

OPERATING SYSTEMS

UNIT 4 — Detailed Notes

Deadlocks

Topics covered: Definition • Necessary & Sufficient Conditions • Deadlock Prevention • Deadlock Avoidance (Banker's Algorithm) • Deadlock Detection and Recovery

 Green box = Sticky Note |  Red box = VVI Hotpoint |  Pink box = Numerical Practice Space |  Yellow box = Diagram Suggestion

1. Definition of Deadlock

1.1 What is a Deadlock?

A Deadlock is a situation in which a set of two or more processes are each waiting for a resource that is held by another process in that same set, and because of this circular waiting, NONE of the processes can ever proceed — they stay blocked forever. In simple words, every process in the set is waiting for an event (release of a resource) that only another waiting process in the same set can cause, so that event never happens.

1.2 A Simple Real-Life Analogy

Imagine two cars facing each other on a single-lane narrow bridge, with no room to reverse. Car A is waiting for Car B to move, and Car B is waiting for Car A to move. Neither can move forward, and neither will reverse — this is exactly a deadlock. Both are stuck waiting for something only the OTHER one can provide.

1.3 System Model (How Deadlock Happens with Resources)

A computer system has a fixed number of resources (CPU cycles, memory space, I/O devices, files, etc.) to be distributed among several competing processes. Every process that needs a resource normally follows this sequence:

- **Request:** The process asks the OS for the resource. If not available immediately, the process must wait.
- **Use:** The process uses the resource (e.g., prints to a printer, writes to a file).
- **Release:** The process gives the resource back to the OS so other processes can use it.

1.4 Classic Deadlock Example

Suppose there are two resources: a Printer and a Scanner, and two processes P1 and P2:

Process P1:		Process P2:
request(Printer);		request(Scanner);
request(Scanner);	<-- WAITS	request(Printer);
// use both		// use both
release(Printer);		release(Scanner);
release(Scanner);		release(Printer);

Suppose:

P1 has already acquired the Printer, and is now requesting the Scanner.
P2 has already acquired the Scanner, and is now requesting the Printer.

P1 waits for Scanner (held by P2)
P2 waits for Printer (held by P1)

Neither can proceed --> DEADLOCK

 **Sticky Note:** Simple memory trick for deadlock: 'Everyone is holding something and waiting for something that someone ELSE is holding.' If you can draw this circular waiting pattern, it's a deadlock.

2. Necessary and Sufficient Conditions for Deadlock

A deadlock can occur in a system if and only if ALL FOUR of the following conditions hold true at the SAME TIME. These are called 'necessary and sufficient' conditions because if even ONE of them is false, deadlock cannot happen — and if all four are true, a deadlock situation becomes possible.

Condition	Meaning	Example
Mutual Exclusion	At least one resource must be held in a non-shareable mode — meaning only one process can use that resource at a time. If another process requests it, that process must wait until the resource is released.	A printer — only one print job can use it at a time.
Hold and Wait	A process must be currently holding at least one resource AND waiting to acquire additional resources that are currently held by other processes.	P1 holds the Printer and is waiting for the Scanner.
No Preemption	Resources cannot be forcibly taken away from a process. A resource can only be voluntarily released by the process holding it, after it has completed its task.	The OS cannot snatch the Printer from P1 — P1 must release it itself.
Circular Wait	There must exist a set of waiting processes {P0, P1, ..., Pn} such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., and Pn is waiting for a resource held by P0 — forming a CYCLE.	P1 waits for P2's resource, P2 waits for P1's resource — a cycle of 2.

★ **VVI Hotpoint (Exam Important):** These 4 conditions — Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait — are the SINGLE MOST asked topic in this unit. Remember the short-form 'MHNC' (Mutual exclusion, Hold & wait, No preemption, Circular wait). Deadlock Prevention techniques work by breaking exactly ONE of these four conditions.

