Unit 3: Process Deadlocks (6 Hrs.)

3.1 Introduction to Deadlocks

Definition and Basic Concepts

A deadlock represents a permanent blocking condition where a set of processes remain indefinitely waiting because each process holds resources needed by another process in the set. This creates a circular wait situation where no process can proceed.

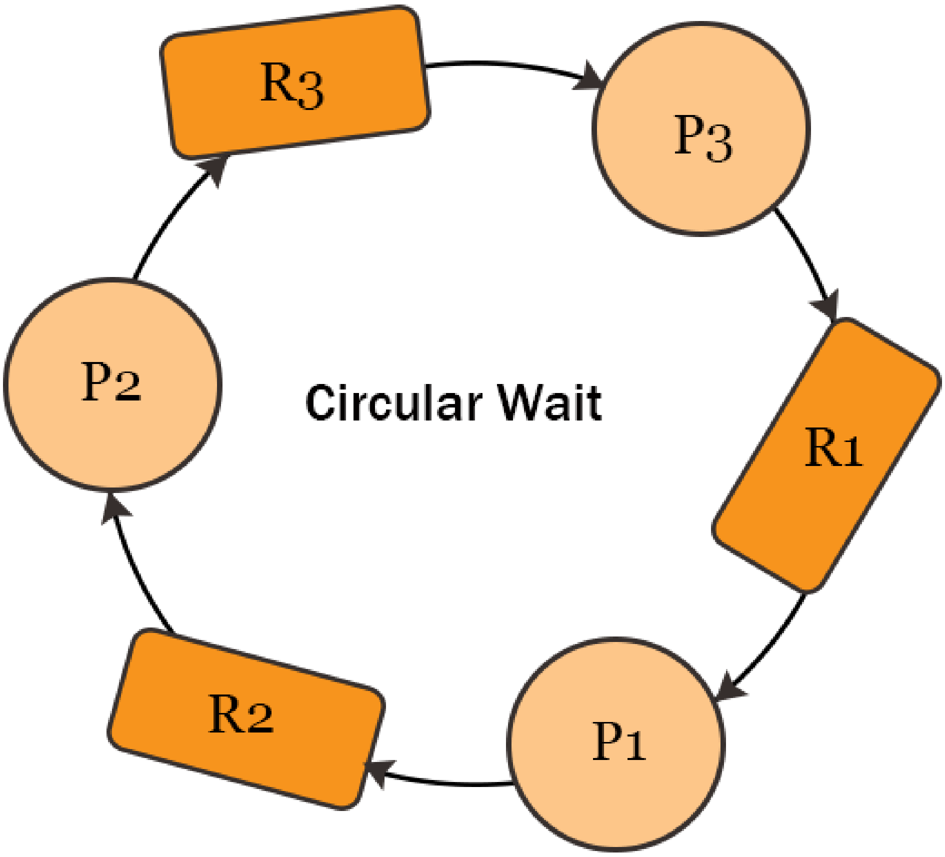
Key Characteristics:

- Involves two or more processes
- Each process holds at least one resource
- Each process requests additional resources held by others
- Circular waiting exists

Example Scenario:

- Process A holds Resource 1 and requests Resource 2
- Process B holds Resource 2 and requests Resource 1
- Neither can proceed without the other releasing resources

Diagram



P1 is waiting for P2 to release R2 , P2 is waiting for P3 to release R3 and P3 is waiting for P1 to release R1

