# Deadlock

## Operating Systems

Yu-Ting Wu

*(with slides borrowed from Prof. Jerry Chou)*

# Outline

- System model
- Deadlock characterization
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection
- Deadlock recovery

# System Model

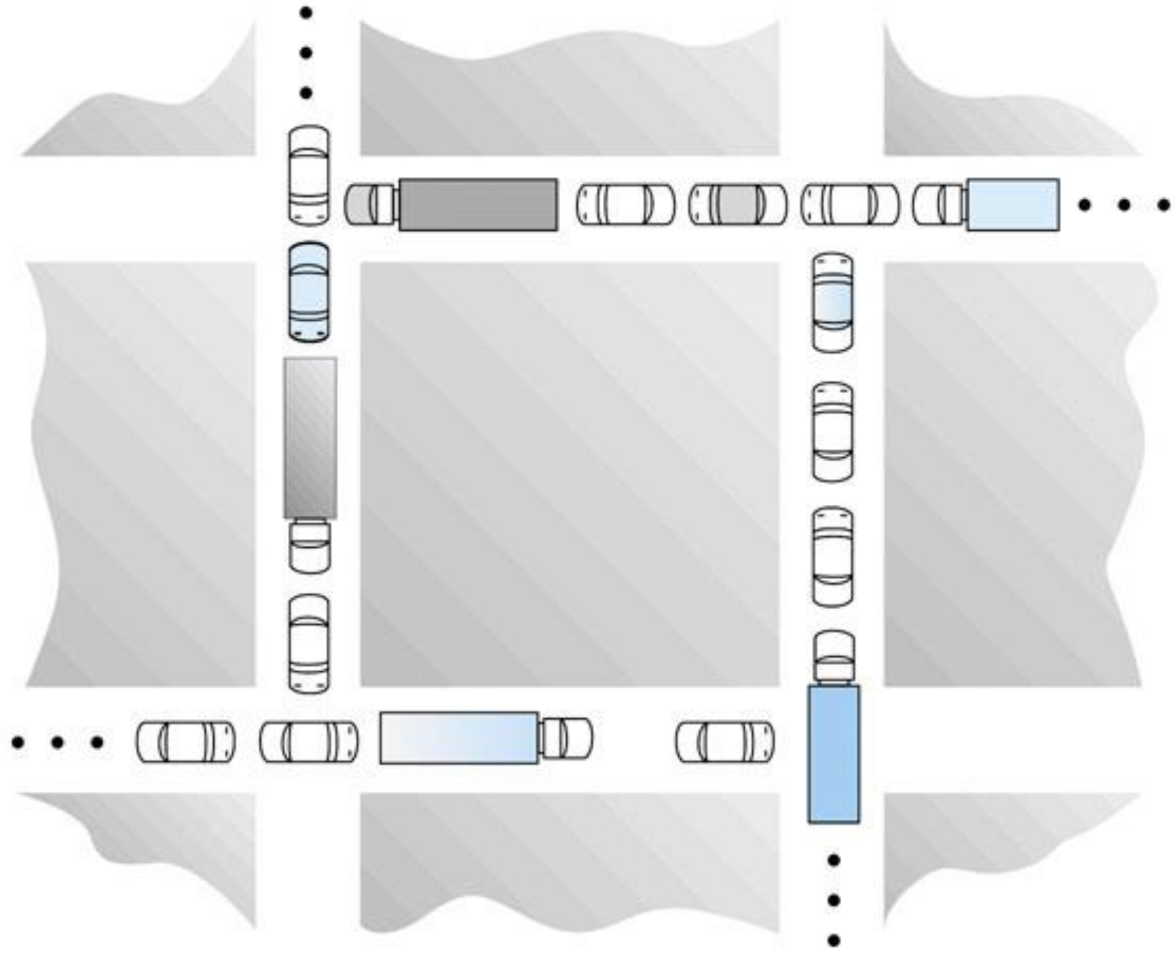
# Deadlock Problem

- A set of blocked processes each **holding** some resources and **waiting** to acquire a resource held by another process in the set
- Example:
  - 2 processes and semaphores A and B
    - $P_1$ (hold B, wait A): **wait (A)**, signal (B)
    - $P_2$ (hold A, wait B): **wait (B)**, signal (A)
- Example:
  - Dining philosophers' problem

