CS3701 - Operating Systems

Chapter 6 - Deadlocks

Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY - 2025/2026

Prepared by Dr. Mahdi Khemakhem, Reviewed by Dr. Essra Aldessouky
Updated on January 22, 2026

Outline

1. Introduction to Deadlocks
   - 1.1 Overview
   - 1.2 System Model
   - 1.3 Deadlock Characterization
2. Resource-Allocation Graph
   - 2.1 RAG Components
   - 2.2 RAG Deadlock Analysis - Basic Facts
   - 2.3 RAG Examples Analysis
   - 2.4 Key Takeaways
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
   - 5.1 Overview
   - 5.2 Safe State Concept
   - 5.3 Resource-Allocation Graph Algorithm
   - 5.4 Banker's Algorithm
   - 5.5 Key Takeaways
6. Deadlock Detection and Recovery
   - 6.1 Detection for Single Instance (Wait-For Graph)

Outline (Contd.)

6.2 Detection for Multiple Instances (Banker's-like Algorithm)
6.3 Recovery from Deadlock
6.4 Key Takeaways

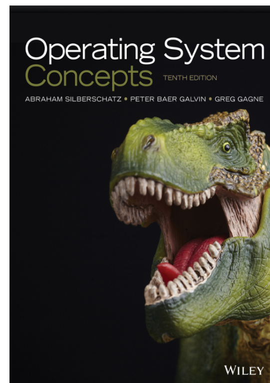
7. Exercise

Textbook Reference

Required Reading

Refer to Chapter 8 of the textbook:

- **Title:** *Operating System Concepts*
- **Authors:** Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
- **Publisher:** John Wiley & Sons
- **Edition:** 10th Edition (2018)



Outline

1. Introduction to Deadlocks
   1.1 Overview
   1.2 System Model
   1.3 Deadlock Characterization

2. Resource-Allocation Graph

3. Methods for Handling Deadlocks

4. Deadlock Prevention

5. Deadlock Avoidance

6. Deadlock Detection and Recovery

7. Exercise

Deadlocks Definition

Deadlock

- A deadlock is a situation in which a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
- In a deadlock state, no process can proceed because each is waiting for an event that can only be caused by another waiting process.

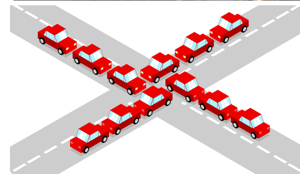


Figure 1: Deadlock Illustration

Real-World Deadlock Example

Database Transaction Deadlock

Two transactions accessing bank accounts:

Transaction T1

1. Lock Account A
2. Wait for Account B
3. Transfer $100 from A to B
4. Release locks

Transaction T2

1. Lock Account B
2. Wait for Account A
3. Transfer $50 from B to A
4. Release locks

⇒**Result:** T1 holds A and waits for B, T2 holds B and waits for A

⇒ **Deadlock! Neither can proceed.**

System Model

System Model

A system consists of:

- **Processes:** $P_1, P_2, \ldots, P_n$
- **Resource Types:** $R_1, R_2, \ldots, R_m$

- **Resource types**: $R_1, R_2, \ldots, R_m$, e.g., CPU cycles, memory space, I/O devices, Files, database records, semaphores, etc.
- **Each resource type R_i has W_i instances**, e.g., multiple CPUs, multiple I/O devices
- Each process utilizes a resource as follows:
  1. **Request**: Process requests the resource type. If request cannot be granted, process must **wait**
  2. **Use**: Process operates on the resource
  3. **Release**: Process releases the resource

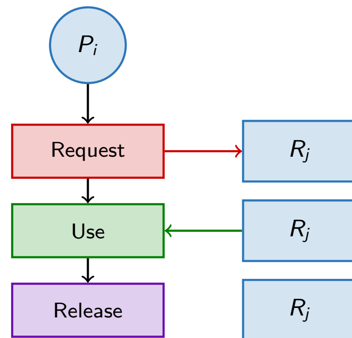


Figure 2: Process Resource Utilization

Necessary Conditions for Deadlock

Four Necessary Conditions

Deadlock can arise if ALL FOUR conditions hold simultaneously. These conditions are not independent, but are necessary:

1. **Mutual Exclusion**
   - At least one resource must be held in a **non-sharable mode**
   - Only one process can use the resource at a time
2. **Hold and Wait**
   - At least one process must be **holding** one resource and **waiting** to acquire additional resources held by other processes
3. **No Preemption**
   - Resources cannot be **forcibly taken** from a process
   - Only released **voluntarily** by the holding process
4. **Circular Wait**
   - Set $\{P_0, P_1, \ldots, P_n\}$ where:
   - P_0 waits for resource held by P_1, P_1 waits for resource held by P_2, ..., P_n waits for resource held by P_0

⇒ **Important: All four conditions must hold simultaneously. Breaking any one prevents deadlock.**

Illustrating the Four Conditions

Example: Communication ports

System has two Communication Ports and two processes: P_1 and P_2

Timeline of Events

1. P_1 requests and gets communication port 1
2. P_2 requests and gets communication port 2
3. P_1 requests communication port 2 (blocked)
4. P_2 requests communication port 1 (blocked)

Four Conditions Met

- **Mutual Exclusion**: Only one process per port, i.e. non-sharable ports
- **Hold and Wait**: P_1 holds port 1, waits for port 2
- **No Preemption**: Cannot force release
- **Circular Wait**: P_1 waits for port 2 held by P_2; P_2 waits for port 1 held by P_1

⇒**Result: DEADLOCK!**

Outline

1. Introduction to Deadlocks

2. Resource-Allocation Graph
2.1 RAG Components
2.2 RAG Deadlock Analysis - Basic Facts
2.3 RAG Examples Analysis
2.4 Key Takeaways

3. Methods for Handling Deadlocks

4. Deadlock Prevention

5. Deadlock Avoidance

6. Deadlock Detection and Recovery

7. Exercises

Resource Allocation Graph (RAG)

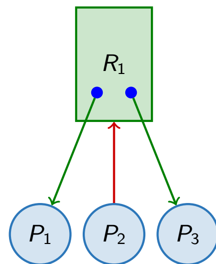
Resource Allocation Graph

A system can be described by a **Resource Allocation Graph (RAG)**: a **directed graph** with **two types of vertices (node) sets** (P and R) and **two types of edges** (**request** and **assignment**).