Deadlock Characterization

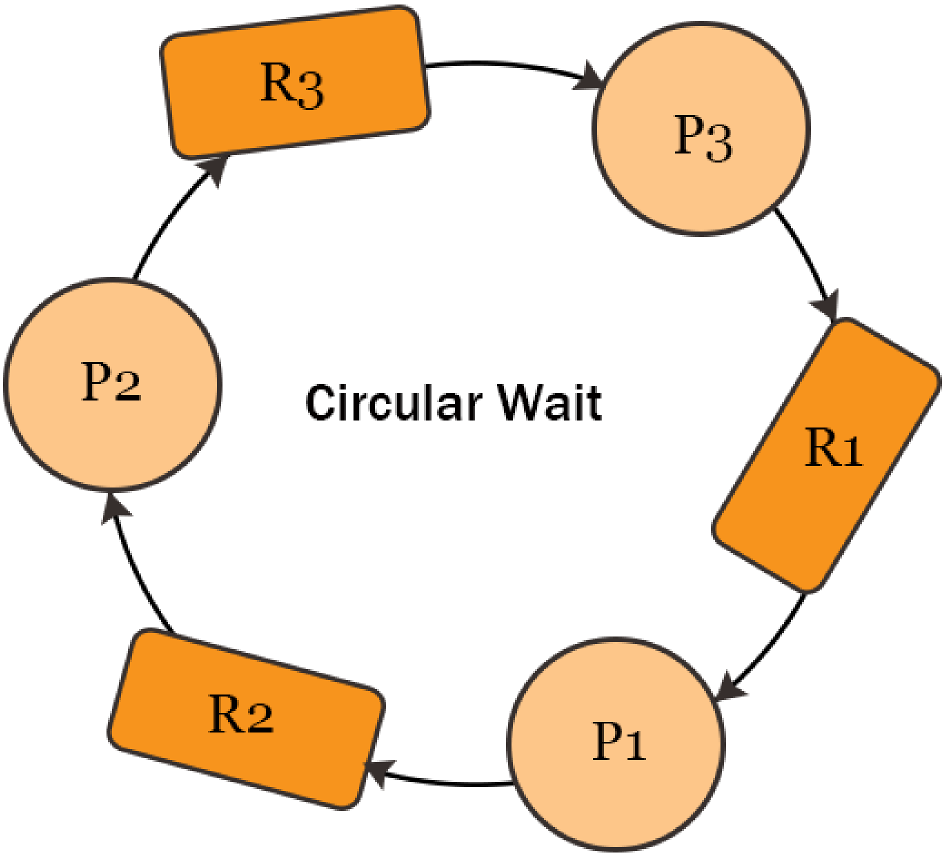
Four Necessary Conditions (Coffman Conditions)

For deadlock to occur, **all** of these conditions must hold simultaneously:

1. **Mutual Exclusion:**
    - Resources are non-sharable
    - Only one process can use a resource at a time
    - Example: Printer can't be simultaneously used by multiple processes
2. **Hold and Wait:**
    - Processes hold resources while waiting for others
    - Example: Process holds scanner while waiting for printer
3. **No Preemption:**
    - Resources cannot be forcibly taken from processes
    - Must be voluntarily released
    - Example: Process won't give up allocated memory until completion
4. **Circular Wait:**
    - Circular chain of processes exists
    - Each waits for resource held by next in chain
    - Example: P1 waits for P2's resource, P2 waits for P1's resource

Important Note: All four conditions must be present for deadlock. Preventing any one condition prevents deadlock.

Diagram



P1 is waiting for P2 to release R2 , P2 is waiting for P3 to release R3 and P3 is waiting for P1 to release R1

Preemptable vs Non-preemptable Resources

Preemptable Resources

Definition: Resources that can be taken away from a process without causing failure.

Characteristics:

- Can be reallocated to other processes
- Process can resume later without problems
- Typically don't cause deadlocks

Examples:

- CPU cycles (via context switching)
- Main memory (via swapping/paging)
- Network bandwidth

Advantages:

- Flexible resource management
- Helps prevent deadlocks
- Enables fair sharing

Disadvantages:

- Overhead from context switching
- Complex implementation

Non-preemptable Resources

Definition: Resources that cannot be taken away without causing process failure.