# Necessary Conditions

- **Mutual exclusion**
  - Only 1 process at a time can use a resource
- **Hold and wait**
  - A process holding some resources and is waiting for another resource
- **No preemption**
  - A resource can be only released by a process **voluntarily**
- **Circular wait**
  - There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \rightarrow P_0$

# Necessary Conditions (cont.)



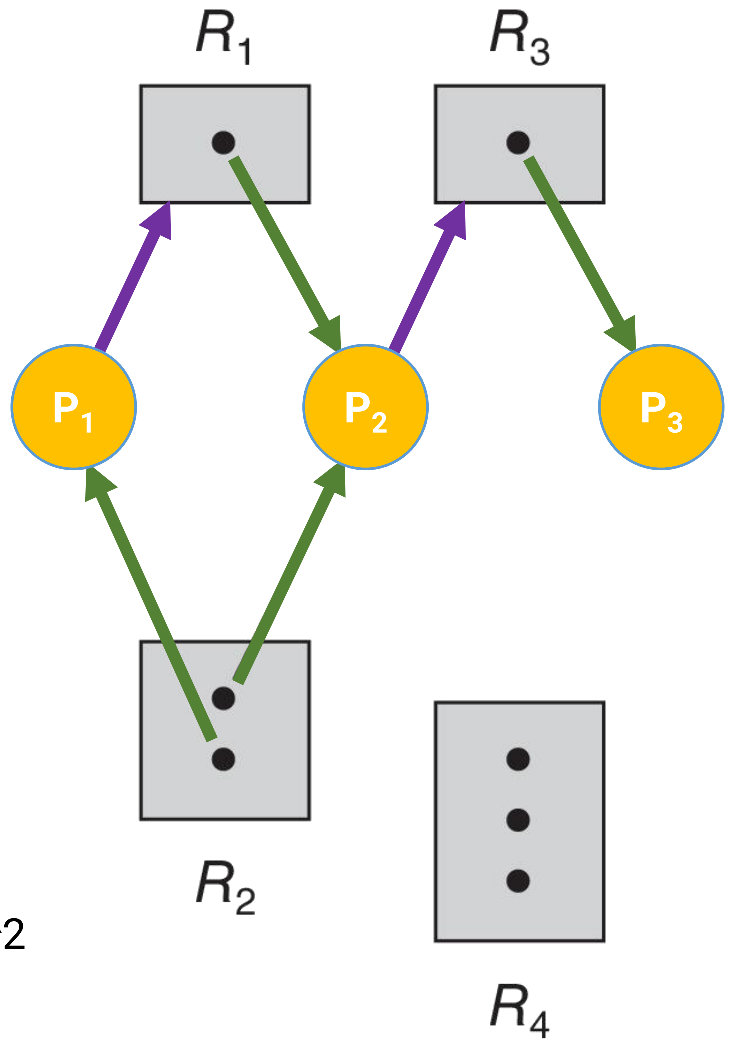
# System Model

- **Resources types** $R_1, R_2, ...., R_m$
  - E.g. CPU, memory pages, I/O devices
- Each resource type $R_i$ has $W_i$ **instances**
  - E.g. a computer has 2 CPUs
- Each process utilizes a resource as follows:
  - **Request ➔ use ➔ release**

# Resource-Allocation Graph

- 3 processes, $P_1$ ~ $P_3$
- 4 resources, $R_1$ ~ $R_4$
  - $R_1$ and $R_3$ each has one instance
  - $R_2$ has two instances
  - $R_4$ has three instances
- **Request edges**
  - $P_1$ ➔ $R_1$: $P_1$ requests $R_1$
- **Assignment edges**
  - $R_2$ ➔ $P_1$: one instance of $R_2$ is allocated to $P_1$

➔ $P_1$ is **holding on** an instance of $R_2$ and **waiting for** an instance or $R_1$

