` enforce synchronization (no produce-on-full, no consume-on-empty). `mutex` enforces mutual exclusion on the buffer pointers. The order of waits matters: always **`wait(empty)` before `wait(mutex)`** (and likewise for consumer). Reversing the order would deadlock — if the buffer were full, the producer would hold `mutex` while waiting for `empty`, preventing the consumer from freeing space.

Q12. (b) Explain the following terms: (7)

(i) Multilevel Feedback Queue Scheduling

Multilevel feedback queue (MLFQ) scheduling uses several ready queues organized by priority. A new process enters the top (highest-priority) queue. Each queue may use a different scheduling algorithm (commonly Round Robin in upper queues with small quanta, and FCFS in the bottom queue). The key feature is that **processes move between queues** based on observed CPU behaviour: if a process uses its full quantum, it is demoted to a lower-priority queue with a longer quantum; if it gives up the CPU early (likely I/O-bound), it stays in or moves up to a higher-priority queue. Five tunable parameters are: number of queues, scheduling policy per queue, promotion rule, demotion rule, and initial queue. MLFQ automatically separates CPU-bound from I/O-bound and interactive processes, providing good responsiveness with high throughput.

(ii) Fixed Partitioning vs Variable Partitioning

Fixed partitioning divides main memory into a fixed number of partitions of pre-determined sizes at boot time. Each partition holds exactly one process. It is simple to implement but suffers from **internal fragmentation** (a small process in a large partition wastes the remainder) and an inflexible cap on the number of concurrent processes.

Variable partitioning allocates only as much memory as each process needs, when it needs it. The OS maintains a free-list of holes. Internal fragmentation is eliminated, but **external fragmentation** appears as small unusable holes accumulate between allocations. Compaction (relocating processes to merge holes) can fix this but is expensive. Placement policies — **first-fit, best-fit, worst-fit** — choose which hole to use.

	Fixed	Variable
Partition sizes	Predetermined	Dynamic
Internal fragmentation	Yes	No
External fragmentation	No	Yes
Implementation	Simple	Free-list mgmt
Memory utilization	Lower	Higher

Q13. (a) Disk: 100 tracks; requests 45, 20, 90, 10, 50, 60, 80, 25, 70; head at 49. Net head movement using SCAN, LOOK, SSTF. (7) Initial assumption (state at top of answer): Head is at track 49 moving towards higher track numbers. Disk tracks numbered 0–99.

Sorted requests: 10, 20, 25, 45, 50, 60, 70, 80, 90.

(i) SCAN Head moves up to disk end (99), then reverses and serves remaining requests downward.

Order: 49 → 50 → 60 → 70 → 80 → 90 → 99 → 45 → 25 → 20 → 10.

- Up: $99 - 49 = 50$
- Down: $99 - 10 = 89$

Total head movement = 50 + 89 = 139 tracks.

(ii) LOOK Same as SCAN but reverses at the **last request** in each direction.

Order: 49 → 50 → 60 → 70 → 80 → 90 → 45 → 25 → 20 → 10.

- Up: $90 - 49 = 41$
- Down: $90 - 10 = 80$

Total head movement = 41 + 80 = 121 tracks.

(iii) SSTF Always serve the closest pending request.

Order: 49 → 50 → 45 → 60 → 70 → 80 → 90 → 25 → 20 → 10.

Distances: 1, 5, 15, 10, 10, 10, 65, 5, 10.

Total head movement = 1 + 5 + 15 + 10 + 10 + 10 + 65 + 5 + 10 = 131 tracks.

Summary

Algorithm	Total head movement
SCAN	**139**
LOOK	**121**
SSTF	**131**

LOOK is the cheapest here because it avoids the wasted trip from track 90 to 99.

Q13. (b) Explain RAID and its characteristics. Various RAID levels. (7)

RAID — Redundant Array of Independent Disks — is a technology that combines multiple physical disks into a single logical unit to improve **performance**, **reliability**, or both. Three core techniques are used in various combinations:

1. **Striping** — data is split across multiple disks so that reads and writes proceed in parallel. Improves throughput.

2. **Mirroring** — identical copies of data are kept on two or more disks. Provides redundancy.
3. **Parity** — a checksum (typically XOR) is computed across a stripe so that any one missing disk can be reconstructed. Balances cost and redundancy.