2.1 Resource Allocation Graph (RAG)

A Resource Allocation Graph is a directed graph used to give a precise, visual description of the state of the system in terms of processes and resources. It has two kinds of nodes (Processes, drawn as circles, and Resources, drawn as squares/rectangles) and two kinds of edges:

- **Request Edge ($P_i \rightarrow R_j$):** An arrow FROM a process TO a resource, meaning that process is currently requesting (waiting for) that resource.
- **Assignment Edge ($R_j \rightarrow P_i$):** An arrow FROM a resource TO a process, meaning that resource instance has already been allocated to that process.

request edge
(process wants resource)

assignment edge
(resource given to process)

```

+-----+      +-----+      +-----+      +-----+
| P1 |  -----> | R2 |          | R1 |  -----> | P1 |
+-----+      +-----+      +-----+      +-----+
(circle = process)      (square/rectangle = resource)
  
```

Example showing a CYCLE (circular wait) -> possible deadlock:

```

+-----+      ----->      +-----+
| P1 |  ----->      | R2 |
+-----+      <-----+      +-----+
      ^                                |
      |                                v
  
```

```

+-----+ <-----+ +-----+
| R1 |          | P2 |
+-----+ -----> +-----+

```

P1 -> R2 (P1 requests R2) R2 -> P2 (R2 is held by P2)
 P2 -> R1 (P2 requests R1) R1 -> P1 (R1 is held by P1)
 This forms a CYCLE: P1 -> R2 -> P2 -> R1 -> P1

Rule for interpreting cycles in a RAG

Resource Type	Rule
Single instance per resource type	A cycle in the graph means a deadlock has definitely occurred (cycle is both necessary AND sufficient for deadlock).
Multiple instances per resource type	A cycle is NECESSARY but NOT SUFFICIENT for deadlock — a cycle may exist without an actual deadlock, because another process might still release an instance of the resource that breaks the cycle.

★ **VVI Hotpoint (Exam Important):** Very common exam line: 'If each resource type has exactly ONE instance, then a cycle in the Resource Allocation Graph IS a sufficient condition for deadlock. If resource types have multiple instances, a cycle is necessary but NOT sufficient.' Memorize this exactly.

3. Methods for Handling Deadlocks

There are three main strategies an OS can use to deal with the deadlock problem:

Strategy	Approach
Deadlock Prevention	Design the system so that at least one of the 4 necessary conditions can NEVER occur — this guarantees deadlock will never happen.
Deadlock Avoidance	Allow all 4 conditions to be possible, but carefully check EVERY resource request in advance and only grant it if the system stays in a 'safe state' afterward (e.g., Banker's Algorithm).
Deadlock Detection & Recovery	Allow deadlocks to happen, but periodically run an algorithm to check if one HAS occurred, and then take action (recovery) to break it.

(A fourth, less formal 'strategy' used by many general-purpose OS like Windows/Linux is to simply IGNORE the problem entirely, assuming deadlocks are rare enough — this is called the Ostrich Algorithm.)

4. Deadlock Prevention

Deadlock Prevention works by making sure that at least ONE of the four necessary conditions can never hold true in the system. Each of the 4 conditions can be attacked separately:

4.1 Denying Mutual Exclusion

Make resources shareable wherever possible, so that multiple processes can use them at the same time without needing exclusive access.

- **Limitation:** Not practically possible for many resources by nature — e.g., a printer genuinely cannot be shared between two print jobs at the exact same moment without mixing up the output.

4.2 Denying Hold and Wait

Ensure that whenever a process requests a resource, it does not hold any other resource at the same time. Two common approaches:

- **Approach 1:** A process must request and be allocated ALL the resources it will ever need, all at once, BEFORE it starts executing.
- **Approach 2:** A process may request resources only when it holds none; if it needs more later, it must first release everything it currently holds, then request again (including what it already had).
- **Limitation:** Leads to very low resource utilization (resources sit idle, reserved but unused for a long time) and can cause Starvation (a process needing many popular resources may never get them all at once).