RAG Components

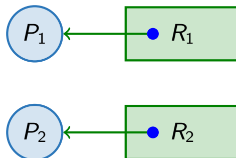
- **Vertices**:
  - $P = \{P_1, \ldots, P_n\}$: **Processes (circles)**
  - $R = \{R_1, \ldots, R_m\}$: **Resources (rectangles). Each resource type R_j has W_j instances (dots inside rectangle)**
- **Edges**:
  - **Request: $P_i \rightarrow R_j$ (process requests resource)**
  - **Assignment: $R_j \rightarrow P_i$ (resource instance allocated to process)**

$\Rightarrow$ **Deadlocks can be analyzed using RAGs! (see next slides)**



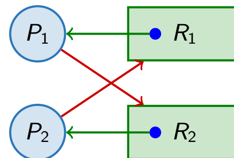
Deadlock Analysis Using RAG

No Cycle - No Deadlock



No cycle ⇒ No deadlock

Cycle - Deadlock

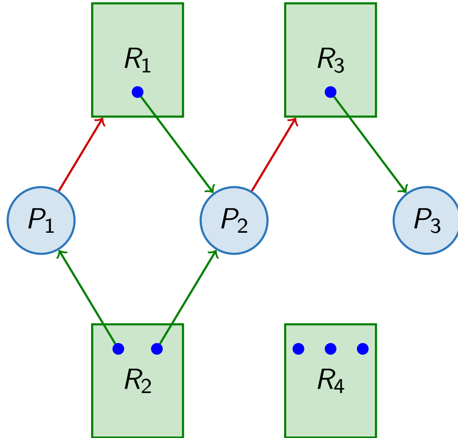


Cycle ⇒ Deadlock!

RAG Theorems (Basic Facts)

- **No cycles ⇒ No deadlock**
- **Cycle + single instance per resource ⇒ Deadlock**
- **Cycle + multiple instances ⇒ Possible deadlock**

RAG Example 1: No Deadlock

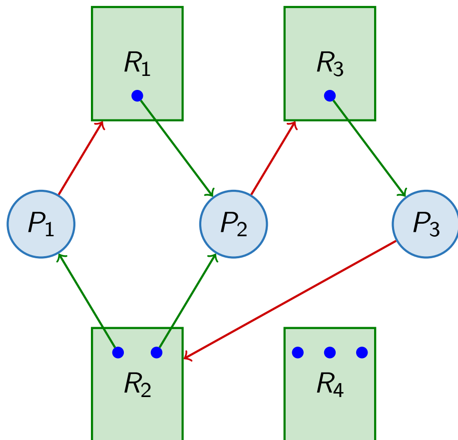


Analysis

- P_1 holds instance from R_2, requests $R_1 \Rightarrow$ **Waiting** state
- P_2 holds instances from R_1 and R_2, requests $R_3 \Rightarrow$ **Waiting** state
- P_3 holds instance from $R_3 \Rightarrow$ **Running/Ready** state
- **No cycle exists**

$\Rightarrow$ **No Deadlock!** All processes can complete.

RAG Example 2: Deadlock Exists



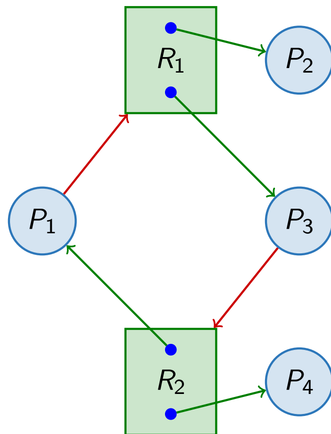
Analysis

- P_1 holds instance from R_2, requests $R_1 \Rightarrow$ **Waiting** state
- P_2 holds instances from R_1 and R_2, requests $R_3 \Rightarrow$ **Waiting** state
- P_3 holds instance from R_3, requests $R_2 \Rightarrow$ **Waiting** state
- **Two embedded cycles exist**:
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

$\Rightarrow$ **Deadlock Exists!** None can complete.

$\Rightarrow$ **Deadlock despite multiple instances of R_2! All instances are held.**

RAG Example 3: Cycle but No Deadlock



Analysis

- P_1 holds instance from R_2, requests R_1 $\Rightarrow$ **Waiting** state
- P_2 holds instance from R_1 $\Rightarrow$ **Running/Ready** state
- P_3 holds instance from R_1, requests R_2 $\Rightarrow$ **Waiting** state
- P_4 holds instance from R_2 $\Rightarrow$ **Running/Ready** state
- **Cycle exists**:
  - $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$\Rightarrow$ **No Deadlock!** P_2 and P_4 can complete, releasing resources for P_1 and P_3.