Characteristics:

- **Fault tolerance** — number of disk failures that can be sustained without data loss.
- **Capacity efficiency** — fraction of raw capacity available to users.
- **Performance** — read/write throughput compared to a single disk.
- **Cost** — number of disks required for a given capacity.
- **Recovery** — time and complexity to rebuild after a failure.

RAID levels:

Level	Technique	Min Disks	Tolerates	Notes
0	Striping only	2	None	Fastest; zero redundancy
1	Mirroring	2	1 disk	High reliability; 50% capacity
2	Bit-level striping + Hamming parity	3+	1 bit	Rare in practice
3	Byte-level striping + dedicated parity disk	3+	1 disk	Large sequential I/O
4	Block-level striping + dedicated parity disk	3+	1 disk	Parity disk bottleneck
5	Block-level striping + **distributed** parity	3+	1 disk	Most popular; good balance
6	Striping + **two** distributed parity blocks	4+	2 disks	Higher reliability than RAID 5
10 (1+0)	Mirror, then stripe across mirrors	4+	1 per pair	Performance and reliability

RAID protects against disk failure but **is not a backup**: it does not protect against accidental deletion, file corruption, or disaster. A complete data-protection strategy combines RAID with off-site backups.

4.2 Paper 2 — TU-830(A), earlier sitting

Section A (2 marks each, ≤ 75 words)

Q1. Briefly define the term Real Time Operating System.

See Paper 1 Q1 — same answer. Quick recap:

A **Real Time Operating System (RTOS)** processes inputs and produces outputs within strict, predictable time deadlines. Correctness depends on both the logical result and the time at which it is delivered. **Hard real-time** systems (avionics, pacemakers) treat missed deadlines as failure; **soft real-time** systems (multimedia, gaming) tolerate occasional misses. Examples: VxWorks, QNX, FreeRTOS, RTLinux. Used in embedded, industrial, robotic, and safety-critical applications.

Q2. What do you mean by concurrent processes?

Concurrent processes are two or more processes whose executions overlap in time. They may run truly in parallel on multiple CPUs, or be interleaved on a single CPU via time-sharing. Concurrency introduces challenges: **race conditions** when accessing shared data, the need for **mutual exclusion** on critical sections, possible **deadlock** and **starvation**. Synchronization tools — semaphores, mutexes, monitors — are used to coordinate concurrent processes safely and correctly.

Q3. What are the performance criteria in CPU scheduling?

CPU scheduling algorithms are evaluated on five primary criteria:

1. **CPU utilization** — fraction of time CPU is busy (maximize).
2. **Throughput** — number of processes completed per unit time (maximize).
3. **Turnaround time** — total time from submission to completion (minimize).
4. **Waiting time** — time spent in the ready queue (minimize).
5. **Response time** — time from submission to first CPU allocation (minimize for interactive systems).

Different algorithms make different trade-offs among these criteria.

Q4. Explain in brief about the Multiprogramming with Fixed Partitions.

In **multiprogramming with fixed partitions (MFT)**, main memory is divided into a **fixed number of partitions** of pre-determined sizes at system boot. Each partition can hold exactly one process. The OS uses either separate input queues per partition or a single queue with placement rules. Multiple processes thus reside in memory simultaneously, sharing the CPU. The scheme is simple but suffers from **internal fragmentation** (a small process in a large partition wastes the remaining space) and an inflexible cap on concurrent processes.

Q5. What do you mean by the I/O Buffering?

See Paper 1 Q4 — same answer.

I/O buffering uses a region of memory (a buffer) to temporarily hold data while it is being transferred between two devices, or between a device and an application. Buffering smooths out the speed mismatch between the fast CPU/memory and slow I/O devices, allows overlap of I/O and computation, and supports block-oriented operations. Three common schemes: **single**, **double**, and **circular** buffering.

Section B (9 marks each, attempt any 2 of 3)

Q6. Explain in detail about the Deadlock system model and Deadlock characterization. (9)

System model. The operating system has n processes $\{P_1, P_2, \ldots, P_n\}$ and m resource types $\{R_1, \ldots, R_m\}$. Each resource type may have several identical instances. To use a resource, a process must follow this sequence:

1. **Request** — issue a request to the OS; if unavailable, the process waits.
2. **Use** — operate on the resource.
3. **Release** — return the resource to the OS.

A **deadlock** is a situation in which a set of processes are each waiting for an event (typically resource release) that can only be triggered by another process in the same set. None of them can ever make progress.