4.3 Denying No Preemption

Allow the OS to forcibly take away (preempt) a resource from a process. A common rule: if a process holding some resources requests another resource that cannot be immediately allocated, then all the resources it is CURRENTLY holding are preempted (taken back). The process is restarted later only when it can regain both its old resources and the new ones it requested.

- **Limitation:** Only practical for resources whose state can be easily saved and restored later (like CPU registers or memory) — NOT practical for resources like a printer (you cannot 'preempt' a half-finished print job cleanly).

4.4 Denying Circular Wait

Impose a strict, total (linear) ordering on ALL resource types, and require that every process can only request resources in STRICTLY INCREASING order of this numbering. For example, if Printer=1, Scanner=2, Disk=3, a process holding the Scanner(2) can only request the Disk(3) next, never go back and request the Printer(1).

- **Why it works:** Since every process must request resources in increasing numeric order, it becomes mathematically impossible to form a circular chain of waiting processes — a cycle would require going both 'up' and 'down' the ordering at some point, which is forbidden.

4.5 Summary Table — Prevention Methods

Condition Denied	Technique	Main Drawback
Mutual Exclusion	Make resources shareable	Not possible for many resource types by nature
Hold and Wait	Request all resources at once, or release before requesting more	Low resource utilization, possible starvation
No Preemption	Preempt resources from a waiting process	Only works for resources that are easy to save/restore
Circular Wait	Impose a total ordering on resource types; request in increasing order	Can be inconvenient for programmers, may still underuse resources

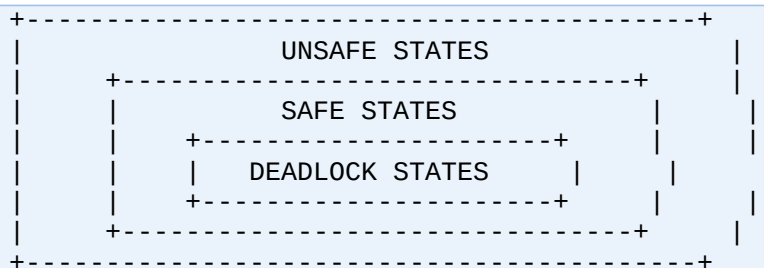
5. Deadlock Avoidance

5.1 Concept of Safe State

Deadlock Avoidance requires the OS to have extra ADVANCE information — specifically, the MAXIMUM number of resources of each type that every process may ever request during its entire lifetime. Using this, before granting ANY resource request, the OS checks whether granting it would leave the system in a Safe State.

- **Safe State:** A state is safe if there EXISTS at least one order (a 'safe sequence') in which ALL processes can be allocated their maximum resource needs one by one and finish successfully, without ever causing a deadlock — even in the worst case where every process asks for its maximum need immediately.
- **Unsafe State:** If no such safe sequence exists. NOTE: an unsafe state does NOT necessarily mean deadlock has already happened — it just means deadlock becomes POSSIBLE. Deadlock Avoidance simply makes sure the system NEVER enters an unsafe state in the first place.

★ **VVI Hotpoint (Exam Important):** Key exam line: 'All safe states are deadlock-free, but NOT all unsafe states lead to deadlock.' Avoidance algorithms are conservative — they reject a request even if it MIGHT still have been okay, just to guarantee safety.



Deadlock states are a SUBSET of unsafe states.
Avoidance keeps the system strictly inside the SAFE STATES region.

5.2 Banker's Algorithm

The Banker's Algorithm (given by Dijkstra) is the most famous Deadlock Avoidance algorithm. It gets its name from the way a bank manager might allocate loans, making sure enough cash is always kept in reserve to satisfy at least one customer's maximum request, so the bank never runs completely dry while customers are still owed money.