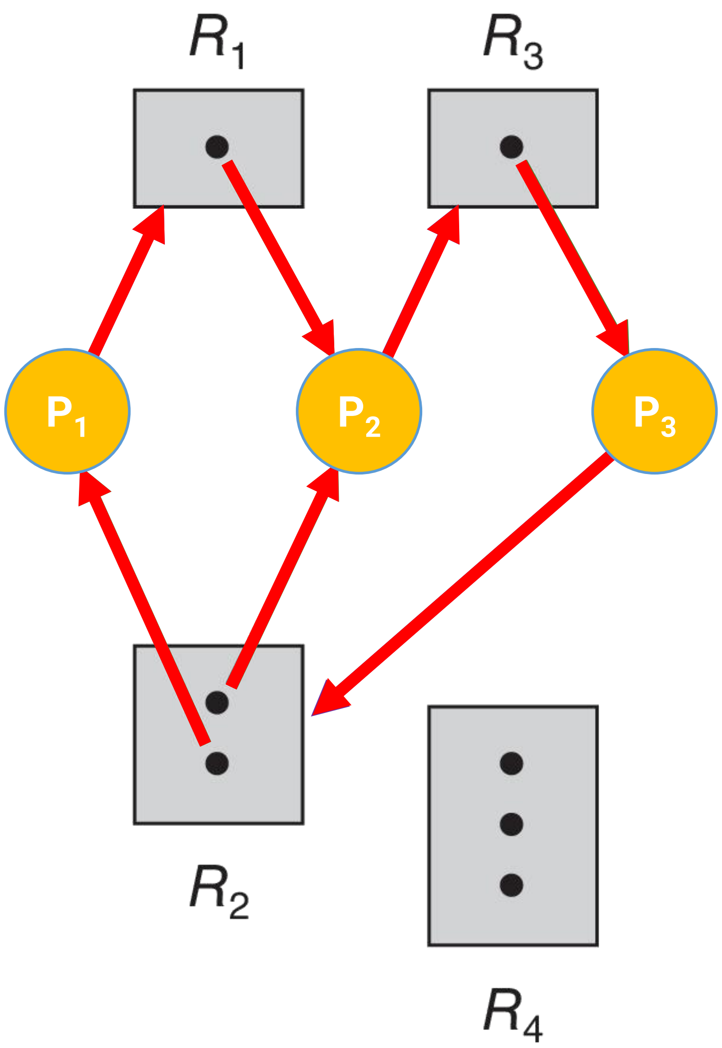


# Resource-Allocation Graph w/ Deadlock

- If the graph contains a **cycle**, a deadlock **may** exist
- In the example
  - $P_1$ is waiting for $P_2$
  - $P_2$ is waiting for $P_3$
  
  ➔ $P_1$ is also waiting for $P_3$

  - Since $P_3$ is waiting for $P_1$ or $P_2$, and they both waiting for $P_3$

  ➔ **Deadlock !**

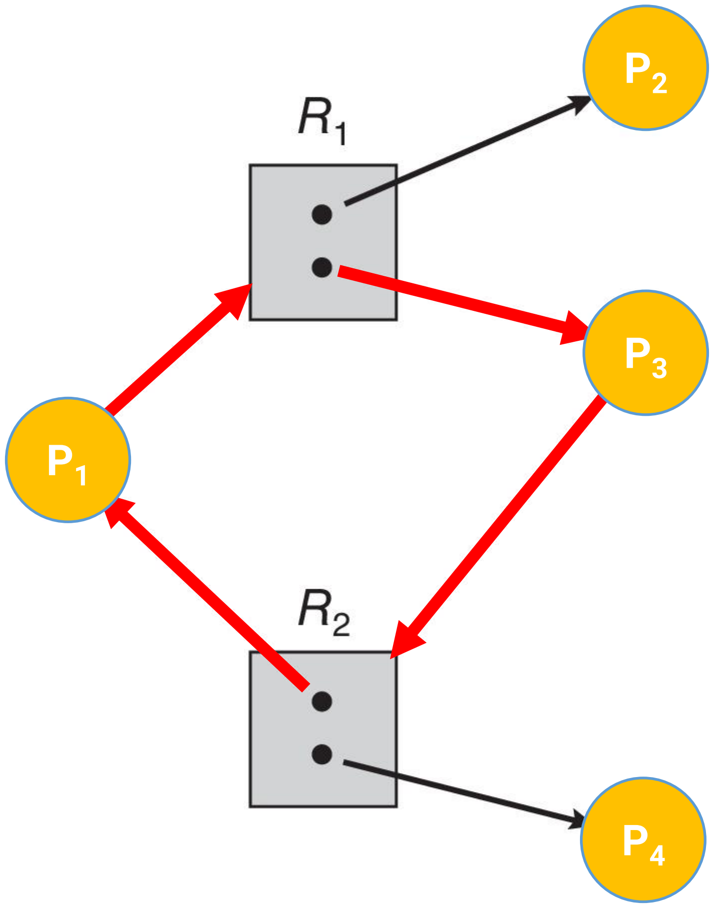


# RA Graph w/ Cycle but NO Deadlock

- If the graph contains a **cycle**, a deadlock **may** exist
- In the example
  - $P_1$ is waiting for $P_2$ or $P_3$
  - $P_3$ is waiting for $P_1$ or $P_4$