Deadlock characterization (Coffman conditions). A deadlock arises if and only if **all four** of the following conditions hold simultaneously in a system:

1. **Mutual exclusion** — at least one resource is non-shareable; only one process can use it at a time.
2. **Hold and wait** — a process holding at least one resource can request additional resources held by others.
3. **No preemption** — resources cannot be forcibly removed from a process; they must be released voluntarily.
4. **Circular wait** — there exists a circular chain of processes $P_1 \to P_2 \to \ldots \to P_n \to P_1$, where each P_i waits for a resource held by the next.

Resource Allocation Graph (RAG). A directed graph where round nodes are processes, square nodes are resources (with dots indicating instances). An edge $P_i \to R_j$ is a **request edge**; $R_j \to P_i$ is an **assignment edge**.

- If the RAG has **no cycle**, no deadlock exists.
- If a cycle exists and **every resource type has exactly one instance**, deadlock is present.
- If a cycle exists with **multiple instances per resource type**, deadlock is **possible** but not certain; further analysis (e.g., banker's algorithm) is required.

Approaches to handle deadlock:

1. **Prevention** — design system so at least one of the four conditions cannot hold (e.g., resource ordering to prevent circular wait).
2. **Avoidance** — use information about future resource needs (Banker's algorithm) to allow only requests that keep the system in a safe state.
3. **Detection and Recovery** — allow deadlock to occur, then detect via RAG cycle analysis and recover by terminating processes or pre-empting resources.
4. **Ignore (Ostrich)** — assume deadlock will not occur; restart manually if it does. Used by mainstream OSes for simplicity.

Q7. Explain in detail about the Dining Philosopher Problem. (9)

See Paper 1 Q9(b) for the core solution. The 9-mark version adds:

The **Dining Philosopher Problem** (Dijkstra, 1965) models a synchronization scenario in which five philosophers sit around a circular table, alternating between thinking and eating noodles. Between each adjacent pair lies a single fork; to eat, a philosopher needs **both** the left and right forks. The problem captures the essence of multiple processes contending for shared, limited resources.

The naive (buggy) semaphore solution. Each fork is a binary semaphore initialized to 1:

```
semaphore fork[5] = {1,1,1,1,1};
philosopher_i():
  while (true) {
    think();
    wait(fork[i]);            // pick up left
    wait(fork[(i+1) % 5]);    // pick up right
    eat();
    signal(fork[i]);
    signal(fork[(i+1) % 5]);
  }
```

If every philosopher simultaneously picks up the left fork, every right wait blocks forever — deadlock. All four Coffman conditions are present.

Solutions:

1. **Allow at most $N - 1$ philosophers at the table.** Use a semaphore `room = 4`. With at most four seated, at least one fork pair is free; hold-and-wait is broken. Deadlock-free **and** starvation-free.
2. **Asymmetric (Tanenbaum) pickup.** Odd-numbered philosophers pick up the right fork before the left; even-numbered do the opposite. At least one philosopher reaches for forks in the opposite order, so no circular wait. Deadlock-free, but starvation possible.
3. **Atomic two-fork pickup using a monitor.** Wrap both `wait` calls inside another mutex so a philosopher either grabs both forks or none. Deadlock-free; reduces concurrency.

Lessons from the problem:

- Deadlock can arise from simple, symmetric code.
- Breaking any one of the four Coffman conditions suffices to prevent deadlock.
- **Starvation-freedom** is a stronger property than deadlock-freedom and may require additional care (e.g., a "test" predicate combined with state tracking).
- Performance and fairness must be balanced — restricting pickup order avoids deadlock but slows the system.

Q8. Disk has 200 cylinders (0–199). Head at 143 (previous request at 125). Queue: 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. SSTF and FCFS. (9)

Note on the request list: the requests 1470, 1774, 1509, 1750 exceed 199. We interpret them in the spirit of the question (the disk has *more* than 200 cylinders, or the head/cylinder values should be taken as given). We work with the literal numbers in the queue. If the question is strict on 0–199, ignore out-of-range requests and only schedule {86, 130, 913 (?), 1022 (?)…}. Below we use the literal values.

State: head at 143; previous request at 125, so head is currently moving toward higher cylinder numbers (used by SCAN-family algorithms; doesn't affect FCFS or SSTF).

(i) FCFS Serve in arrival order: 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.

From	To	Distance
143	86	57
86	1470	1384
1470	913	557
913	1774	861
1774	948	826
948	1509	561
1509	1022	487
1022	1750	728
1750	130	1620

FCFS total = 57 + 1384 + 557 + 861 + 826 + 561 + 487 + 728 + 1620 = 7081 cylinders.