5.2.1 Data Structures Used (for n processes, m resource types)

Structure	Type	Meaning
Available	Vector of length m	$Available[j]$ = number of available instances of resource type R_j
Max	$n \times m$ Matrix	$Max[i][j]$ = maximum number of instances of resource type R_j that process P_i may request
Allocation	$n \times m$ Matrix	$Allocation[i][j]$ = number of instances of R_j CURRENTLY allocated to P_i
Need	$n \times m$ Matrix	$Need[i][j] = Max[i][j] - Allocation[i][j]$ (remaining resources P_i may still request)

5.2.2 Safety Algorithm (checks if current state is safe)

1. Let $Work = Available$ (copy); let $Finish[i] = false$ for all $i = 0..n-1$
2. Find an index i such that:
 $Finish[i] == false$ AND $Need[i] \leq Work$ (component-wise)
 If no such i exists, go to step 4.
3. $Work = Work + Allocation[i]$
 $Finish[i] = true$
 Go back to step 2.
4. If $Finish[i] == true$ for ALL i , the system is in a SAFE STATE,
 and the order in which processes were marked $Finish = true$ is a safe sequence.
 Otherwise (some $Finish[i]$ still false), system is in an UNSAFE STATE.

5.2.3 Resource-Request Algorithm (checks if a NEW request can be safely granted)

When process P_i makes a $Request[]$ for resources:

- Step 1: If $Request \leq Need[i]$, go to step 2; otherwise ERROR — process asked for more than its declared maximum.
- Step 2: If $Request \leq Available$, go to step 3; otherwise P_i must WAIT, since resources aren't available right now.
- Step 3: PRETEND to allocate the requested resources ($Available -= Request$, $Allocation[i] += Request$, $Need[i] -= Request$), then RUN the Safety Algorithm on this new, pretend state.
- **If the new state is SAFE:** Actually grant the request (the pretend allocation becomes real/permanent).
- **If the new state is UNSAFE:** P_i must wait, and the system is rolled back to the OLD state (undo the pretend allocation) as if the request was never made.

5.3 Fully Solved Numerical — Banker's Algorithm

Given a system with 5 processes (P0-P4) and 3 resource types: A (10 instances), B (5 instances), C (7 instances):

Process	Alloc-A	Alloc-B	Alloc-C	Max-A	Max-B	Max-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

Given: Available = (3, 3, 2) [Total resources: A=10, B=5, C=7]

Step 1 — Calculate the Need Matrix (Need = Max – Allocation)

Process	Need-A	Need-B	Need-C
P0	7-0=7	5-1=4	3-0=3
P1	3-2=1	2-0=2	2-0=2
P2	9-3=6	0-0=0	2-2=0
P3	2-2=0	2-1=1	2-1=1
P4	4-0=4	3-0=3	3-2=1

Step 2 — Run the Safety Algorithm

Start: Work = Available = (3, 3, 2). Check each process whose Need <= Work:

Work = (3, 3, 2)

Check P0: Need(7,4,3) <= Work(3,3,2)? NO (7>3) --> skip for now

Check P1: Need(1,2,2) <= Work(3,3,2)? YES

--> Work = Work + Allocation[P1] = (3+2, 3+0, 2+0) = (5, 3, 2)

--> Finish[P1] = true Safe sequence so far: <P1>

Check P2: Need(6,0,0) <= Work(5,3,2)? NO (6>5) --> skip for now

Check P3: Need(0,1,1) <= Work(5,3,2)? YES

--> Work = Work + Allocation[P3] = (5+2, 3+1, 2+1) = (7, 4, 3)

--> Finish[P3] = true Safe sequence so far: <P1, P3>

Check P4: Need(4,3,1) <= Work(7,4,3)? YES

--> Work = Work + Allocation[P4] = (7+0, 4+0, 3+2) = (7, 4, 5)

--> Finish[P4] = true Safe sequence so far: <P1, P3, P4>

Check P0: Need(7,4,3) <= Work(7,4,5)? YES

--> Work = Work + Allocation[P0] = (7+0, 4+1, 5+0) = (7, 5, 5)

--> Finish[P0] = true Safe sequence so far: <P1, P3, P4, P0>

Check P2: Need(6,0,0) <= Work(7,5,5)? YES

--> Work = Work + Allocation[P2] = (7+3, 5+0, 5+2) = (10, 5, 7)

--> Finish[P2] = true Safe sequence so far: <P1, P3, P4, P0, P2>

ALL processes Finished --> SYSTEM IS IN A SAFE STATE

★ **VVI Hotpoint (Exam Important):** FINAL ANSWER: The system is in a SAFE STATE, with Safe Sequence = <P1, P3, P4, P0, P2>. This exact numerical (10 A, 5 B, 7 C resources) is a famous textbook example — practice it until you can redo it without looking.

Step 3 — Testing a New Request (Resource-Request Algorithm)

Now suppose process P1 requests Request1 = (1, 0, 2). Let's check if this can be safely granted:

- Check 1: Is Request1 (1,0,2) $\leq$ Need[P1] (1,2,2)? YES, $1 \leq 1$, $0 \leq 2$, $2 \leq 2$ — request does not exceed what P1 declared it may need.
- Check 2: Is Request1 (1,0,2) $\leq$ Available (3,3,2)? YES, $1 \leq 3$, $0 \leq 3$, $2 \leq 2$ — resources are currently available.

Step 3: Pretend to allocate — update the values as if the request is granted:

```
New Available = Available - Request1 = (3-1, 3-0, 2-2) = (2, 3, 0)
New Alloc[P1] = Alloc[P1] + Request1 = (2+1, 0+0, 0+2) = (3, 0, 2)
New Need[P1]  = Need[P1] - Request1 = (1-1, 2-0, 2-2) = (0, 2, 0)
```

Now re-run the Safety Algorithm with Work = New Available = (2, 3, 0):

Check P1: Need(0,2,0) $\leq$ Work(2,3,0)? YES --> Work=(2+3,3+0,0+2)=(5,3,2), safe seq: <P1>

Check P3: Need(0,1,1) $\leq$ Work(5,3,2)? YES --> Work=(5+2,3+1,2+1)=(7,4,3), safe seq: <P1, P3>

Check P4: Need(4,3,1) $\leq$ Work(7,4,3)? YES --> Work=(7,4,5), safe seq: <P1, P3, P4>

Check P0: Need(7,4,3) $\leq$ Work(7,4,5)? YES --> Work=(7,5,5), safe seq: <P1, P3, P4, P0>

Check P2: Need(6,0,0) $\leq$ Work(7,5,5)? YES --> Work=(10,5,7), safe seq: <P1, P3, P4, P0, P2>

New state is SAFE too --> Request1 = (1,0,2) from P1 is IMMEDIATELY GRANTED.

● **Sticky Note:** Notice the safe sequence after granting P1's request turned out to be the SAME as before: <P1,P3,P4,P0,P2>. This will not always be the case — always re-run the full safety check on the NEW numbers, never assume it stays the same.

✏ **Numerical Practice Space:** Practice Yourself: Using the SAME system above (after P1's request is granted, so Available is now (2,3,0)), check whether a new request from P4 of Request4 = (3,3,0) can be granted. (Hint: compare Request4 with Need[P4]=(4,3,1) first — if it fails this very first check, the request must be rejected immediately as invalid, without even checking Available.)

6. Deadlock Detection and Recovery

6.1 When is Detection Used?

If a system does NOT use Prevention or Avoidance, then a deadlock may actually occur. In that case, the OS needs a Detection Algorithm to periodically check whether a deadlock HAS happened, and if so, a Recovery method to break it.

6.2 Deadlock Detection Algorithm (Multiple Instances of Resources)

This algorithm is very similar to the Banker's Safety Algorithm, but it does NOT use the Max or Need matrices (since we are not avoiding deadlock in advance) — it only uses Available, Allocation, and a new matrix called Request (what each process is CURRENTLY asking for right now, not its future maximum).

1. Let $Work = Available$ (copy); for each process P_i :
 if $Allocation[i] == 0$ (holds nothing), set $Finish[i] = true$
 else set $Finish[i] = false$
2. Find an index i such that:
 $Finish[i] == false$ AND $Request[i] \leq Work$ (component-wise)
 If no such i exists, go to step 4.
3. $Work = Work + Allocation[i]$
 $Finish[i] = true$
 Go back to step 2.
4. If $Finish[i] == false$ for SOME i , then the system IS in a deadlocked state, and each process P_i with $Finish[i] == false$ is DEADLOCKED.

6.3 Solved Numerical — Deadlock Detection

5 processes (P0-P4), 3 resource types A, B, C. Available = (0, 0, 0):

Process	Alloc-A	Alloc-B	Alloc-C	Req-A	Req-B	Req-C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

$Work = Available = (0, 0, 0)$

Check P0: $Request(0,0,0) \leq Work(0,0,0)$? YES
--> $Work = Work + Alloc[P0] = (0+0, 0+1, 0+0) = (0, 1, 0)$
--> $Finish[P0] = true$

Check P2: $Request(0,0,0) \leq Work(0,1,0)$? YES
--> $Work = Work + Alloc[P2] = (0+3, 1+0, 0+3) = (3, 1, 3)$
--> $Finish[P2] = true$

Check P1: $Request(2,0,2) \leq Work(3,1,3)$? YES
--> $Work = Work + Alloc[P1] = (3+2, 1+0, 3+0) = (5, 1, 3)$
--> $Finish[P1] = true$

```
Check P3: Request(1,0,0) <= Work(5,1,3)? YES
--> Work = Work + Alloc[P3] = (5+2, 1+1, 3+1) = (7, 2, 4)
--> Finish[P3] = true
```

```
Check P4: Request(0,0,2) <= Work(7,2,4)? YES
--> Work = Work + Alloc[P4] = (7+0, 2+0, 4+2) = (7, 2, 6)
--> Finish[P4] = true
```

ALL processes Finished --> NO DEADLOCK in this system.

Now suppose P2's request changes to (0, 0, 1) instead of (0, 0, 0):

Let's redo the detection. P0 finishes first exactly as before (Work becomes (0,1,0)). But now check P2:

After P0 finishes: Work = (0, 1, 0)

```
Check P2: Request(0,0,1) <= Work(0,1,0)? NO (C: 1 > 0) --> P2 must wait
Check P1: Request(2,0,2) <= Work(0,1,0)? NO (A: 2 > 0) --> P1 must wait
Check P3: Request(1,0,0) <= Work(0,1,0)? NO (A: 1 > 0) --> P3 must wait
Check P4: Request(0,0,2) <= Work(0,1,0)? NO (C: 2 > 0) --> P4 must wait
```

No process can proceed any further.

Finish[P1] = Finish[P2] = Finish[P3] = Finish[P4] = false

--> SYSTEM IS DEADLOCKED. Processes P1, P2, P3, and P4 are deadlocked.

★ **VVI Hotpoint (Exam Important):** This exact modification (changing P2's request from (0,0,0) to (0,0,1)) is the standard textbook way to show how a SMALL change can take a system from 'safe/no deadlock' to 'deadlocked'. Great question to practice: show both versions, with and without deadlock, from the same base data.

6.4 Recovery from Deadlock

Once a deadlock is detected, the system must recover from it. There are two broad approaches:

6.4.1 Process Termination

Method	Description	Advantage	Disadvantage
Abort all deadlocked processes	Terminates every process involved in the deadlock at once.	Guaranteed to break the deadlock completely and immediately	Very expensive — all partial computation/work done by these processes is lost
Abort one process at a time	Terminates ONE deadlocked process, then re-runs the detection algorithm to check if the deadlock is resolved; repeat if needed.	Less wasteful — only as many processes as truly necessary are killed	Costly due to repeatedly re-running the detection algorithm after each termination

When choosing WHICH process to terminate (in the 'one at a time' approach), the OS typically considers factors like: process priority, how long the process has run and how much longer it needs, how many and what type of resources it has used, how many more resources it needs to finish, how many other processes will need to be terminated, and whether the process is interactive or a batch job.

6.4.2 Resource Preemption

Instead of killing whole processes, the OS can preempt (forcibly take back) some resources from one or more processes and give them to other processes, until the deadlock cycle is broken. Three key issues must be handled carefully:

- **Selecting a victim:** Decide which resources and which processes to preempt from, to minimize overall cost (similar factors as process termination above).
- **Rollback:** Since the resource is taken away mid-task, that process generally cannot continue normally — it must be rolled back to some earlier safe point (often all the way back to the beginning) and restarted from there.
- **Starvation:** The OS must make sure the SAME process is not repeatedly chosen as the victim every single time — otherwise that one unlucky process could starve forever. A common fix is to include the number of times a process has already been a victim as a factor in the cost calculation.

Quick Revision Summary

Topic	One-Line Recap
Deadlock	A situation where a set of processes are each waiting for a resource held by another process in the same set, forming a cycle of waiting that never breaks.
4 Necessary Conditions	Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait — ALL four must hold simultaneously for deadlock to be possible.
Resource Allocation Graph	Shows processes (circles) and resources (squares) with request/assignment edges; a cycle means deadlock if resources have single instances.
Deadlock Prevention	Design the system so one of the 4 necessary conditions can never occur, guaranteeing deadlock never happens (but often reduces resource utilization).
Deadlock Avoidance	Use advance knowledge of maximum resource needs and only grant requests that keep the system in a Safe State (Banker's Algorithm).
Safe State	A state from which there exists at least one order (safe sequence) to run all processes to completion without deadlock.
Banker's Algorithm	Uses Available, Max, Allocation, Need matrices; Safety Algorithm checks if current state is safe; Resource-Request Algorithm tests new requests before granting.
Deadlock Detection	Used when prevention/avoidance is not applied; periodically checks (using Available, Allocation, Request) whether a deadlock currently exists.
Deadlock Recovery	Process Termination (abort all, or abort one-by-one) or Resource Preemption (victim selection, rollback, avoiding starvation).

15 Exam-Style Questions — Unit 4

- Q1.** Define Deadlock. Explain with a suitable example involving two processes and two resources.
- Q2.** State and explain the four necessary and sufficient conditions for a deadlock to occur.
- Q3.** What is a Resource Allocation Graph? Explain with a diagram how a cycle in the graph relates to deadlock.
- Q4.** Explain the various methods available for handling deadlocks in an operating system.
- Q5.** Explain Deadlock Prevention. How can each of the four necessary conditions be denied to prevent deadlock?
- Q6.** What is a Safe State? Differentiate between a Safe State, an Unsafe State, and a Deadlocked State.
- Q7.** Explain the data structures used in the Banker's Algorithm (Available, Max, Allocation, Need).
- Q8.** Explain the Safety Algorithm used in Banker's Algorithm with its steps.
- Q9.** Explain the Resource-Request Algorithm used in Banker's Algorithm.
- Q10.** Given a system with processes, Allocation, Max and Available matrices, find the Need matrix and determine if the system is in a safe state using Banker's Algorithm (numerical).
- Q11.** Given the current state of a system and a new resource request from a process, use Banker's Algorithm to determine whether the request can be safely granted (numerical).
- Q12.** Explain the Deadlock Detection Algorithm for a system with multiple instances of each resource type.
- Q13.** Given Allocation and Request matrices with Available resources, use the Deadlock Detection Algorithm to determine if the system is deadlocked, and identify the deadlocked processes (numerical).
- Q14.** Explain the different approaches for Recovery from Deadlock through Process Termination.
- Q15.** Explain Recovery from Deadlock through Resource Preemption, and the three key issues (victim selection, rollback, starvation) that must be addressed.