Key Takeaways

Key Takeaways

- **Deadlocks occur** when four necessary conditions hold simultaneously: **Mutual Exclusion**, **Hold and Wait**, **No Preemption**, and **Circular Wait**.
- A **Resource Allocation Graph (RAG)** models the state of resource allocation in a system using **processes** and **resources** vertices, and **request/assignment** edges.
- Deadlock analysis using RAGs:
  - **No cycles** in RAG $\Rightarrow$ **No deadlock**
  - **Cycle + single instance per resource** $\Rightarrow$ **Deadlock**
  - **Cycle + multiple instances** $\Rightarrow$ **Possible deadlock**

Outline

1. Introduction to Deadlocks
2. Resource-Allocation Graph
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection and Recovery
7. Exercise

Strategies for Handling Deadlocks

Key Strategies

Three main strategies to handle deadlocks:

- **Ignore** the problem altogether and **pretend** deadlocks never occur in the system (*no mechanism for this strategy*)
- Use a protocol to **prevent** or **avoid** deadlocks from occurring (**prevention** *and* **avoidance** *methods discussed in next sections*)
- **Allow** deadlocks to occur, **detect** them, and **recover** from them (**detection** *and* **recovery** *methods discussed in later sections*)

Methods for Handling Deadlocks

Three Main Approaches

1. **Deadlock Prevention**
   - **Ensure at least one of the four conditions cannot hold**
   - **Prevent deadlock from ever occurring**
   - **May reduce resource utilization**
2. **Deadlock Avoidance**
   - **Requires advance information about resource requests**
   - **System decides whether to grant or delay requests**
   - Uses **Banker's Algorithm**
3. **Detection and Recovery**
   - **Allow deadlocks to occur**
   - **Detect** when deadlock happens (using **Wait-For Graphs** or **Banker's-like algorithms**)
   - **Recover** from deadlock

⇒**Trade-offs:** **Each method has different costs, complexity, and performance implications.**

Outline

1. Introduction to Deadlocks
2. Resource-Allocation Graph
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection and Recovery
7. Exercise

Deadlock Prevention

Prevention Strategy

Ensure at least one of the four necessary conditions cannot hold to prevent deadlocks from occurring.

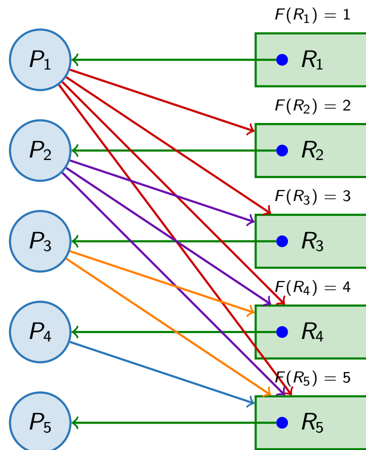
1. Breaking **"Mutual Exclusion"**(Mutual Inclusion)
   - Some resources are **shareable** (e.g., read-only files, printers with spooling)
   - **Disadvantage**: Not always possible
2. Breaking **"Hold and Wait"**
   - **Protocol 1**: Request all resources at once
   - **Protocol 2**: Release all before requesting new ones
   - **Disadvantage**: Low utilization, starvation
3. Breaking **"No Preemption"**
   - If request fails, release all held resources
   - **Disadvantage**: Not always possible (e.g., printers, tapes)
4. **Circular Wait**
   - Impose total ordering: $R_1, R_2, \ldots, R_m$
   - Request resources in **increasing order**
   - **Disadvantage**: May be difficult to define ordering

Prevention Example: Resource Ordering

Breaking Circular Wait

Resource Ordering Strategy

- Assign a unique integer $F(R_j)$ to each resource type R_j in order define a hirarchical ordering.
- Processes P_i holding resource instance R_j can request resource instance R_k only if $F(R_j) > F(R_k)$.



⇒ Deadlock cannot occur as circular wait is prevented!

Note: The ordering integer $F(R_j)$ can be assigned according to the demand rate history of resource R_j (lower demand → lower $F(R_j)$).

Key Takeaways

Key Takeaways

- **Deadlock Prevention** ensures that at least one of the four necessary conditions for deadlock cannot hold, effectively preventing deadlocks from occurring.
- Methods to break each condition:
  - **Mutual Exclusion**: Make resources shareable when possible.
  - **Hold and Wait**: Require processes to request all resources at once or release held resources before requesting new ones.
  - **No Preemption**: Allow preemption of resources when requests cannot be granted.
  - **Circular Wait**: Impose a total ordering on resource types and require processes to request resources in increasing order.
- $\Rightarrow$ **Trade-offs**: While prevention methods can effectively eliminate deadlocks, they may lead to reduced resource utilization and increased complexity in resource management.

Outline

1. Introduction to Deadlocks

2. Resource-Allocation Graph

3. Methods for Handling Deadlocks

4. Deadlock Prevention

5. Deadlock Avoidance
5.1 Overview
5.2 Safe State Concept
5.3 Resource-Allocation Graph Algorithm
5.4 Banker's Algorithm
5.5 Key Takeaways

6. Deadlock Detection and Recovery

Deadlock Avoidance: Overview

Avoidance Approach

Requires that the system has some additional a priori information available

Key Requirements

- **Simplest model**: Each process declares the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm **dynamically examines** the resource-allocation state
- Ensures that there can never be a **circular-wait condition**
- **Resource-allocation state** defined by:
  - **Number of available and allocated resources**
  - **Maximum demands of the processes**

Two Algorithms

1. **Resource-Allocation Graph Algorithm**: For single instance per resource type
2. **Banker's Algorithm**: For multiple instances per resource type

Safe State Definition

Safe State

System is in a safe state if there exists a safe sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of ALL processes such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all P_j with $j < i$.

Safe Sequence Properties

- If P_i resource needs are not immediately available, then P_i can **wait** until all P_j ($j < i$) have finished
- When P_j finishes, P_i can **obtain needed resources**, **execute**, **return allocated resources**, and **terminate**
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on...