(ii) SSTF Always pick the closest pending request.

Step	Head	Pending closest	Pick	Move
1	143	130 (13), 86 (57), 913 (770)…	**130**	13
2	130	86 (44), 913 (783)…	**86**	44
3	86	913 (827), 948 (862), 1022 (936)…	**913**	827
4	913	948 (35), 1022 (109), 1470 (557)…	**948**	35
5	948	1022 (74), 1470 (522)…	**1022**	74
6	1022	1470 (448), 1509 (487)…	**1470**	448
7	1470	1509 (39), 1750 (280), 1774 (304)	**1509**	39
8	1509	1750 (241), 1774 (265)	**1750**	241
9	1750	1774 (24)	**1774**	24

SSTF total = 13 + 44 + 827 + 35 + 74 + 448 + 39 + 241 + 24 = 1745 cylinders.

Summary

Algorithm	Total head movement
FCFS	7081
SSTF	1745

SSTF reduces total head movement dramatically. The trade-off is potential starvation of far-away requests — none here, but a constant stream of near-cylinder requests could indefinitely delay a far request.

Section C (14 marks each, attempt any 3 of 5)

Q11. (a) Banker's Algorithm: Need matrix and safety check. (7)

The **Banker's Algorithm** is a deadlock-avoidance algorithm. It works for systems with multiple instances of multiple resource types. Each process must declare its **maximum** demand for each resource type in advance. When a process requests resources, the OS only grants the request if doing so leaves the system in a **safe state** — meaning some sequence of process completions can satisfy all remaining maximum demands.

Step 1: Compute the Need matrix.

$$\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$$

Need is what each process may still ask for in the future.

Worked example with resources A (10), B (5), C (7):

Process	Allocation (A,B,C)	Max (A,B,C)	**Need (A,B,C)**
P0	0,1,0	7,5,3	**7,4,3**
P1	2,0,0	3,2,2	**1,2,2**
P2	3,0,2	9,0,2	**6,0,0**
P3	2,1,1	2,2,2	**0,1,1**
P4	0,0,2	4,3,3	**4,3,1**

Sum of Allocation = (7, 2, 5). **Available = (10–7, 5–2, 7–5) = (3, 3, 2).**

Step 2: Safety algorithm.

Initialize: Work = (3,3,2), Finish = false for all.

Iteration	Work	Find P_i with $\text{Need}[i] \leq \text{Work}$	Action
1	(3,3,2)	P1 needs (1,2,2) ≤ (3,3,2) ✓	Run P1. Work = (3+2, 3+0, 2+0) = (5,3,2)
2	(5,3,2)	P3 needs (0,1,1) ≤ (5,3,2) ✓	Run P3. Work = (5+2, 3+1, 2+1) = (7,4,3)
3	(7,4,3)	P0 needs (7,4,3) ≤ (7,4,3) ✓; P4 needs (4,3,1) ≤ (7,4,3) ✓	Run P4 first. Work = (7+0, 4+0, 3+2) = (7,4,5)
4	(7,4,5)	P0 needs (7,4,3) ≤ (7,4,5) ✓	Run P0. Work = (7+0, 4+1, 5+0) = (7,5,5)
5	(7,5,5)	P2 needs (6,0,0) ≤ (7,5,5) ✓	Run P2. Work = (7+3, 5+0, 5+2) = (10,5,7)

All processes have completed. **The system is in a safe state.**

Safe sequence: ⟨ P1, P3, P4, P0, P2 ⟩.

Q11. (b) Write short notes on:

(i) File system protection and security

See Paper 1 Q6 — same content. Tight 3.5-mark version: file system protection is the OS mechanism controlling who can perform which operations (read, write, execute, append, delete, list) on a file. Common schemes are **Access Control Lists** (per-file (user, rights) pairs) and the **Unix 9-bit owner/group/others** model. Security defends against external threats — viruses, worms, trojans, denial-of-service, buffer overflows — using authentication (passwords, biometrics, multi-factor), encryption, auditing, sandboxing, and the principle of least privilege.

(ii) Linked file allocation methods

In **linked allocation**, a file is stored as a chain of disk blocks scattered across the disk; each block contains data plus a pointer to the next block. The directory entry records only the start (and optionally the end) block. Strengths: no external fragmentation, easy file growth. Weaknesses: slow **random access** (must follow the chain), some space lost to the pointer in each block, single-pointer corruption can lose the rest of the file. The **File Allocation Table (FAT)** keeps all pointers in one in-memory table, dramatically speeding traversal while retaining non-contiguity. Used by MS-DOS, FAT16, FAT32, and many flash drives.