Characteristics:

- Must be voluntarily released
- Critical to process operation
- Primary cause of deadlocks

Examples:

- Printers
- Tape drives
- Database records
- Specialized hardware

Advantages:

- Ensures process consistency
- Prevents data corruption

Disadvantages:

- Potential for deadlocks
- May cause resource starvation

Comparison Table:

Feature	Preemptable	Non-preemptable
Can be taken away	Yes	No
Causes deadlock?	Rarely	Commonly
Examples	CPU, Memory	Printer, Database locks
Allocation flexibility	High	Low
Implementation complexity	Moderate	Simple

Resource-Allocation Graph (RAG)

Graph Representation

A directed graph used to model resource allocation state:

Components:

- **Process Nodes (Circles):** Represent processes
- **Resource Nodes (Rectangles):** Represent resource types
  - Dots inside represent instances
- **Request Edges (P→R):** Process requesting resource
- **Assignment Edges (R→P):** Resource assigned to process

Graph Rules:

1. Single instance resources: One dot in rectangle
2. Multiple instance resources: Multiple dots in rectangle
3. Edges show current allocations and requests

Deadlock Detection Using RAG

1. **No Cycles:** System is deadlock-free
2. **Cycle with Single Instance Resources:** Deadlock exists
3. **Cycle with Multiple Instance Resources:** Possible deadlock (needs further analysis)

Example Scenario:

- Process P1 holds R1 and requests R2
- Process P2 holds R2 and requests R1
- Graph shows cycle: P1→R2→P2→R1→P1

Graph Analysis Steps:

1. Draw all processes and resources
2. Add assignment edges (resources to processes)
3. Add request edges (processes to resources)
4. Check for cycles

Advantages of RAG:

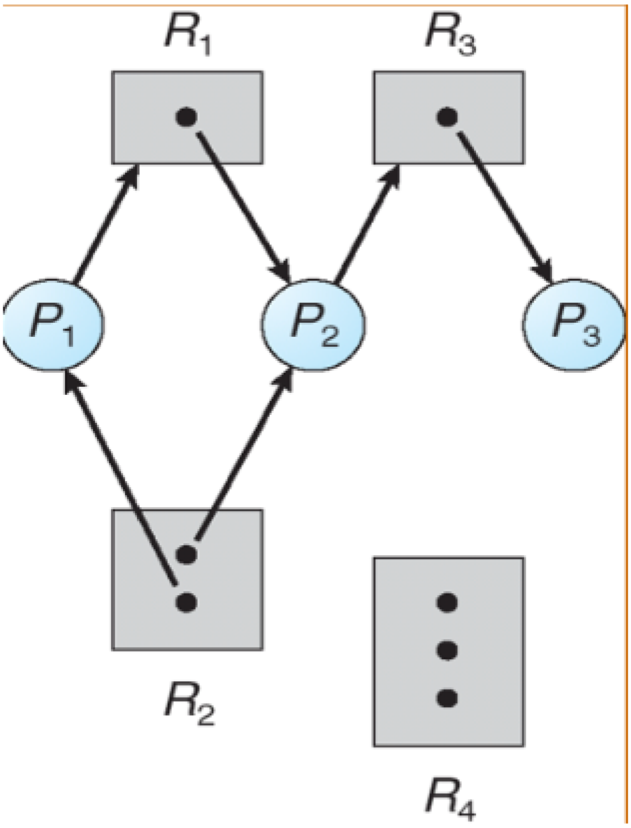
- Visual representation of system state
- Simple deadlock detection for single instances
- Useful for teaching concepts

Disadvantages of RAG:

- Becomes complex with many processes/resources
- Less effective for multiple instance resources
- Doesn't show future requests