  - Since $P_2$ and $P_4$ wait for no one

**➔ No Deadlock between $P_1$ and $P_3$**



# Deadlock Detection

- If the graph contains **no cycle** ➔ **no deadlock**
  - **Circular wait cannot be held**
- If the graph contains a cycle
  - If **one instance** per resource type ➔ **deadlock**
  - If **multiple instances** per resource type ➔ **possibility** of deadlock

# Handling Deadlocks

- Ensure the system will **never** enter a deadlock state
  - **Deadlock prevention**: ensure that at least one of the **4 necessary conditions** cannot hold
  - **Deadlock avoidance**: **dynamically** examines the resource-allocation state before allocation
- Allow to **enter a deadlock state** and then **recover**
  - **Deadlock detection**
  - **Deadlock recovery**
- **Ignore the problem** and pretend that deadlocks never occur in the system
  - **Used by most operating systems, including UNIX**

# Deadlock Prevention

# Deadlock Prevention

- **Mutual exclusion (ME):** do not require ME on sharable resources
  - E.g. there is no need to ensure ME on read-only files
  - However, some resources are not shareable (e.g. printer)

- **Hold and wait:**
  - When a process requests a resource, it does not hold any resource
  - Pre-allocate all resources before executing
  - **Resource utilization is low; starvation is possible**

# Deadlock Prevention (cont.)

- **No preemption:**
  - When a process is waiting on a resource, all its holding resources are preempted
    - E.g. $P_1$ request $R_1$, which is allocated to $P_2$, which in turn is waiting on $R_2$ ($P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2$)
    - **R1 can be preempted and reallocated to P1**
  - **Applied to resources whose states can be easily saved and restored later**
    - E.g. CPU registers and memory
  - It cannot easily be applied to other resources
    - E.g. printers and tape drives

# Deadlock Prevention (cont.)

- **Circular wait:**
  - Impose a **total ordering** of all resource types
  - A process requests resources in an increasing order
    - Let R = {$R_0$, $R_1$, … $R_n$} be the set of resource types
    - **When request $R_k$, should release all $R_i$, $i \geq k$**
- Example
  - F (disk drive) = 5, F(printer) = 12
  - A process must request disk drive before printer
- Proof: counter-example does not exist
  - $P_0(R_0) \rightarrow R_1, P_1(R_1) \rightarrow R_2, \ldots, P_n(R_n) \rightarrow R_0$ ← $P_n$ **holds on** $R_n$, **waiting for** $R_0$
  - Conflict: $R_0 < R_1 < R_2 < \ldots R_n < R_0$