Q12. (a) Difference between External and Internal Fragmentation. How to solve using paging? (7)

Internal fragmentation is wasted space **inside** an allocated memory block — the block is bigger than the process actually needs, and the leftover space is wasted but still allocated. It typically arises with **fixed-size partitions** or any allocator that rounds up requests to a fixed unit. For example, if a process needs 9 KB and the OS gives it a 16 KB partition, 7 KB are internally fragmented.

External fragmentation is wasted space **outside** any allocation, scattered as small unusable holes between the allocated regions. It accumulates over time in **variable-size partition** schemes as processes load and depart. Total free space may be ample, but no single hole is large enough for a new request.

	Internal	External
Where wasted	Inside allocated block	Between allocated blocks
Typical scheme	Fixed partitioning, paging	Variable partitioning
Fix	Smaller partitions / better fit	Compaction or paging

Paging eliminates external fragmentation. The OS divides physical memory into equal-sized **frames** and logical memory into equally sized **pages**. Any page

can map to any free frame — contiguity is no longer required. The page table maps logical page numbers to physical frame numbers. Because every page is the same size and every frame is interchangeable, no scattered free holes can accumulate as "external" waste. However, paging still causes **internal fragmentation** at the last page of each process (which is usually not fully used) — average loss is about half a page per process.

Q12. (b) What is Thrashing? Cause? Detection? How to eliminate? (7)

Thrashing is a condition in which the operating system spends more time **paging** than executing useful work, causing system throughput to collapse. The CPU sits idle while the disk system is overwhelmed with page-fault traffic.

Cause. Too many processes are loaded into memory, so each one's allotted frames are smaller than its **working set** — the set of pages it actively references. Every reference triggers a page fault; the new page evicts another page that will be needed again shortly, triggering another fault. Worse, when the CPU drops idle waiting for paging I/O, the long-term scheduler often misinterprets this as "CPU is idle, must need more processes" and **admits more processes**, deepening the problem.

Detection. The classic symptoms are simultaneous **high disk-queue length, low CPU utilization, and very high page-fault frequency**. Two practical techniques:

1. **CPU utilization vs degree of multiprogramming curve** — utilization rises with multiprogramming up to a threshold, then drops sharply. The drop is thrashing.
2. **Page-Fault Frequency (PFF)** — monitor each process's PFF. A value above an upper bound indicates the process is short of frames; below a lower bound indicates it has more than enough.

Elimination:

1. **Working Set Model.** Define the working set $W(t, \Delta)$ as the pages referenced in the last Δ memory accesses. Sum working-set sizes; if the total exceeds available frames, **suspend** some processes (swap them out) until total demand fits. As frames free up, processes resume.
2. **Page-Fault-Frequency Control.** If a process's PFF is too high, give it more frames. If too low, take frames away. If no frames available globally, suspend a process to free memory.

Both approaches break the vicious feedback loop by **bounding** the multiprogramming degree based on memory pressure rather than CPU idle time.

Q13. (a) What are files? Explain access methods for files. (7)

A **file** is a logical, named collection of related information stored on secondary storage. From the user's view, a file is the smallest unit of logical storage that can be created, named, read, written, shared, or deleted. The OS hides the physical storage details (sectors, blocks, cylinders) behind a uniform file interface.

File attributes include name, identifier, type, location, size, protection bits, owner, and timestamps.

File operations include create, open, read, write, seek (reposition), close, delete, truncate, and append.

Access methods describe how a program reads and writes the file's contents:

1. **Sequential access.** Bytes (or records) are read or written in order. The file maintains an implicit position pointer that advances after each operation. To restart, the file must be rewound. This is the simplest, oldest, and most common access pattern, used by editors, compilers, and text-processing tools. Operations: `read_next`, `write_next`, `reset`.
2. **Direct (random) access.** The file is viewed as a sequence of equal-sized, numbered blocks; any block can be read or written directly given its block number. Useful for large files where random access is essential, such as databases and indexed data files. Operations: `read n`, `write n`, `position n`. The block number is sometimes relative (to the start of the file) rather than absolute on disk.
3. **Indexed access.** An **index file** holds pointers to records (or blocks) of the data file. To read a record, the program first searches the index, then jumps to the data block. Used in large indexed file systems and many database engines. Variants include **indexed sequential** access, where the index is sparse and within each index range the file is sequential — used by ISAM and B-tree-based systems.