Diagrams showing Multiple RAG examples

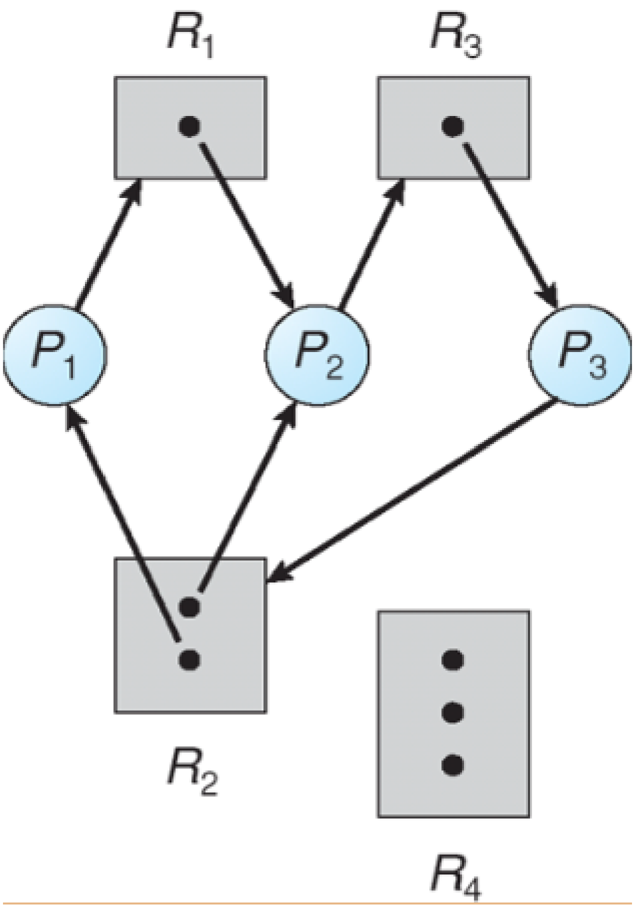
1. Deadlock-free scenario



In the above figure:

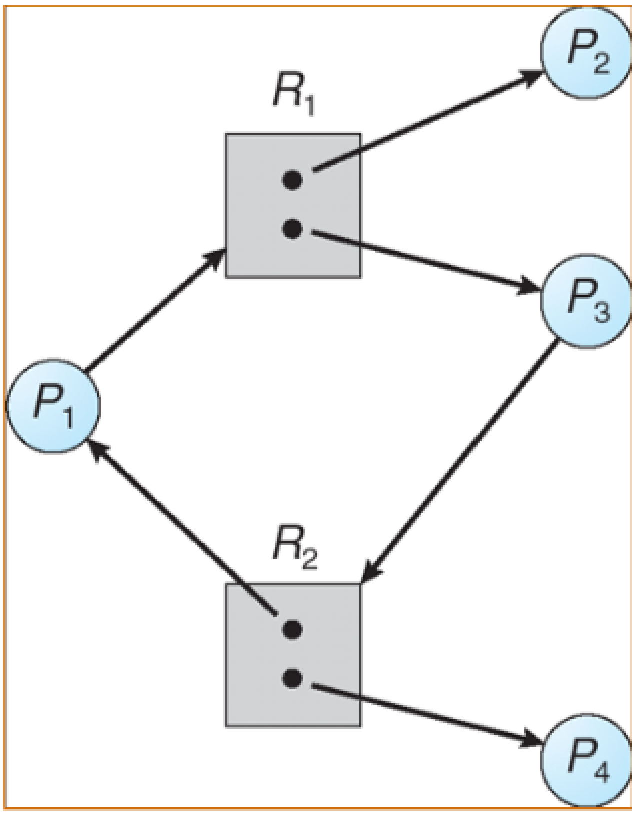
- P1 is holding an instance of R2 (R2 -> P2) and requesting or waiting for resource R1 (P1 -> R1).
- Process P2 is holding an instance of R2 (R2 -> P2), an instance of R1 (R1 -> P2) and requesting a resource R3 (P2 R3).
- Process P3 is holding an instance of R3 (R3 -> P3).

2. Deadlock scenario in cycle



- P1->R1->P2->R3->P3->R2->P1
- P2->R3->P3->R2->P2
- Processes P1, P2, P3 are deadlocked because P2 is waiting for the resource R3 which is already allocated to P3.
- Similarly, P3 is waiting for either P1 or P2 to release R2 and P1 is waiting for R2.

3. Deadlock-free scenario in cycle



- Above figure shows the graph with a cycle but no deadlock.
- If the graph does not contain any cycle the process is not deadlock.
- Even if there is a cycle, there might not be a chance of deadlock if the resource contains multiple instances.
- Here, cycle exist in the system but does not contains any deadlock:
- P1->R1->P3->R2->P1
- P4 may release its instance of resource type R2. Such resources can be allocated to P3 breaking the cycle.