$\Rightarrow$ **Important: If no such sequence exists, the system is in an unsafe state.**

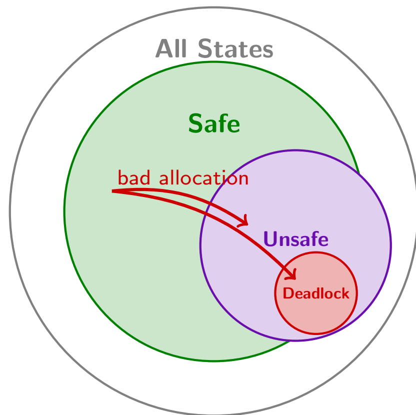
Safe vs. Unsafe vs. Deadlock States

Basic Facts

- Safe state ⇒ **No deadlock**
- Unsafe state ⇒ **Possibility of deadlock**
- Deadlock state ⇒ **Deadlock**

Avoidance Strategy

- **Ensure** that the system will **never enter** an **unsafe state**
- When a process requests a resource, **ask**: *"If we grant this request, will the system remain in a safe state?"*
  - If yes ⇒ **Grant**
  - If no ⇒ **Deny/Delay**



Goal: Stay in safe region!

Safe State: Illustration Example

System with 12 Resources

3 processes: P_0, P_1, P_2; Total: 12 resource instances

At time t_0

Process	Max	Allocated	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
Available			3

- P_1: needs ≤ 2 (available: 3) ✓
- P_0: needs ≤ 5 (available: 5) ✓
- P_2: needs ≤ 7 (available: 10) ✓

Safe sequence: $\langle P_1, P_0, P_2 \rangle$
$\Rightarrow$ **Safe state!**

At time t_1: P_2 requests 1 more

Process	Max	Current	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6
Available			2

Analysis:

- Only P_1 can complete (needs ≤ 2)
- After P_1: Available $= 4$
- P_0 needs 5 **(cannot satisfy!)**
- P_2 needs 6 **(cannot satisfy!)**

$\Rightarrow$ **Unsafe state!**

RAG Algorithm for Avoidance

For Single Instance per Resource Type

Use Resource-Allocation Graph with Claim Edges

New Edge Type

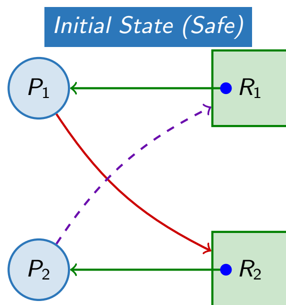
- **Claim edge $P_i \dashrightarrow R_j$:** Process P_i **may request** resource R_j in the future (represented by **dashed line**)
- **Claim edge** converts to **request edge** when process requests the resource
- **Request edge** converts to **assignment edge** when resource is allocated
- **Assignment edge** reconverts to **claim edge** when resource is released

Safety Check Algorithm

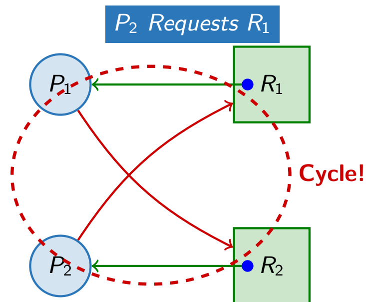
Suppose process P_i requests resource R_j:

- Request can be granted **only if** converting the **claim edge** $P_i \dashrightarrow R_j$ to a **request edge** $P_i \rightarrow R_j$ **does not create a cycle** in the RAG
- If cycle would form $\Rightarrow$ **Deny request** (unsafe state)

RAG Avoidance: Example



P_1 holds R_1, requests R_2
P_2 holds R_2, may claim R_1
$\Rightarrow$ **Safe state (no cycle)**



Claim --→ Request: Forms cycle!
Cycle: $P_2 \to R_1 \to P_1 \to R_2 \to P_2$
$\Rightarrow$ **Deny request**

$\Rightarrow$**Decision:** System **denies** P_2's request for R_1 to avoid entering unsafe state

Banker's Algorithm: Overview

For Multiple Instances per Resource Type

Use Banker's Algorithm to ensure system remains in safe state. It follows the principle of banking: *never allocate resources that may not be available in the future*.

Requirements

- Each process must **a priori claim** maximum use
- When a process requests a resource, it may have to **wait**
- When a process gets all its resources, it must **return them in finite time**

Data Structures (Let $n = \#$ processes, $m = \#$ resource types)

- **Available**[m]: Array of length m. Available[j] $= k$ means k instances of R_j available
- **Max**[n][m]: $n \times m$ matrix. Max[i][j] $= k$ means P_i may request at most k instances of R_j
- **Allocation**[n][m]: $n \times m$ matrix. Allocation[i][j] $= k$ means P_i currently holds k instances of R_j
- **Need**[n][m]: $n \times m$ matrix. Need[i][j] $= k$ means P_i may need k more instances of R_j
  - *Need[i][j] = Max[i][j] - Allocation[i][j]*

Banker's Algorithm: Safety Algorithm

Algorithm 1: Safety Algorithm - Check if system is in safe state

Input: n processes, m resource types, Available, Allocation, Need matrices
Output: Determine if system is in safe state (SAFE or UNSAFE)

```
   // Step 1: Initialize
1  Let Work and Finish be vectors of length m and n
2  Work[j] ← Available[j] for all j ∈ {0, . . . , m − 1}
3  Finish[i] ← false for all i ∈ {0, . . . , n − 1}
   // Step 2: Find process that can finish
4  repeat
5  |  Find index i ∈ {0, . . . , n − 1} such that:
6  |            (a) Finish[i] == false
7  |            (b) Need[i][j] ≤ Work[j] for all j ∈ {0, . . . , m − 1}
8  |  if such i exists then
   |  |  // Step 3: Simulate allocation
9  |  |  Work[j] ← Work[j] + Allocation[i][j] for all j ∈ {0, . . . , m − 1}
10 |  |  Finish[i]s ← true
11 |  end
12 until no such i exists;
   // Step 4: Check if all processes can finish
13 if Finish[i] == true for all i ∈ {0, . . . , n − 1} then
14 |  return SAFE
15 else
16 |  return UNSAFE
17 end
```

Banker's Algorithm: Example Setup

System with 5 Processes and 3 Resource Types

Resource Types: A (10 instances), B (5 instances), C (7 instances)

Snapshot at time T_0

	Max			**Allocation**			**Need**		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

Available		
A	B	C
3	3	2

Resource Summary

- **Need = Max - Allocation**
- **Total resources:** A = 10, B = 5, C = 7
- **Allocated resources:**
  - A: 0 + 2 + 3 + 2 + 0 = 7
  - B: 1 + 0 + 0 + 1 + 0 = 2
  - C: 0 + 0 + 2 + 1 + 2 = 5
- **Available resources:**
  - A: 10 - 7 = 3
  - B: 5 - 2 = 3
  - C: 7 - 5 = 2

Banker's Algorithm: Safety Check

Is the system in a safe state?

	Max			Allocation			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

Apply Safety Algorithm

Initialize:

- Work = Available = (3,3,2)
- Finish[i] = false for all i = [F,F,F,F,F]

Iterations:

1. P_1: Need = (1,2,2) $\leq$ (3,3,2) ✓
   - Work = (3,3,2) + (2,0,0) = **(5,3,2)**
   - Finish[1] = true → **[F,T,F,F,F]**
2. P_3: Need = (0,1,1) $\leq$ (5,3,2) ✓
   - Work = (5,3,2) + (2,1,1) = **(7,4,3)**
   - Finish[3] = true → **[F,T,F,T,F]**
3. P_4: Need = (4,3,1) $\leq$ (7,4,3) ✓
   - Work = (7,4,3) + (0,0,2) = **(7,4,5)**
   - Finish[4] = true → **[F,T,F,T,T]**
4. P_2: Need = (6,0,0) $\leq$ (7,4,5) ✓
   - Work = (7,4,5) + (3,0,2) = **(10,4,7)**
   - Finish[2] = true → **[F,T,T,T,T]**
5. P_0: Need = (7,4,3) $\leq$ (10,4,7) ✓
   - Work = (10,4,7) + (0,1,0) = **(10,5,7)**
   - Finish[0] = true → **[T,T,T,T,T]**

⇒**Result:** All Finish[i] = true
Safe sequence: $\langle P_1, P_3, P_4, P_2, P_0 \rangle$

System is in SAFE state!

Banker's Algorithm: Resource-Request Algorithm

Algorithm 2: Resource-Request Algorithm - Process P_i requests resources

Input: Request[i] = request vector for process P_i of length m
Output: Grant or deny the request

```
   // Step 1: Check if request exceeds maximum claim
1  if Request[i][j] > Need[i][j] for any j ∈ {0, ..., m − 1} then
2  |  raise error (exceeded maximum claim)
3  end
   // Step 2: Check if resources are available
4  if Request[i][j] > Available[j] for any j ∈ {0, ..., m − 1} then
5  |  P_i must WAIT (resources not available)
6  end
   // Step 3: Pretend to allocate
7  Available[j] ← Available[j] - Request[i][j] for all j ∈ {0, ..., m − 1}
8  Allocation[i][j] ← Allocation[i][j] + Request[i][j] for all j ∈ {0, ..., m − 1}
9  Need[i][j] ← Need[i][j] - Request[i][j] for all j ∈ {0, ..., m − 1}
   // Step 4: Run safety algorithm (see Algorithm 1)
10 if system is in safe state then
11 |  GRANT resources to P_i (keep changes)
12 else
13 |  DENY request, P_i must WAIT or ROLLBACK
14 |  Restore old state (undo changes)
15 end
```

Banker's Algorithm: Request Example 1

P_1 requests (1, 0, 2)

Step 1: Check against Need

- $\text{Request}_1 = (1,0,2)$, $\text{Need}_1 = (1,2,2)$
- $\text{Request}_1 \leq \text{Need}_1$? **YES** ✓

Step 2: Check against Available

- $\text{Request}_1 = (1,0,2)$, $\text{Available} = (3,3,2)$
- $\text{Request}_1 \leq \text{Available}$? **YES** ✓

Step 3: Pretend to allocate

	Max			Allocation			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	3	0	2	0	2	0
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

Available		
A	B	C
2	3	0

Step 4: Run Safety Algorithm

$\text{Work} = \text{Available} = (2,3,0)$

1. P_1: $\text{Need} = (0,2,0) \leq (2,3,0)$ ✓ $\Rightarrow \text{Work} = (5,3,2)$
2. P_3: $\text{Need} = (0,1,1) \leq (5,3,2)$ ✓ $\Rightarrow \text{Work} = (7,4,3)$
3. P_4: $\text{Need} = (4,3,1) \leq (7,4,3)$ ✓ $\Rightarrow \text{Work} = (7,4,5)$
4. P_0: $\text{Need} = (7,4,3) \leq (7,4,5)$ ✓ $\Rightarrow \text{Work} = (7,5,5)$
5. P_2: $\text{Need} = (6,0,0) \leq (7,5,5)$ ✓ $\Rightarrow \text{Work} = (10,5,7)$

$\Rightarrow$**Decision: Safe sequence** $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ exists
System remains in **SAFE state**

GRANT the request to P_1

Banker's Algorithm: Request Example 2

P_4 requests (3, 3, 0) - Apply on the original state

Step 1: Check against Need

- $\text{Request}_4 = (3,3,0)$, $\text{Need}_4 = (4,3,1)$
- $\text{Request}_4 \leq \text{Need}_4$? **YES** ✓

Step 2: Check against Available

- $\text{Request}_4 = (3,3,0)$, $\text{Available} = (3,3,2)$
- $\text{Request}_4 \leq \text{Available}$? **YES** ✓

Step 3: Pretend to allocate

	Max			Allocation			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	3	3	2	1	0	1

Available		
A	B	C
0	0	2

Step 4: Run Safety Algorithm

Work = Available = (0,0,2)

Check all processes:

- P_0: Need $= (7,4,3) \not\leq (0,0,2)$ **NO**
- P_1: Need $= (1,2,2) \not\leq (0,0,2)$ **NO**
- P_2: Need $= (6,0,0) \not\leq (0,0,2)$ **NO**
- P_3: Need $= (0,1,1) \not\leq (0,0,2)$ **NO**
- P_4: Need $= (1,0,1) \not\leq (0,0,2)$ **NO**

No process can proceed!

⇒Decision: No safe sequence exists
System would be in **UNSAFE state**

DENY the request to P_4

Banker's Algorithm: Request Example 3

P_0 requests (0, 2, 0) - Apply on the original state

Step 1: Check against Need

- $\text{Request}_0 = (0,2,0)$, $\text{Need}_0 = (7,4,3)$
- $\text{Request}_0 \leq \text{Need}_0$? **YES** ✓

Step 2: Check against Available

- $\text{Request}_0 = (0,2,0)$, $\text{Available} = (3,3,2)$
- $\text{Request}_0 \leq \text{Available}$? **YES** ✓

Step 3: Pretend to allocate

	Max			Allocation			Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	3	0	7	2	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

Available		
A	B	C
3	1	2

Step 4: Run Safety Algorithm

$\text{Work} = \text{Available} = (3,1,2)$

1. P_3: $\text{Need} = (0,1,1) \leq (3,1,2)$ ✓ $\Rightarrow \text{Work} = (5,2,3)$
2. P_1: $\text{Need} = (1,2,2) \leq (5,2,3)$ ✓ $\Rightarrow \text{Work} = (7,2,3)$
3. P_0: $\text{Need} = (7,2,3) \leq (7,2,3)$ ✓ $\Rightarrow \text{Work} = (7,5,3)$
4. P_4: $\text{Need} = (4,3,1) \leq (7,5,3)$ ✓ $\Rightarrow \text{Work} = (7,5,5)$
5. P_2: $\text{Need} = (6,0,0) \leq (7,5,5)$ ✓ $\Rightarrow \text{Work} = (10,5,7)$

⇒**Decision: Safe sequence** $\langle P_3, P_1, P_0, P_4, P_2 \rangle$ exists
System remains in **SAFE state**

GRANT the request to P_0

Key Takeaways

Key Takeaways

- **Deadlock avoidance** ensures that system will never enter an unsafe state by **dynamically** examining resource-allocation state to ensure that there can never be a circular-wait condition.
- The **Banker's Algorithm** is a deadlock-avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.
- The **safety algorithm** is used to determine whether a system is in a safe state or an unsafe state.
- When a process makes a request for resources, the **Banker's Algorithm** determines whether the request can be granted immediately or must be delayed until it is safe to do so.

Outline

1. Introduction to Deadlocks

2. Resource-Allocation Graph

3. Methods for Handling Deadlocks

4. Deadlock Prevention

5. Deadlock Avoidance

6. Deadlock Detection and Recovery

6.1 Detection for Single Instance (Wait-For Graph)

6.2 Detection for Multiple Instances (Banker's-like Algorithm)

6.3 Recovery from Deadlock

6.4 Key Takeaways

Allow-Detection-Recovery Approach: Overview

Allow-Detect-Recover Strategy

If system does **not** employ **prevention** or **avoidance** techniques, then it **must** have mechanism that **allows** system to enter **deadlock state**, provides algorithm to **detect** deadlock, and provides algorithm to **recover** from deadlock.

Key Questions

1. **When to invoke** detection algorithm?
   - Depends on how often deadlocks occur
   - Depends on how many processes affected
2. **Which algorithm** to use?
   - Single instance: **Wait-for Graph**
   - Multiple instances: **Detection Algorithm** (similar to Banker's)
3. **How to recover**?
   - Process termination
   - Resource preemption

Detection Algorithm: Single Instance per Resource Type

Wait-For Graph Construction

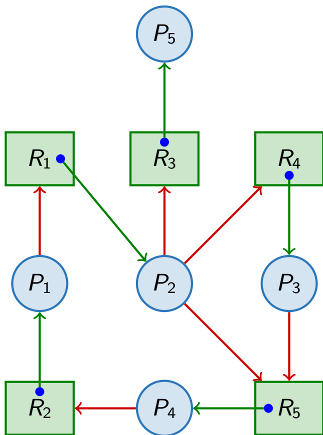
- **Nodes**: Processes only (no resource nodes)
- **Edge** $P_i \rightarrow P_j$: Process P_i is **waiting for** process P_j to release a resource
- **Conversion from RAG to Wait-for graph consists:**
  - If P_i holds resource R and P_j requests R, create edge $P_j \rightarrow P_i$
  - If multiple resources held/requested, create multiple edges

Detection Algorithm

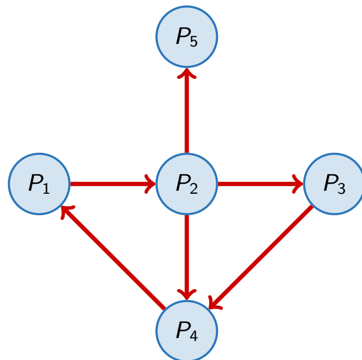
- Periodically invoke algorithm that searches for **cycle**
- If cycle exists $\Rightarrow$ **Deadlock**!
- Algorithm requires $O(n^2)$ operations, where n = number of vertices (processes)
- **Reducing RAG to Wait-for graph reduce complexity of cycle detection from $O(m+n)$ to $O(n^2)$, where m = number of edges**

Wait-For Graph: Example

Resource Allocation Graph



Wait-For Graph



$\Rightarrow$ **2 cycles can be easily detected here!**

Detection Algorithm: Multiple Instances per Resource Type

Principle

For multiple instances per resource type: use an algorithm similar to Banker's Algorithm to **detect** if system is in **deadlock state** but **without** making use of **Max** and **Need** matrices. Use only **Allocation** and **Request** matrices.

Data Structures (Let $n = \#$ processes, $m = \#$ resource types)

- **Available**$[m]$: Number of available resources of each type
- **Allocation**$[n][m]$: Resources currently allocated to each process
- **Request**$[n][m]$: Current request of each process
  - Request$[i][j] = k$ means P_i is requesting k more instances of R_j

⇒Key Difference from Banker's: Uses Request (current requests) instead of Need (maximum future needs)

Detection Algorithm: Multiple Instances per Resource Type

Algorithm 3: Detection Algorithm - Banker's-like for Multiple Instances

Input: n processes, m resource types, Available, Allocation, Request matrices
Output: Determine if deadlock exists and identify deadlocked processes

```
    // Step 1: Initialize
 1  Let Work and Finish be vectors of length m and n
 2  Work[j] ← Available[j] for all j ∈ {0, 1, ..., m − 1}
 3  Finish[i] ← false (if Allocation[i] ≠ 0) true (if Allocation[i] = 0) for all i ∈ {0, 1, ..., n − 1}
    // Step 2: Find process whose request can be satisfied
 4  repeat
 5      Find index i such that:
 6              (a) Finish[i] == false
 7              (b) Request[i][ j] ≤ Work[j] for all j ∈ {0, 1, ..., m − 1}
 8      if such i exists then
            // Step 3: Simulate release
 9          Work[j] ← Work[j] + Allocation[i][j] for all j ∈ {0, 1, ..., m − 1}
10          Finish[i] ← true
11      end
12  until no such i exists;
    // Step 4: Check for deadlock
13  if Finish[i] == false for some i then
14      return DEADLOCK exists
15      Deadlocked processes: all P_i where Finish[i] == false
16  else
17      return NO DEADLOCK
18  end
```

Detection Algorithm: Example 1 (No Deadlock)

5 processes, 3 resource types

Resources Types: A (7 instances), B (2 instances), C (6 instances)

Snapshot at time T_0

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	0
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2
Available				0	0	0

Note: Total resources = (7,2,6), Allocated = (7,2,6), Available = (0,0,0)

Step 1: Initialize

- Work = Available = (0,0,0)
- Finish = [F, F, F, F, F] (all hold resources)

Step 2-3: Find processes that can complete

1. P_0: Request = (0,0,0) $\leq$ (0,0,0) ✓
   - Work = (0,0,0) + (0,1,0) = **(0,1,0)**
   - Finish[P_0] = true → **[T, F, F, F, F]**
2. P_2: Request = (0,0,0) $\leq$ (0,1,0) ✓
   - Work = (0,1,0) + (3,0,3) = **(3,1,3)**
   - Finish[P_2] = true → **[T, F, T, F, F]**
3. P_3: Request = (1,0,0) $\leq$ (3,1,3) ✓
   - Work = (3,1,3) + (2,1,1) = **(5,2,4)**
   - Finish[P_3] = true → **[T, F, T, T, F]**

Detection Algorithm: Example 1 (Contd.)

5 processes, 3 resource types (Contd.)

4. P_1: Request $=$ (2,0,2) $\leq$ (5,2,4) ✓
   - Work $=$ (5,2,4) $+$ (2,0,0) $=$ **(7,2,4)**
   - Finish[P_1] $=$ true $\rightarrow$ **[T, T, T, T, F]**
5. P_4: Request $=$ (0,0,2) $\leq$ (7,2,4) ✓
   - Work $=$ (7,2,4) $+$ (0,0,2) $=$ **(7,2,6)**
   - Finish[P_4] $=$ true $\rightarrow$ **[T, T, T, T, T]**

Step 4: Check for deadlock

- All Finish[i] $=$ true for all $i \in \{0, 1, 2, 3, 4\}$
- Sequence: $\langle P_0, P_2, P_3, P_1, P_4 \rangle$

$\Rightarrow$**Result: NO DEADLOCK**
All processes can complete successfully

Detection Algorithm: Example 2 (With Deadlock)

Same system, P_2 requests additional instance of C

Modified Request Matrix

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	1
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2
Available				0	0	0

Apply Detection Algorithm

- Work = (0,0,0)
- P_0: Request = (0,0,0) ✓ $\Rightarrow$ Work = (0,1,0)
- P_2: Request = (0,0,1) $\leq$ (0,1,0)? **NO!**
- P_3: Request = (1,0,0) $\leq$ (0,1,0)? **NO!**
- P_1, P_4: Cannot satisfy requests

Final State

- Finish = [true, false, false, false, false]
- Can only reclaim resources from P_0
- Insufficient resources to fulfill other processes' requests

*Rightarrow***Result: DEADLOCK EXISTS!**
Deadlocked processes: P_1, P_2, P_3, P_4
These processes form one or more **disjoint cycles**

Observations

- Single additional request by P_2 caused deadlock
- Four out of five processes are deadlocked
- Recovery mechanism must now be invoked

Detection Algorithm Usage

When to Invoke Detection Algorithm? Decision depends on two factors:

1. **How Often Deadlocks Occur**
   - If deadlocks occur **frequently** ⇒ invoke detection **frequently**
   - If deadlocks occur **rarely** ⇒ invoke detection **less often**
2. **How Many Processes Affected**
   - More cycles ⇒ More processes affected
   - Difficult to identify which process "**caused**" the deadlock

Invocation Strategies

1. **Every resource request**: High overhead, but can identify cause (culprit) easily
2. **Fixed time intervals**: Reasonable overhead
3. **CPU utilization drops**: When system appears idle

⇒**Trade-off:** **More frequent detection = Higher overhead BUT easier to identify cause**

Recovery: Overview

Once deadlock is detected, system must break the deadlock. Two main approaches exist:

1. Process Termination

- Kill one or more processes
- Free their resources
- Simple to implement
- **Problem**: Lose work done

2. Resource Preemption

- Take resources from processes
- Give to other processes
- May require rollback
- **Problem**: Starvation risk

Process Termination

Method 1: Abort All Deadlocked Processes

- **Advantage**: Simple, definitely breaks deadlock
- **Disadvantage**: Very expensive - all partial computations lost

Method 2: Abort One Process at a Time

- Kill one process
- Run detection algorithm again
- Repeat until deadlock is eliminated
- **Advantage**: Less waste
- **Disadvantage**: High overhead (repeated detection)

Selection Criteria: Which Process to Terminate?

1. **Priority** of the process
2. **Computation time** and remaining time to completion
3. **Resources** used and needed
4. **Number of processes** that will need to be terminated

Resource Preemption

Successively preempt resources until deadlock is broken

Three Issues to Address

1. **Selecting a Victim**
   - Which resources and processes to preempt?
   - **Goal**: Minimize cost
   - **Cost factors**: Number of resources held, execution time consumed
2. **Rollback**
   - If preempt resource from process, what to do with process?
   - **Total rollback**: Abort and restart process
   - **Partial rollback**: Return to some **safe state**
   - Requires system to maintain *checkpoint* information
3. **Starvation**
   - Same process may always be picked as victim
   - Process never completes its task
   - **Solution**: Include **number of rollbacks** in cost factor

Recovery Situation Analysis

What is the situation?

- A deadlock has occurred involving multiple processes $P_1, P_2, \ldots, P_n$
- None of the processes can continue because each is **waiting for resources held by others**
- The operating system must **recover** from deadlock

The situation is evaluated based on these metrices:

- **Priority** → how important each process is
- **Time** → percentage of work already completed by each process
- **Resources** → number of resources currently held by each process

⇒**Tip: Choose to terminate or preempt from the process that minimizes overall cost based on these metrices**. In practice, this is often the process with **lowest priority**, **least progress made**, and **fewest resources held**.

Recovery: Example Scenario Analysis

Process Termination vs. Preemption

Deadlock: P_1, P_2, P_3, P_4 are deadlocked

Process Information

Process	Priority	Time	Resources
P_1	High	90%	3
P_2	Low	10%	2
P_3	Medium	50%	4
P_4	Medium	80%	2

Time = % completion

Decision Analysis

- **Terminate** P_1: High priority, high progress lost
- **Terminate** P_3: Medium priority, moderate progress lost
- **Terminate** P_4: Medium priority, high progress lost
- **Terminate** P_2: Low priority, minimal progress lost
- **Preempt from** P_1: High priority, rollback significant progress
- **Preempt from** P_3: Medium priority, rollback moderate progress
- **Preempt from** P_4: Medium priority, rollback high progress
- **Preempt from** P_2: Low priority, rollback minimal progress

⇒**Best Strategy: Terminate or Preempt from** P_2

Key Takeaways

Key Takeaways

- **Deadlock detection** identifies deadlocks using Resource Allocation Graphs (Wait-For Graphs) for single instances or Banker's-like algorithm for multiple instances
- **Detection frequency** depends on how often deadlocks occur and how many processes are affected
- **Recovery** from deadlock involves either process termination or resource preemption
- **Process termination** can be done by aborting all deadlocked processes or one at a time
- **Resource preemption** requires selecting a victim, rolling back, and avoiding starvation

Outline

1. Introduction to Deadlocks
2. Resource-Allocation Graph
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection and Recovery
7. Exercise

Exercise: System State Analysis

System State Analysis

Let's suppose the following snapshot of the system at a given moment. It contains:

- 5 processes P_0, P_1, P_2, P_3 and P_4
- 4 types of resources A, B, C and D with 12, n, 8 and m instances, respectively
- Matrices "*Allocation*" and "*Max*" that represent, for each process, the allocated instances from each resource type, and its maximum need, respectively

	Allocation				Max			
	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	y
P_1	3	1	2	1	7	2	5	2
P_2	2	1	0	3	6	3	1	6
P_3	1	3	1	2	5	4	2	4
P_4	1	x	3	2	3	6	5	5

Questions:

1. Fill the vector "*Available*"
2. Fill the matrix "*Need*"
3. What are the two conditions (expressed by n, m, x and y), according to them, P_0 can be executed firstly then the system *can* be in a safe state?
4. What are the two conditions (expressed by n, m and x), according to them, P_4 can be executed firstly then the system *can* be in a safe state?

Solution: (Questions a & b)

Calculating Available and CurrentNeed

a) Available vector:

Total resources - Allocated resources = Available

- Available[A] = 12 - (2+3+2+1+1) = 12 - 9 = **3**
- Available[B] = n - (0+1+1+3+x) = n - (5+x) = n**-5**-x
- Available[C] = 8 - (0+2+0+1+3) = 8 - 6 = **2**
- Available[D] = m - (1+1+3+2+2) = m - 9 = m**-9**

b) Need matrix: Need = Max - Allocation

	CurrentNeed			
	A	B	C	D
P_0	**2**	**2**	**1**	y**-1**
P_1	**4**	**1**	**3**	**1**
P_2	**4**	**2**	**1**	**3**
P_3	**4**	**1**	**1**	**2**
P_4	**2**	**6**-x	**2**	**3**

Solution: (Questions c & d)

Conditions for Safe State

c) For P_0 to execute first:
CurrentNeed[P_0] $\leq$ Available

- $(2, 2, 1, y\text{-}1) \leq (3, n\text{-}5\text{-}x, 2, m\text{-}9)$

This gives us:

1. $n\text{-}\mathbf{5}\text{-}x \geq \mathbf{2} \Rightarrow n \geq \mathbf{7}+x$
2. $m \geq y+\mathbf{8}$

d) For P_4 to execute first:
CurrentNeed[P_4] $\leq$ Available

- $(2, 6\text{-}x, 2, 3) \leq (3, n\text{-}5\text{-}x, 2, m\text{-}9)$

This gives us:

1. $n\text{-}\mathbf{5}\text{-}x \geq \mathbf{6}\text{-}x \Rightarrow n \geq \mathbf{11}$
2. $m \geq \mathbf{12}$

End of Chapter 6