Access method (program-facing) and file organization (storage layout) are related but distinct concepts: a sequentially organized file can still be accessed randomly via byte offset, though efficiency suffers.

Q13. (b) (Note: appeared as part of File concept question — see (a) above for the access method content, and Paper 1 Q9(a)(ii) for linked allocation if requested.)

Other Section C topics from Paper 2 (covered in earlier topic teachings):

- **External vs Internal Fragmentation + Paging** → §3.12 and Q12(a) above.
- **Thrashing** → §3.13 and Q12(b) above.

Part 5: Quick Reference Card

One spread that you can mentally photograph the night before. Everything that has *ever* shown up in either paper, distilled.

5.1 Core Tuples & Definitions

Concept	Definition / Tuple
Process	Program in execution, with PC, stack, data section, heap
PCB	(PID, state, PC, registers, memory limits, open files, scheduling info)
Process states	new, ready, running, waiting, terminated
Thread	Lightweight unit of execution inside a process, shares code/data/files but has own stack & registers
Semaphore	Integer with atomic `wait(S)` (P) and `signal(S)` (V) operations
Critical section	Code region accessing shared resource; mutex needed
Deadlock	Set of processes each waiting for a resource held by another in the set
Coffman conditions	Mutual Exclusion, Hold-and-Wait, No-Preemption, Circular-Wait (all 4 required)
Safe state	At least one safe sequence exists in which every process can finish
Page fault	Reference to a page not currently in any frame
Belady's anomaly	More frames → more page faults (FIFO only)
Thrashing	CPU spends more time paging than computing
Working set	Set of pages a process actually references in a recent time window
Fragmentation (Internal)	Wasted space *inside* an allocated block
Fragmentation (External)	Wasted space *between* allocated blocks
Paging	Memory divided into fixed-size frames; logical address split into (page number, offset)
File	Named collection of related information recorded on secondary storage
RAID	Redundant Array of Independent Disks, combining drives for performance + reliability
RTOS	OS where correctness depends on logical result *and* time deadline

5.2 All Formulas in One Place

What	Formula
Effective access time (paging)	$EAT = (1-p)\cdot m + p\cdot(\text{page fault service})$
Effective access time (TLB)	$EAT = h\cdot(T_{TLB}+T_{mem}) + (1-h)\cdot(T_{TLB}+2T_{mem})$
Logical address split	LA $=$ (page no., offset) where page no. indexes the page table
Physical address split	PA $=$ (frame no., offset)
Disk access time	$T_{access} = T_{seek} + T_{rot} + T_{transfer}$
Avg rotational latency	$T_{rot} = \frac{1}{2}\cdot\frac{60}{\text{RPM}}$ seconds
Banker's Need	$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$
Banker's safety check	$\text{Need}_i \le \text{Work}$? then $\text{Work} += \text{Allocation}_i$
Throughput	processes completed / unit time
Turnaround time	completion time − arrival time
Waiting time	turnaround time − burst time
Response time	first CPU time − arrival time

5.3 Scheduling Algorithms at a Glance

Algorithm	Preemptive?	Selects	Strength	Weakness
FCFS	No	Earliest arrival	Simple, fair	Convoy effect
SJF	Either	Shortest burst	Min avg waiting	Starvation, burst unknown
Priority	Either	Highest priority	Flexible	Starvation (fix: aging)
Round Robin	Yes	Next in queue (time slice)	Fair, responsive	Quantum tuning hard
MLFQ	Yes	Highest-priority non-empty queue	Adapts to behavior	Complex parameters

5.4 Page Replacement Algorithms

Algorithm	Replaces	Pros	Cons
FIFO	Oldest page in memory	Trivial	Belady's anomaly
LRU	Least recently used	Strong, no anomaly	Needs hardware/extra bookkeeping
Optimal	Page used farthest in future	Lower bound (best possible)	Not implementable in real OS
LFU	Least frequently used	Captures access count	Old-popular pages stay forever
Second-chance	FIFO with reference-bit reprieve	Cheap LRU approximation	Still scans queue