# Deadlock Avoidance

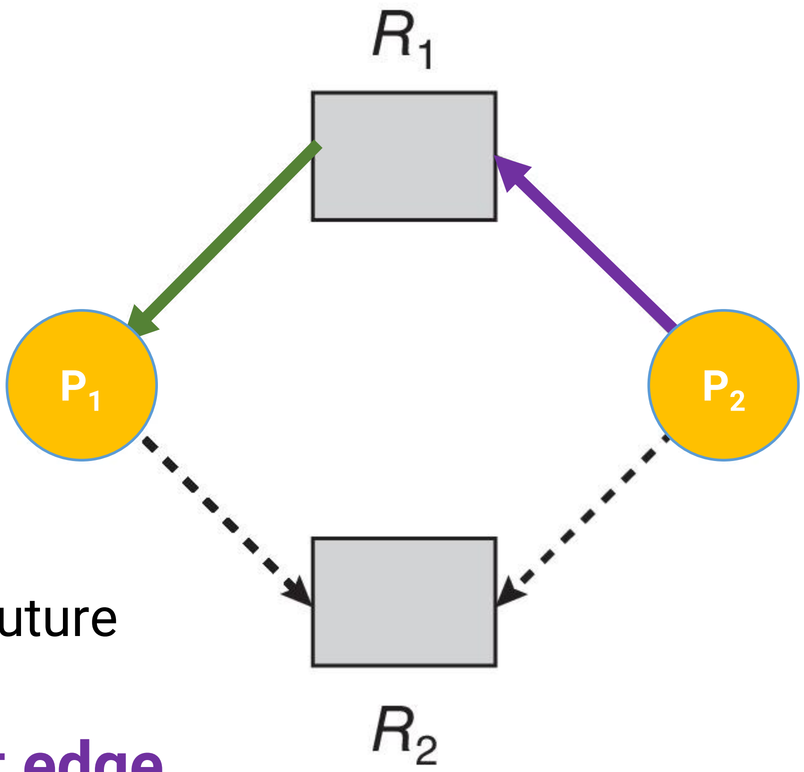
# Avoidance Algorithms

- **Single instance of a resource type**
  - **Resource-allocation graph (RAG) algorithm** based on **circle detection**

- **Multiple instance of a resource type**
  - **banker's algorithm** based on safe **sequence detection**

# Resource-Allocation Graph Algorithm

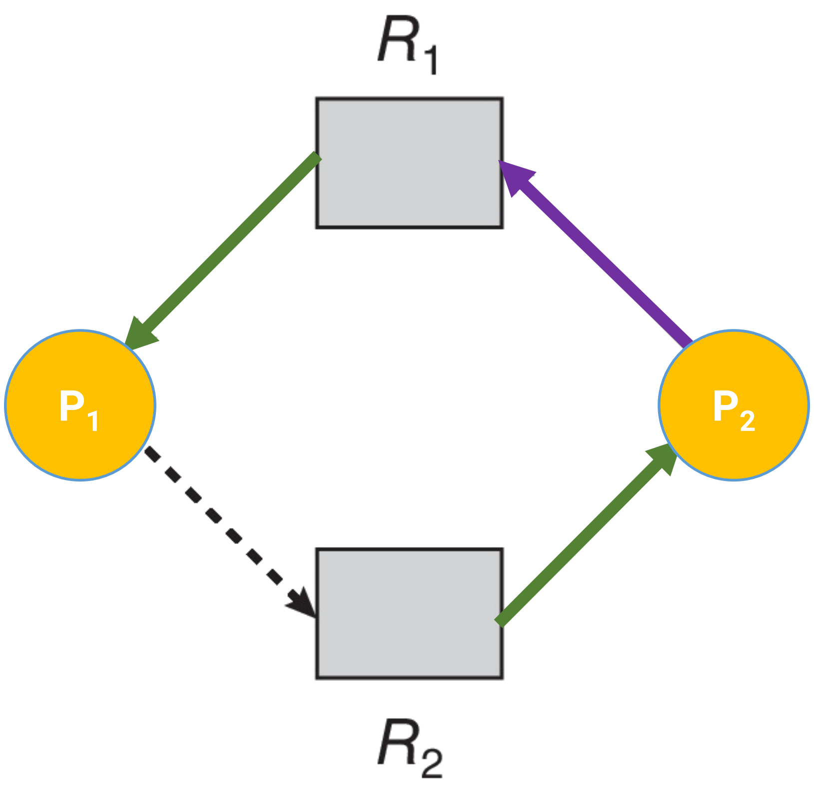
- **Request edges**
  - $P_i$ ➔ $R_j$: $P_i$ **is waiting** for resource $R_j$
- **Assignment edges**
  - $R_j$ ➔ $P_i$: Resource $R_j$ **is allocated** and held by $P_i$




- **Claim edge**
  - Process $P_i$ **may** request $R_j$ in the future
- **Claim edge** converts to **request edge**
  - When a resource **is requested** by process
- **Assignment edge** converts back to a **claim edge**
  - When a resource is released by a process

# Resource-Allocation Graph Algorithm (cont.)

- Resource **must be claimed a priori** in the system
- **Grant a request** only if NO cycle created
- Check for safety using a **cycle-detection algorithm**, $O(n^2)$
- Example: $R_2$ cannot be allocated to $P_2$