Conditions for Deadlock

Detailed Examination of Each Condition

1. **Mutual Exclusion**
    - **Implementation Level:** Kernel enforces exclusive access
    - **Example:** File locks prevent concurrent writes
    - **Prevention Approach:** Use shareable resources where possible
2. **Hold and Wait**
    - **Occurrence Patterns:**
        - Process holds resource A
        - While blocked waiting for resource B
    - **Prevention Approach:**

- Require processes to request all resources at start
- Allow resource requests only when holding none

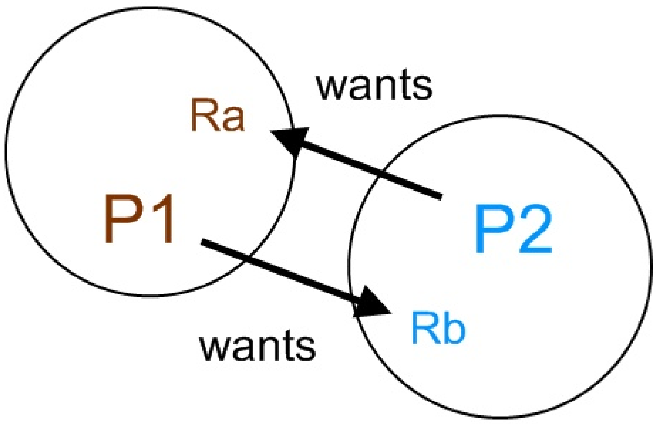
3. **No Preemption**

    - **System Impact:**
        - Resources can't be forcibly reclaimed
        - Processes retain resources until done
    - **Prevention Approach:**
        - Implement preemptable resources
        - Allow resource stealing with state saving

4. **Circular Wait**

    - **Detection Methods:**
        - Resource allocation graph analysis
        - Wait-for graphs
    - **Prevention Approach:**
        - Impose total ordering on resource types
        - Require processes request resources in order

Diagram



Summary

1. **Deadlock Definition:** Circular wait with all four conditions
2. **Resource Types:** Preemptable vs non-preemptable
3. **RAG Interpretation:** Cycle detection methods
4. **Condition Prevention:** Techniques for each condition

Handling Deadlocks

3.2 Handling Deadlocks

1. Ostrich Algorithm (Deadlock Ignorance)

Definition: A policy of deliberately ignoring deadlocks, based on the assumption that they occur rarely and the cost of prevention/detection outweighs the impact.

Implementation Approaches:

- **Unix/Linux Approach:** No deadlock handling for user processes
- **Windows Approach:** Limited deadlock detection in kernel objects

When Used:

1. Deadlocks are extremely rare
2. System can tolerate occasional deadlocks
3. Prevention costs are prohibitive

Advantages:

- No runtime overhead
- Simple implementation
- Suitable for general-purpose OS

Disadvantages:

- Unacceptable for critical systems
- May require manual intervention
- Can lead to resource starvation

Example Scenario:

- Desktop OS allows user processes to deadlock
- User resolves by manually killing processes

---

2. Deadlock Prevention

Strategy: Design system to eliminate **at least one** of the four necessary conditions.

a. Preventing Mutual Exclusion

Approach: Make resources shareable when possible

Implementation:

- Read-only files can be shared
- Spooling for printers
- Copy-on-write techniques

Limitations:

- Not all resources can be made shareable
- Example: Write operations require exclusivity

b. Preventing Hold and Wait

Approach 1: Require processes to request **all** resources at start

- **Advantage:** No partial allocations
- **Disadvantage:** Poor resource utilization

Approach 2: Allow requests only when holding no resources

- **Advantage:** Breaks circular possibilities
- **Disadvantage:** May cause starvation

Example: Database transaction requiring all locks upfront

c. Preventing No Preemption

Approach: Allow resource preemption

1. If a process can't get all resources, it releases held resources
2. Process restarts when all resources available

Implementation:

- CPU scheduling (preemptive)
- Virtual memory page reclaiming

Limitations:

- Not applicable to all resources (e.g., printer output)
- Complex state saving required

d. Preventing Circular Wait

Approach: Impose total ordering of resource types

- Processes must request resources in numerical order
- No process can request lower-numbered resource while holding higher

Example: Resource types ordered:

1. Scanner (R1)
2. Printer (R2)
3. Tape Drive (R3)

Processes must request in order R1→R2→R3

Advantages:

- Guarantees no circular waits
- Easy to implement

Disadvantages:

- Restricts programming flexibility
- May force unnecessary resource acquisition

---

3. Deadlock Avoidance

Strategy: Dynamically assess whether granting a resource request could lead to deadlock.

Banker's Algorithm (Dijkstra's Algorithm)

Key Concepts:

- **Max Demand:** Maximum resources a process may request
- **Allocation:** Currently assigned resources
- **Available:** Unassigned resources
- **Need:** Max Demand - Allocation

Safety Algorithm Steps:

- This algorithm finds out whether the system is in safe state or not. This algorithm can be described as:

- Step 1: **Need matrix = max – allocation**

- Step 2:

  - if (**need < = available**){
    - Execute process;
    - **New Available Resources= Available Resources + allocation** }

  - else{ Do not execute go forward }

Resource Request Algorithm:

- It determines whether requests can be safely granted or not.

- Step 1: if **request < = need** then go to step 2

  - Else error

- Step 2: if **request < = available**, go to step 3

  - Else wait

- Step 3:

  - **Available = available – request**
  - **Allocation = allocation + request**
  - **Need = need – request**

Advantages:

- More flexible than prevention
- Allows higher resource utilization

Disadvantages:

- Requires advance knowledge of max needs
- High computational overhead
- Doesn't work well with dynamic requests

Problem Statement

How starvation differs from deadlock? Consider the following situation of processes and resources:

Process	Currently Holds	Maximum Required
P1	2	6
P2	1	5
P3	2	5
P4	2	6

- a. What will happen if process P3 requests 1 resource?
- b. What will happen if process P4 requests 1 resource?

Solution:

Given the current system state:

Process	Has (number of resources)	Max (number of resources)
P1	2	6
P2	1	5
P3	2	5
P4	2	6

Total Resources in System: Not explicitly given, but can be calculated as:

- **Total Allocated =** 2 (P1) + 1 (P2) + 2 (P3) + 2 (P4) = **7 resources**
- Assume **Total Resources = 10 resources** (common practice when not specified)
- **Available = Total - Allocated = 10 - 7 = 3 resources**

Starvation vs Deadlock

- **Deadlock:** Circular waiting where processes block each other permanently
- **Starvation:** Process waits indefinitely due to unfair resource allocation

Banker's Algorithm Components

1. **Need = Max - Has** (remaining resources each process may request)
2. **Safety Algorithm:** Checks if system can allocate resources without deadlock
3. **Resource-Request Algorithm:** Evaluates if a specific request can be granted safely

Part (a): P3 Requests 1 Resource

Step 1: Calculate Initial Needs

Process	Has	Max	Need (Max - Has)
P1	2	6	4
P2	1	5	4
P3	2	5	3
P4	2	6	4

Step 2: Evaluate P3's Request

- **Request = 1**
- Check if request ≤ Need (1 ≤ 3) → **Valid**
- Check if request ≤ Available (1 ≤ 3) → **Valid**

Step 3: Pretend to Allocate

- **New Allocation:**
  - P3: Has = 2 + 1 = 3 resources
  - Available = 3 - 1 = 2 resources
- **Updated Need Table:**
  - P3: Need = 5 - 3 = 2 resources

Step 4: Safety Check Find a safe sequence where all processes can complete:

1. **Work = Available Resources = 2**
   - Compare with Needs:
     - P1: Need=4 > 2 → Cannot run
     - P2: Need=4 > 2 → Cannot run
     - P3: Need=2 ≤ 2 → **Can run**
     - P4: Need=4 > 2 → Cannot run
   - **Execute P3:**
     - Work(New available) = 2 (available) + 3 (P3's allocation) = 5
2. **Work = Available Resources = 5**
   - Check remaining processes:
     - P1: Need=4 ≤ 5 → **Can run**
     - P2: Need=4 ≤ 5 → **Can run**
     - P4: Need=4 ≤ 5 → **Can run**
   - Arbitrarily choose **P1**:
     - Work = 5 + 2 = 7
3. **Work = Available Resources = 7**
   - Remaining processes:
     - P2: Need=4 ≤ 7 → **Can run**
     - P4: Need=4 ≤ 7 → **Can run**
   - Choose **P2**:
     - Work = 7 + 1 = 8

4. **Work = Available Resources = 8**
    - Only P4 remains:
        - P4: Need=4 ≤ 8 → **Can run**
        - Work = 8 + 2 = 10

Safe Sequence Found: P3 → P1 → P2 → P4
Conclusion: Request can be granted safely.

Part (b): P4 Requests 1 Resource

Step 1: Use Original Need Table (Before Part a's Allocation)

Process	Need
P1	4
P2	4
P3	3
P4	4

Step 2: Evaluate P4's Request

- **Request = 1**
- Check if request ≤ Need (1 ≤ 4) → **Valid**
- Check if request ≤ Available (1 ≤ 3) → **Valid**

Step 3: Pretend to Allocate

- **New Allocation:**
    - P4: Has = 2 + 1 = 3 resources
    - Available = 3 - 1 = 2 resources
- **Updated Need Table:**
    - P4: Need = 6 - 3 = 3 resources

Step 4: Safety Check Attempt to find a safe sequence:

1. **Work = Available Resources= 2**
    - Compare with Needs:
        - P1: Need=4 > 2 → Cannot run
        - P2: Need=4 > 2 → Cannot run
        - P3: Need=3 > 2 → Cannot run
        - P4: Need=3 > 2 → Cannot run
    - **No process can run!**

Deadlock Imminent: No safe sequence exists.
Conclusion: Request **cannot** be granted (would lead to deadlock).

---

4. Deadlock Detection

For Single Instance Resources

Method: Wait-for graph

- Nodes represent processes
- Edge P1→P2 means P1 waits for resource held by P2
- Deadlock exists if cycle detected

Algorithm:

1. Construct wait-for graph
2. Periodically check for cycles
3. If cycle found, invoke recovery

Example:

- P1 → P2 → P3 → P1 indicates deadlock

For Multiple Instance Resources

Method: Modified Banker's algorithm approach

1. Initialize Work = Available
2. Find process where Need ≤ Work
3. Add its Allocation to Work
4. Mark as finished
5. Repeat until all finished (no deadlock) or none remain (deadlock)

Detection Frequency Tradeoffs:

- Frequent checks: High overhead
- Infrequent checks: Long deadlock durations

---

5. Recovery From Deadlock

Process Termination

Approach 1: Abort all deadlocked processes

- **Advantage:** Guaranteed resolution
- **Disadvantage:** All work lost

Approach 2: Abort one process at a time

- **Advantage:** Minimal work lost
- **Disadvantage:** Multiple detection attempts needed

Selection Criteria:

1. Process priority
2. Computation time used
3. Resources held
4. Interactive vs batch

Resource Preemption

Steps:

1. Select victim process/resource
2. Rollback process to safe state
3. Allocate resource to waiting process
4. Restart victim later

Challenges:

- Selecting victim (cost minimization)
- Rollback implementation
- Starvation prevention

Rollback Techniques:

- **Total Rollback:** Restart process from beginning
- **Partial Rollback:** Return to predefined checkpoint

Example: Database transaction rollback using logs

Summary

Key Comparison Table

Method	Principle	Advantages	Disadvantages	When Used
Ostrich	Ignore	No overhead	Unreliable	General-purpose OS
Prevention	Eliminate conditions	Guaranteed safety	Reduced flexibility	Critical systems
Avoidance	Safe state checks	Balanced approach	Needs advance info	Medium-criticality
Detection	Periodic checks	Flexible	Recovery needed	Systems with tolerance

---