5.5 Disk Scheduling Algorithms

Algorithm	Order	Strength	Weakness
FCFS	Request arrival order	Fair, simple	Long seeks
SSTF	Nearest request to head	Low avg seek	Starvation of far requests
SCAN	Sweep one end, reverse, sweep back (elevator)	No starvation	End requests wait longer
C-SCAN	Sweep one end, jump back, sweep again	More uniform wait	Larger total head movement
LOOK	SCAN but reverses at last request, not disk end	Less wasted travel	Slightly more complex
C-LOOK	C-SCAN but at last request	Uniform + efficient	Same

5.6 RAID Quick Card

Level	Technique	Min disks	Fault tolerance	Use case
0	Striping	2	None	Speed only
1	Mirroring	2	1 disk	OS volumes
2	Bit-level + Hamming	3	1 disk	Historic
3	Byte-level + parity disk	3	1 disk	Rare
4	Block-level + parity disk	3	1 disk	Parity bottleneck
5	Block-level + distributed parity	3	1 disk	General-purpose
6	Block-level + double parity	4	2 disks	Large arrays
10	Mirror + Stripe	4	up to 1 per mirror	Databases

5.7 File Allocation Methods

Method	Layout	Pros	Cons
Contiguous	Sequential blocks	Fast access, simple	External fragmentation, hard to grow
Linked	Each block points to next	No external fragmentation, easy growth	Slow random access, pointer overhead
Indexed	Index block holds all pointers	Fast random access	Index block overhead, max file size
FAT (linked variant)	Pointers in central table	Used in DOS/FAT32	Table size, slow random

5.8 IPC Mechanisms

Mechanism	Direction	Buffer	Notes
Shared memory	Bidirectional	User memory	Fastest; needs sync
Message passing	Direct or indirect	Kernel queue	Slower, safer
Pipe	Unidirectional	Kernel	Same machine, related procs
Named pipe (FIFO)	Bidirectional (with two)	Kernel	Unrelated procs
Socket	Bidirectional	Kernel	Cross-machine

Part 6: Top Mistakes to Avoid in the Exam

Twenty traps that will leak marks if you slip on them. Read this twice the night before.

6.1 Disk Scheduling

1. **Forgetting to state the direction.** SCAN and LOOK need a stated direction (toward higher or lower cylinders). Examiner wants to see "head moves toward 0" or "head moves toward 199".
2. **Mixing SCAN and LOOK.** SCAN goes to the *end of the disk* (track 0 or last track). LOOK only goes to the *last request* in that direction. Different totals.
3. **Forgetting absolute value.** Head movement is always $|\text{new} - \text{old}|$, never signed.
4. **Out-of-range requests.** If a request is outside the disk's track range (e.g. 1774 on a 200-cylinder disk), note it and ignore. Don't try to "serve" it.
5. **Wrong starting point.** Always start from the *current* head position, not from 0.

6.2 Page Replacement

6. **Counting initial loads wrong.** First reference to any page is always a page fault (cold start). Don't skip the first three.
7. **LRU vs FIFO confusion.** FIFO evicts oldest *arrival*; LRU evicts oldest *use*. They diverge as soon as any page is re-referenced.
8. **Optimal needs the full future.** When picking which page to evict in Optimal, scan the *entire remaining* reference string, not just the next few.
9. **Belady's anomaly applies only to FIFO.** LRU and Optimal are stack algorithms; more frames → ≤ faults. Don't say "Belady's affects LRU".

6.3 Banker's Algorithm

10. **Computing Need wrong.** Need = Max − Allocation. Sign matters; you subtract Allocation *from* Max, not the other way.
11. **Available not updated.** After a process is granted in the safety check, you must add its Allocation row to Work for the next iteration. Forgetting this leads to wrong unsafe verdict.
12. **Stating sequence wrong.** The safe sequence is the *order in which processes finish*, not the order you tested them.

6.4 Synchronization

13. **Wrong order of `wait()` calls.** In producer/consumer, the producer must `wait(empty)` *before* `wait(mutex)`. Reverse order causes deadlock. Same trap in dining philosophers.
14. **Forgetting `signal()` after critical section.** Every `wait` on a mutex must be matched by a `signal`. Drop the signal and the system locks up.
15. **Semaphore initial values.** `mutex=1`, `empty=n`, `full=0`. Setting `full=n` is the classic exam slip.

6.5 Memory Management

16. **Internal vs External fragmentation.** Internal = *inside* a block (paging). External = *between* blocks (variable partitioning). They are *not* the same; paging removes external but introduces a small internal at the last page.
17. **Page table location.** Page table lives in main memory (with optional TLB cache). Saying "in the page itself" loses marks.

6.6 Deadlock

18. **Coffman conditions are necessary, not sufficient.** All four must hold *simultaneously* for deadlock. Question often asks for all four; missing one drops a mark.
19. **RAG with multiple instances.** Cycle in a Resource-Allocation Graph implies deadlock *only* if each resource type has one instance. With multiple instances, cycle = *possible* deadlock, not certain.

6.7 General Exam Hygiene

20. **Word-limit slips.** Section A is 75-word max; Section B is 200-word max. Examiners do count. Tight, structured answers score higher than rambling.

    One golden rule: if a question says "explain in detail", give (definition + diagram + example + at least one merit/demerit). That formula alone covers 70% of the marking scheme.

Part 7: One-Page Final Cheat Sheet

Photograph this page in your mind. Glance at it ten minutes before the exam.

OS Functions (memorize 6)

Process management · Memory management · File management · I/O management · Security & protection · Networking & communication.

Process states

new → ready → running → (waiting ↔ ready) → terminated.

Coffman 4 (deadlock requires *all* four)

Mutual Exclusion · Hold-and-Wait · No-Preemption · Circular-Wait.

Banker's

Need $=$ Max $-$ Allocation. Safe iff a sequence exists with Need$_i \leq$ Work at each step. After granting, Work $+=$ Allocation$_i$.

Page-fault formula (3-frame example)

Track frame contents per reference. Hit = page already in frame. Miss = page fault, evict by policy.

Disk-scheduling totals (always $\sum |\textbf{new} - \textbf{old}|$)

FCFS: in queue order. SSTF: pick nearest. SCAN: sweep to end, reverse. LOOK: sweep to last request, reverse.

Fragmentation

Internal = wasted *inside* a block. External = wasted *between* blocks. Paging fixes external; introduces tiny internal at last page.

File access methods

Sequential · Direct (random) · Indexed.

File allocation

Contiguous · Linked (incl. FAT) · Indexed.

RAID-0 vs RAID-1 vs RAID-5

0 = stripe, no redundancy. 1 = mirror. 5 = stripe + distributed parity (tolerates 1 disk).

Scheduling criteria

CPU utilization (↑) · Throughput (↑) · Turnaround time (↓) · Waiting time (↓) · Response time (↓).

Producer-Consumer semaphores

`mutex=1`, `empty=n`, `full=0`. Producer: `wait(empty); wait(mutex); add; signal(mutex); signal(full)`. Consumer: mirror with `full` and `empty` swapped.

Dining Philosopher fixes

(a) odd/even asymmetry · (b) at most $n-1$ allowed · (c) pick both forks atomically.

Microkernel vs Monolithic

Monolithic = all in kernel space, fast, fragile. Microkernel = minimal kernel, services in user space, modular, slower.

Multiuser vs Multithreaded

Multiuser = many *people* share one system. Multithreaded = one *process* runs many threads concurrently. Orthogonal concepts.

Thrashing

CPU spends more time paging than computing. Cause: too many processes, too little memory. Detect: CPU utilization low while paging rate high. Fix: reduce multiprogramming degree, working-set model, page-fault frequency control.

Part 8: A Note Before You Go In

You've now read more about Operating Systems in one document than half the class will read all semester. That's worth something.

Three things to keep in mind tomorrow:

One. When you sit down with the question paper, spend the first three minutes just *reading*. Mark the questions you want to attempt in Section B (2 of 3) and Section C (3 of 5). Don't start writing until you've made the choice. Switching mid-answer costs more time than the choice itself.

Two. Lead every long answer with a one-line definition and a diagram if relevant. Examiners scan first, read second. A diagram and a bold key-term on line one tells them you know the topic before they read a paragraph.

Three. If you blank on something, write *anything related* and move on. A page-replacement question you only half-remember is still worth 4 marks if you draw the table cleanly with two correct entries. Zero is the only result you can guarantee by leaving it blank.

The papers in front of you for the last however-many weeks are not random. The PYQs you saw here will guide what shows up. Disk scheduling, Banker's, page replacement, Dining Philosopher, files, fragmentation, IPC. If you're solid on those seven, you've already secured your 60.

Go in calm. Write clean. Underline key terms. Number sub-parts. Don't panic if a question looks unfamiliar in the first read; it usually maps onto something you already know once you parse the language.

You've got this. Now close this PDF, get some sleep, and write a paper that ends this subject on your terms.

— *Notes prepared for you, one chai-fueled session at a time.*