# Avoidance Algorithms

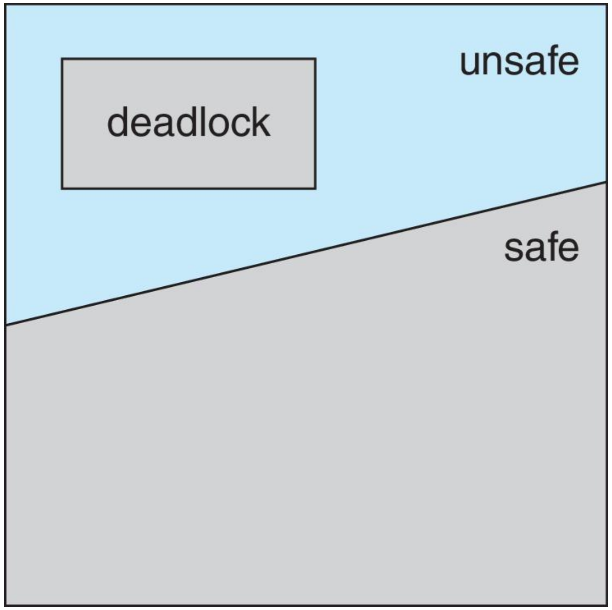
- **Single instance of a resource type**
  - **Resource-allocation graph (RAG) algorithm** based on **circle detection**

- **Multiple instance of a resource type**
  - **banker's algorithm** based on safe **sequence detection**

# Deadlock Avoidance

- **Safe state:** a system is in a safe state if there exists **a sequence of allocations** to satisfy requests by all processes
  - This sequence of allocations is called **safe sequence**
- **Safe state ➔ no deadlock**
- **Unsafe state ➔ possibility of deadlock**
- **Deadlock avoidance ➔**

  **ensure that a system never enters an unsafe state**



# Safe State with Safe Sequence

- There are 12 tape drives
- Assuming at $t_0$:

hints from processes

| | Max Needs | Current Holding |
|---|---|---|
| P0 | 10 | 5 |
| P1 | 4 | 2 |
| P2 | 9 | 2 |

➔ $\langle P_1, P_0, P_2 \rangle$ is a safe sequence

# Safe State with Safe Sequence

- There are 12 tape drives
- Assuming at $t_0$:

| | Max Needs | Current Holding | Available |
|---|---|---|---|
| P0 | 10 | 5 | |
| P1 | 4 | 2 | 3 |
| P2 | 9 | 2 | |

➔ $<P_1, P_0, P_2>$ is a safe sequence

1. $P_1$ satisfies its allocation with 3 available resources

# Safe State with Safe Sequence

- There are 12 tape drives
- Assuming at $t_0$:

| | Max Needs | Current Holding | Available |
|---|---|---|---|
| P0 | 10 | 5 | 5 |
| P1 | 4 | 0 | |
| P2 | 9 | 2 | |

➔<$P_1$, $P_0$, $P_2$> is a safe sequence

1. $P_1$ satisfies its allocation with 3 available resources
2. $P_0$ satisfies its allocation with 5 available resources

# Safe State with Safe Sequence

- There are 12 tape drives
- Assuming at $t_0$:

| | Max Needs | Current Holding | Available |
|---|---|---|---|
| P0 | 10 | 5 | |
| P1 | 4 | 0 | |
| P2 | 9 | 2 | 10 |

➔<$P_1$, $P_0$, $P_2$> is a safe sequence

1. $P_1$ satisfies its allocation with 3 available resources
2. $P_0$ satisfies its allocation with 5 available resources
3. $P_2$ satisfies its allocation with 10 available resources

# Safe State with Safe Sequence

- There are 12 tape drives
- Assuming at $t_1$:

| | Max Needs | Current Holding | Available |
|---|---|---|---|
| P0 | 10 | 5 | |
| P1 | 4 | 2 | 2 |
| P2 | 9 | 3 | |

- If $P_2$ requests and is allocated 1 more resource
  - ➔ No safe sequence exist …
  - ➔ This allocation makes the system enter an unsafe state
- **A request is only granted if the allocation leaves the system in a safe state**

# Banker's Algorithm

- Use for **multiple instances** of each resource type

- **Banker's Algorithm**
  - Use a general safety algorithm to **pre-determine** if any **safe sequence** exists after allocation
  - **Only proceed the allocation if safe sequence exists**
- **Safety algorithm**
  1. Assume processes need **maximum** resources
  2. Find a process that can be satisfied by free resources
  3. Free the resource usage of the process
  4. Repeat to step 2 until all processes are satisfied

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: 3, B: 3, C: 2

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- Safe sequence: $P_1$

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: **5**, B: 3, C: 2

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- Safe sequence: $P_1$, $P_3$

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: **7**, B: **4**, C: **3**

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- Safe sequence: $P_1$, $P_3$, $P_4$

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: 7, B: 4, C: **5**

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- Safe sequence: $P_1$, $P_3$, $P_4$, $P_2$

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: **10**, B: 4, C: **7**

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- Safe sequence: $P_1$, $P_3$, $P_4$, $P_2$, $P_0$

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: 3, B: 3, C: 2

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- If Request ($P_1$) = (1, 0, 2) …

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: **2**, B: 3, C: **0**

| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 |

- If Request ($P_1$) = (1, 0, 2): P1 allocation ➔ (3, 0, 2)
  - Enter another safe state (Safe sequence: $P_1$, $P_3$, $P_4$, $P_0$, $P_2$)

# Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: **0**, B: **0**, C: 2

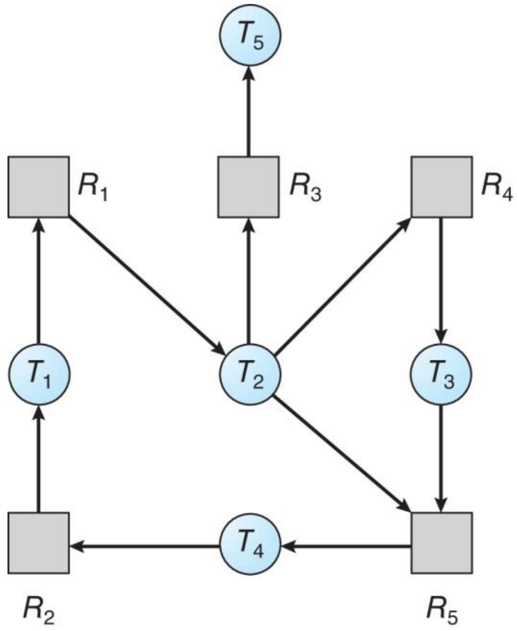
| | Max | | | Allocation | | | Need (Max – Alloc.) | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 7 | 5 | 3 | 0 | 1 | 0 | 7 | 4 | 3 |
| $P_1$ | 3 | 2 | 2 | 2 | 0 | 0 | 1 | 2 | 2 |
| $P_2$ | 9 | 0 | 2 | 3 | 0 | 2 | 6 | 0 | 0 |
| $P_3$ | 2 | 2 | 2 | 2 | 1 | 1 | 0 | 1 | 1 |
| $P_4$ | 4 | 3 | 3 | **3** | **3** | **2** | **1** | **0** | **1** |

- If Request ($P_4$) = (3, 3, 0): $P_4$ allocation ➔ (3, 3, 2)
  - Enter into an unsafe state (no safe sequence can be found)

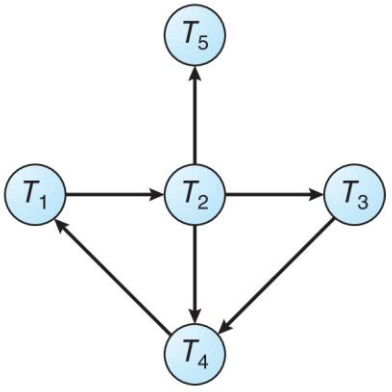
# Deadlock Detection

# Deadlock Detection

- **Single instance** of each resource type
  - Convert request/assignment edges into **wait-for graph**
  - Deadlock exists if there is a cycle in the wait-for graph




**Resource-Allocation graph**




**corresponding wait-for graph**

# Multiple Instance for Each Resource Type

- Total instances: A: 7, B: 2, C: 6
- Available instances: A: 0, B: 0, C: 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 |

- The system is in a safe state ➔ $<P_0, P_2, P_3, P_1, P_4>$

  ➔ No deadlock

# Multiple Instance for Each Resource Type

- Total instances: A: 7, B: 2, C: 6
- Available instances: A: 0, B: 0, C: 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 | **1** |
| $P_3$ | 2 | 1 | 1 | 1 | 0 | 0 |
| $P_4$ | 0 | 0 | 2 | 0 | 0 | 2 |

- If $P_2$ requests (0, 0, 1) ➔ no safe sequence can be found

  ➔ The system is deadlocked

# Deadlock Recovery

# Deadlock Recovery

- **Process termination**
  - Abort all deadlocked processes
  - Abort 1 process at a time until the deadlock cycle is eliminated
    - Which process should we abort first?

- **Resource preemption**
  - Select a victim: which one to preempt?
  - Rollback: partial rollback or total rollback?
  - Starvation: can the same process be preempted always?

# Objective Review

- Illustrate how deadlock can occur
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm