




# Operating System Concepts

Che-Wei Chang

[chewei@mail.cgu.edu.tw](mailto:chewei@mail.cgu.edu.tw)

Department of Computer Science and Information Engineering, Chang Gung University

# Contents

1. Introduction
2. System Structures
3. Process Concept
4. Multithreaded Programming
5. Process Scheduling
6. Synchronization
-  7. Deadlocks
8. Memory-Management Strategies
9. Virtual-Memory Management
10. File System
11. Implementing File Systems
12. Secondary-Storage Systems





# Chapter 7. Deadlocks

# 專題展參與

- ▶ 2023/01/04(三)當天一樣會請助教點名，時間地點前一週公布。
- ▶ 有參與點名，總成績+1。
- or
- ▶ 有參與點名且有交心得，心得包含五組以上每組一百字以上，總成績+3
- or
- ▶ 沒點名但有交心得，心得包含五組以上每組一百字以上，總成績+1
- ▶ 心得於2023/01/11中午12:00前上傳e-learning

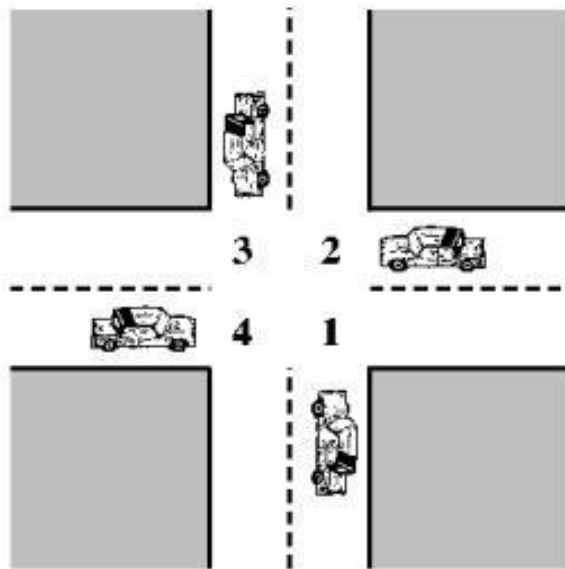


# Objectives

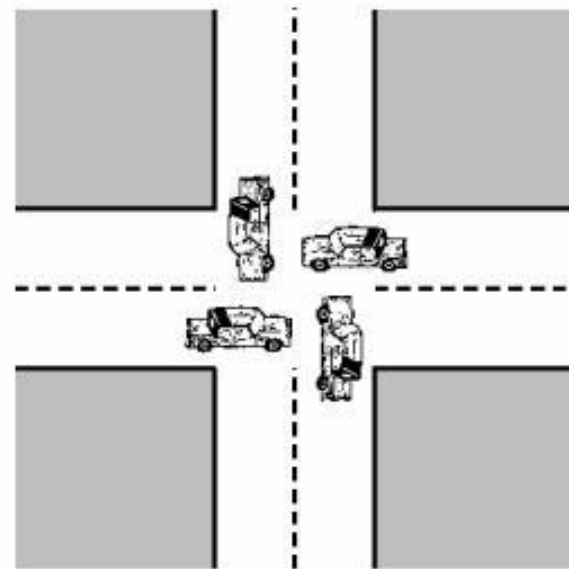
- ▶ To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- ▶ To present a number of different methods for preventing or avoiding deadlocks in a computer system



# Illustration of Deadlock



(a) Deadlock possible



(b) Deadlock

# Deadlocks

- ▶ A set of process is in a **deadlock** state when every process in the set is waiting for an event that can be caused by only another process in the set
- ▶ System Model
  - System consists of resources
  - Resource types  $R_1, R_2, \dots, R_m$ 
    - e.g. CPU, memory space, I/O devices, ...
    - Each resource type  $R_i$  has  $W_i$  instances
  - Each process utilizes a resource as follows:
    - **request**
    - **use**
    - **release**



# Deadlock Characterization

- ▶ **Mutual exclusion:** only one process at a time can use a resource
  - ▶ **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - ▶ **No preemption:** a resource can be released only voluntarily by the process holding it
  - ▶ **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_{n-1}\}$  of waiting processes such that each  $P_i$  is waiting for a resource that is held by  $P_{(i+1)\%n}$
- ➔ **Deadlock can arise if four conditions hold simultaneously**





# Resource–Allocation Graph

- ▶ A set of vertices  $V$  and a set of edges  $E$
- ▶  $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- ▶  $E$  has two types:
  - Request edge : directed edge  $P_i \rightarrow R_j$
  - Assignment edge : directed edge  $R_j \rightarrow P_i$



# Resource-Allocation Graph

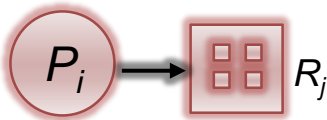
- ▶ Process



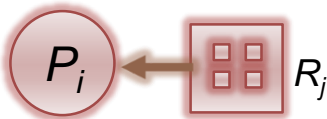
- ▶ Resource Type with 4 instances



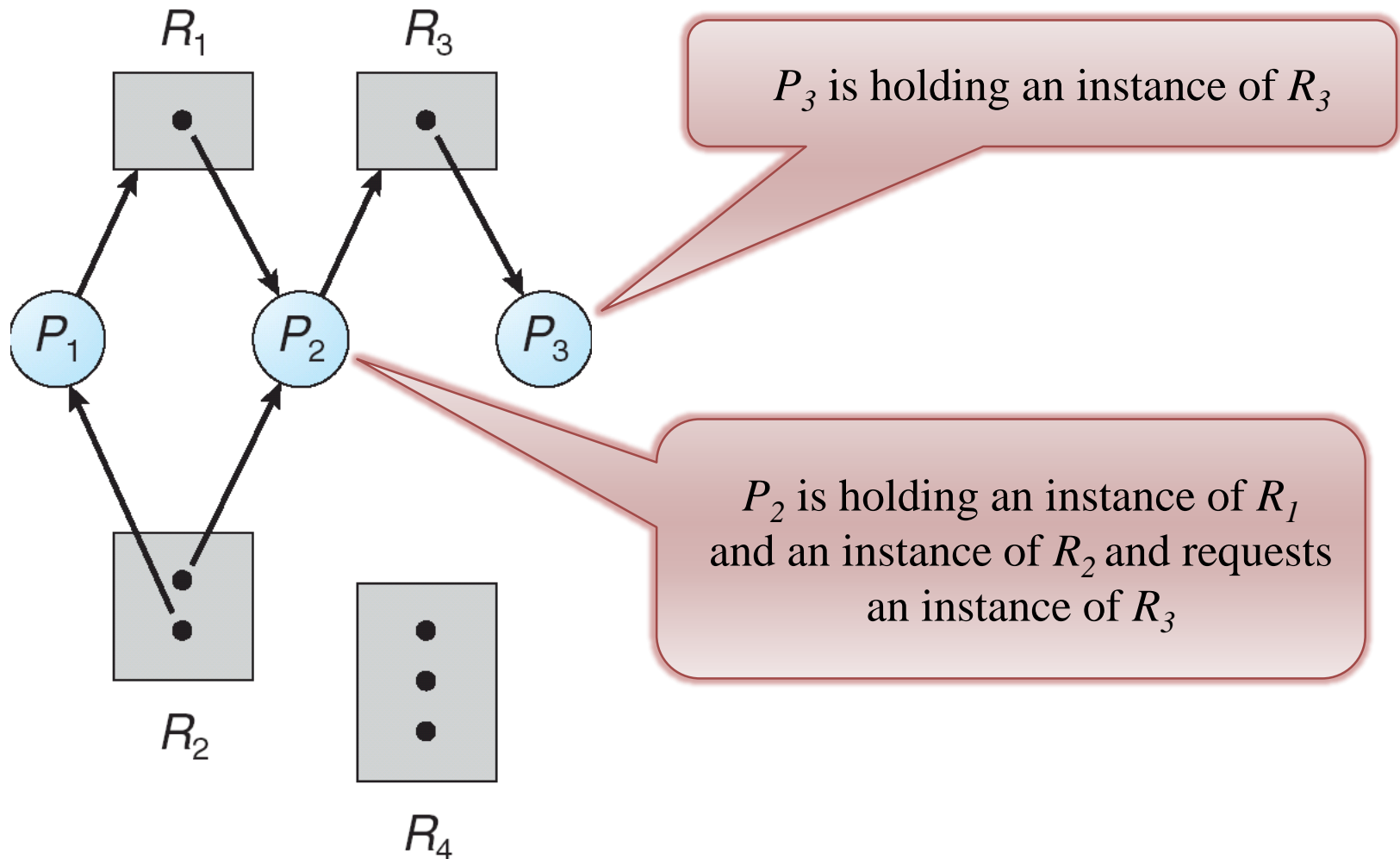
- ▶  $P_i$  requests an instance of  $R_j$



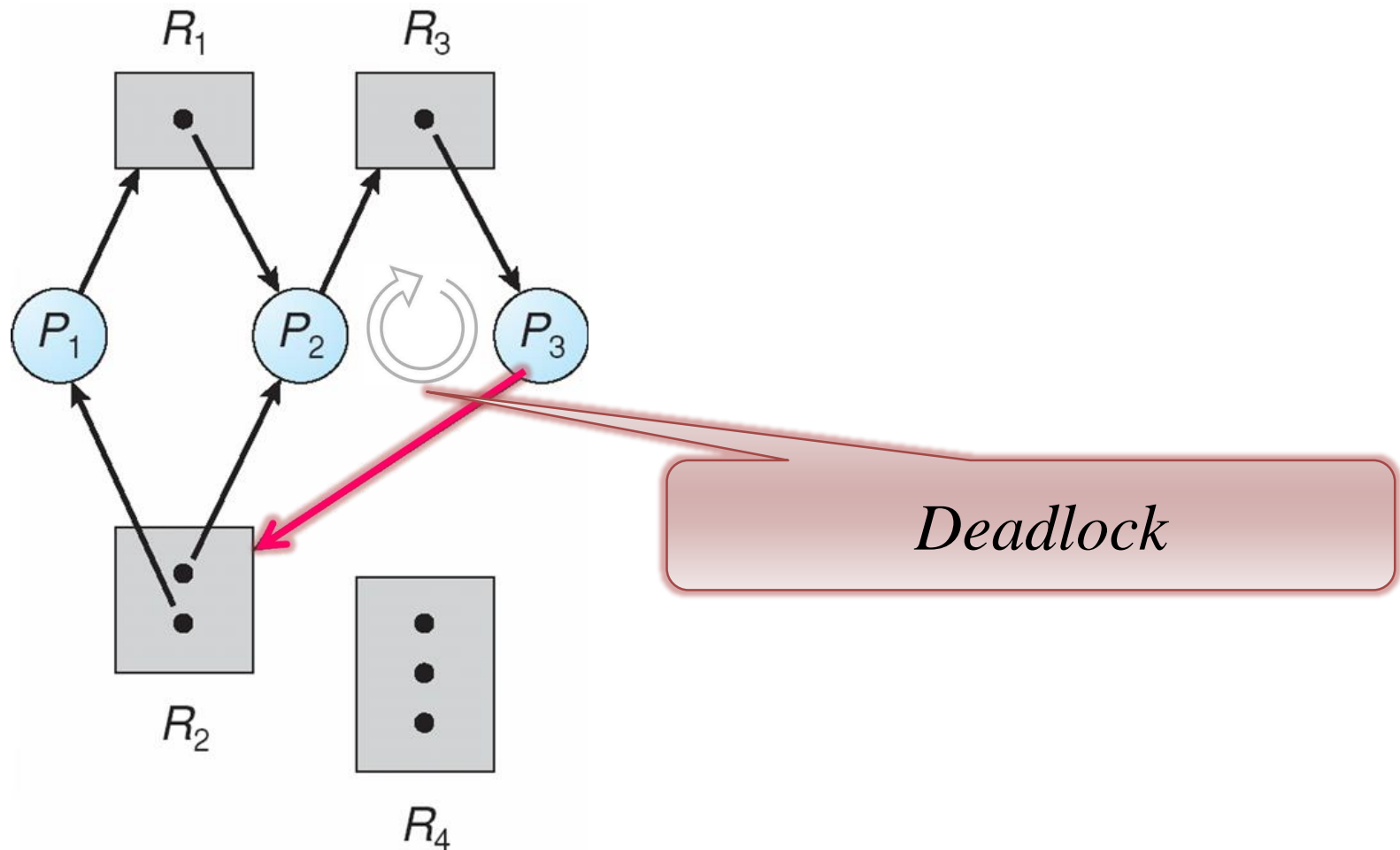
- ▶  $P_i$  is holding an instance of  $R_j$



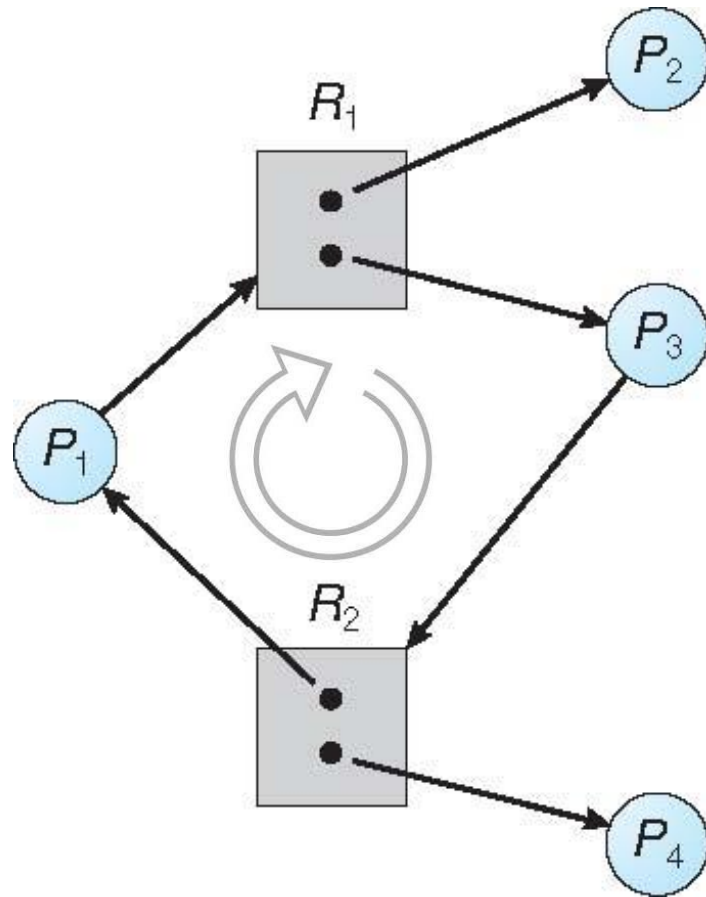
# A Resource Allocation Graph



# A Deadlock in a Resource Allocation Graph



# No Deadlock in a Cycle



The cycle will be broken  
after  $P_2$  is finished

# An Example of Deadlock

Code:

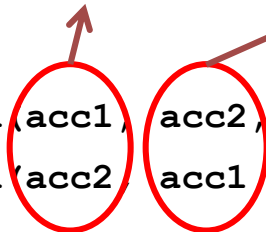
```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get lock(from);
    lock2 = get lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Use:

```
transaction(acc1, acc2, 1000);
transaction(acc2, acc1, 4000);
```

Hold

Wait



# Methods for Handling Deadlocks

- ▶ Make sure that the system never has a deadlock
  - **Deadlock Prevention**: Prevent the necessary conditions
  - **Deadlock Avoidance**: Make sure that the system always stays at a “safe” state
- ▶ Do recovery if the system is deadlocked
  - Deadlock Detection
  - Recovery
- ▶ Ignore the possibility of deadlock occurrences
  - Restart the system manually if the system seems to be deadlocked or stops functioning
  - Note that the system may be frozen temporarily



# Deadlock Prevention

## ► Goal:

- Try to fail anyone of the necessary conditions
- The Necessary Conditions
  - Mutual Exclusion
    - Some resources, such as a printer, are intrinsically non-sharable
  - Hold and Wait
  - No Preemption
  - Circular Wait





# Deadlock Prevention— Hold and Wait

## ► Rules

- Acquire all needed resources before its execution
- or
- Release allocated resources before request additional resources

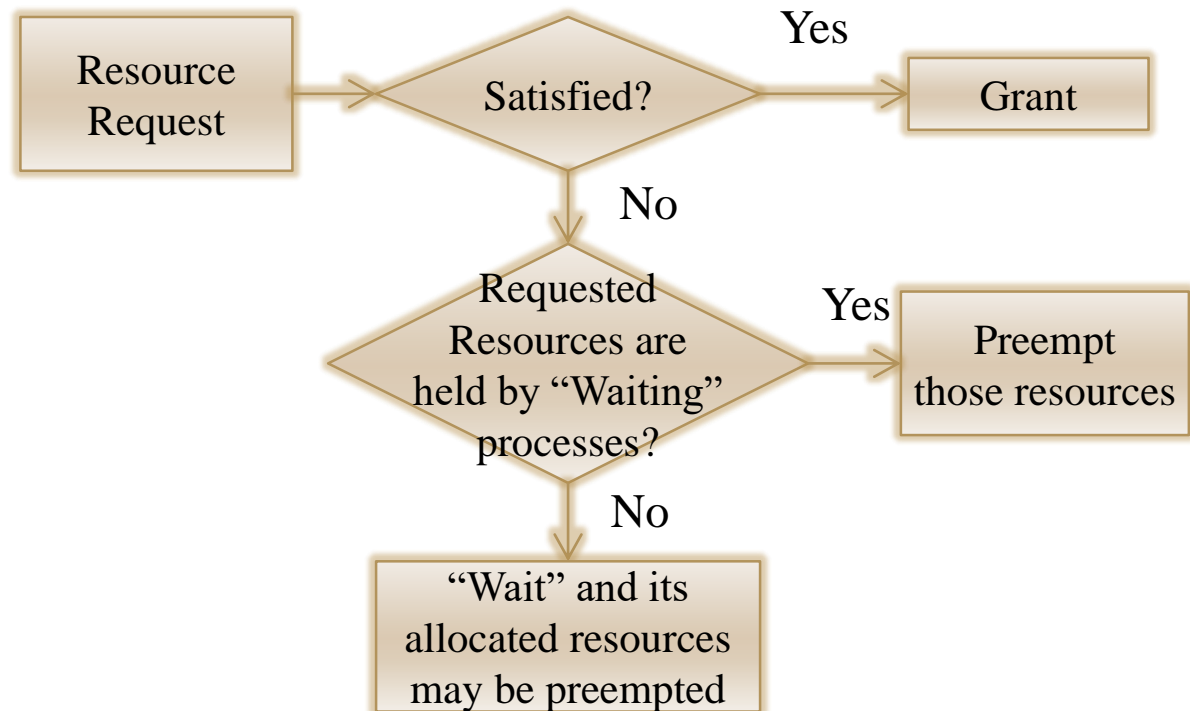
## ► Disadvantage

- Low resource utilization
- Starvation



# Deadlock Prevention— No Preemption

- ▶ Related protocols are only applied to resources whose states can be saved and restored, e.g., CPU registers & memory space, instead of printers or tape drives
- ▶ Example



# Deadlock Prevention—Circular Wait

## ▶ Rule

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

## ▶ Example

```
/* thread one runs in this function */  
void *do work one(void *param)  
{  
    lock(&first mutex);  
    lock(&second mutex);  
    /** * Do some work */  
    unlock(&second mutex);  
    unlock(&first mutex);  
    exit(0);  
}
```

The order is not allowed

```
/* thread two runs in this function */  
void *do work two(void *param)  
{  
    lock(&second mutex);  
    lock(&first mutex);  
    /** * Do some work */  
    unlock(&first mutex);  
    unlock(&second mutex);  
    exit(0);  
}
```



# Deadlock Avoidance

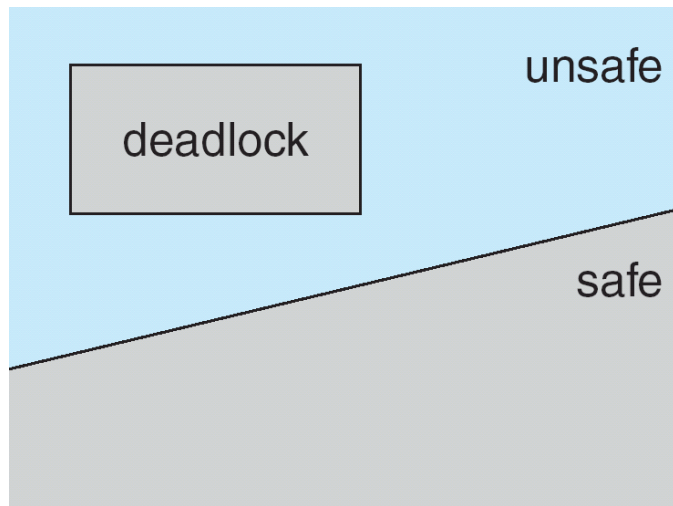
## ► Goal:

- Dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- i.e., keep the system at a **safe state**
- Require that the system has some additional information
  - For each resource
    - Count the allocated amount
    - Log the available amount
  - For each process
    - Know the maximum demand of each resource
    - Count the allocated amount of each resource



# Safe State (1 / 2)

- ▶ If a system is in safe state → no deadlocks
- ▶ If a system is in unsafe state → possibility of deadlock
- ▶ Avoidance → ensure that a system will never enter an unsafe state



# Safe State (2 / 2)

- ▶ System is in a safe state if there exists a **safe sequence** of all processes
- ▶ A sequence  $\langle P_1, P_2, \dots, P_n \rangle$  is safe if for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources plus the resources held by all the  $P_j$ , with  $j < i$
- ▶ That is:
  - When  $P_j$  is finished,  $P_i$  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



# Deadlock Avoidance

- ▶ Example: for only one type of resources

	Max needs	Allocated	Available
$P_0$	10	5	3
$P_1$	4	2	
$P_2$	9	2	

- ▶ The existence of a safe sequence  $\langle P_1, P_0, P_2 \rangle$
  - ▶ If  $P_2$  got two more, the system state is unsafe
- ➔ How to ensure that the system will always remain in a safe state?



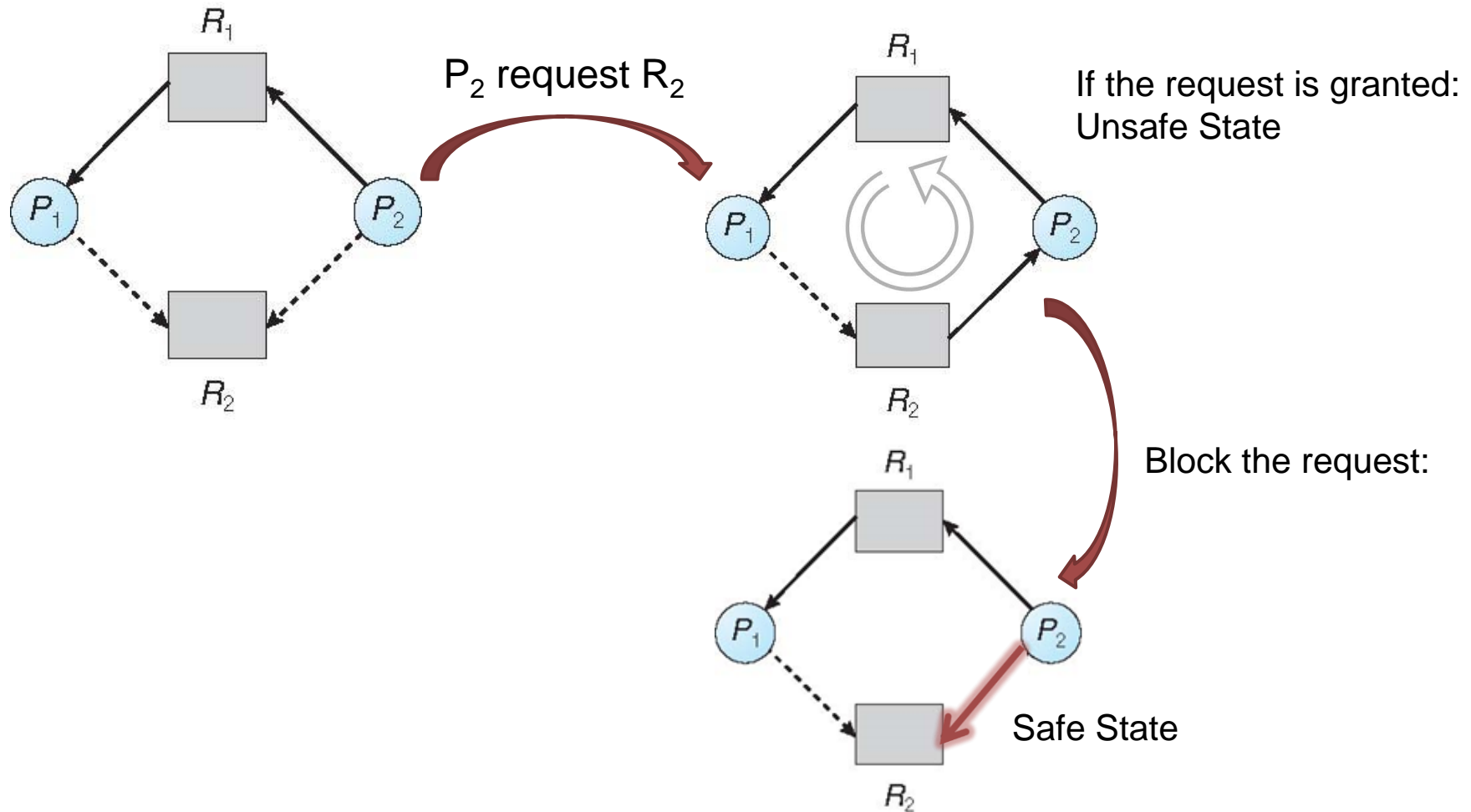
# Resource-Allocation Graph Scheme (1 / 2)

- ▶ **Claim edge**  $P_i \rightarrow R_j$  indicated that process  $P_i$  may request resource  $R_j$ ; represented by a dashed line
- ▶ Claim edge converts to request edge when a process requests a resource
- ▶ Request edge converted to an assignment edge when the resource is allocated to the process
- ▶ When a resource is released by a process, assignment edge reconverts to a claim edge





# Resource-Allocation Graph Scheme (2/2)



# Banker's Algorithm (1 / 3)

- ▶ Available [m]
    - If Available [i] = k, there are k instances of resource type  $R_i$  available
  - ▶ Max [n,m]
    - If Max [i,j] = k, process  $P_i$  may request at most k instances of resource type  $R_j$
  - ▶ Allocation [n,m]
    - If Allocation [i,j] = k, process  $P_i$  is currently allocated k instances of resource type  $R_j$
  - ▶ Need [n,m]
    - If Need [i,j] = k, process  $P_i$  may need k more instances of resource type  $R_j$
- Need [i,j] = Max [i,j] – Allocation [i,j]

$n$ : number of processes  
 $m$ : number of resource types



# Banker's Algorithm (2 / 3)

## —Safe State Checking

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively

Initialize:

**Work**[ $i$ ]  $\leftarrow$  **Available**[ $i$ ] for  $i = 0, 1, \dots, m-1$ , which means the current available instances of each resource

**Finish**[ $i$ ]  $\leftarrow$  **false** for  $i = 0, 1, \dots, n-1$ , which means if process  $P_i$  is finished

2. Find a process  $P_i$  such that both:

(a) **Finish**[ $i$ ] == **false**

(b) **Need**[ $i$ ]  $\leq$  **Work**

If no such  $i$  exists, go to step 4

●  $X \leq Y$  if  $X[k] \leq Y[k]$  for all  $k$

3. **Work**  $\leftarrow$  **Work** + **Allocation**[ $i$ ]

**Finish**[ $i$ ]  $\leftarrow$  **true**

go to Step 2

●  $X \leftarrow X + Y$  means  $X[k] \leftarrow X[k] + Y[k]$  for all  $k$

4. If **Finish** [ $i$ ] == **true** for all  $i$ , then the system is in a safe state; otherwise, the system is unsafe



# Banker's Algorithm (3 / 3)

## —Resource-Request Algorithm

**Request[i]** is the request vector for process  $P_i$ . If **Request[i,j] = k** then process  $P_i$  wants **k** instances of resource type  $R_j$

1. If  $\text{Request}[i] \leq \text{Need}[i]$ , then goto Step 2; otherwise, Trap
2. If  $\text{Request}[i] \leq \text{Available}$ , then goto Step3; otherwise,  $P_i$  must wait
3. Have the system pretend to have allocated resources to process  $P_i$  by setting:  
     $\text{Available} \leftarrow \text{Available} - \text{Request}[i];$   
     $\text{Allocation}[i] \leftarrow \text{Allocation}[i] + \text{Request}[i];$   
     $\text{Need}[i] \leftarrow \text{Need}[i] - \text{Request}[i];$
4. Execute “**Safe State Checking**”. If the system state is safe, the request is granted; otherwise,  $P_i$  must wait, and the old resource allocation state is restored



# Deadlock Avoidance Example (1 / 2)

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Is it in a safe state now?

Yes, a safe sequence is  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$



# Deadlock Avoidance Demo

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
4										3	3	2
1										5	3	2
P2	3	0	2	9	0	2	6	0	0	7	4	3
2										7	4	5
3										7	5	5



# Deadlock Avoidance Example (2 / 2)

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Let  $P_1$  make a request  $\text{Request}[1] = (1,0,2)$   $\text{Request}[1] \leq \text{Available}$  (i.e.,  $(1,0,2) \leq (3,3,2)$ )

Should we grant it? Yes, there is still a safe sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

If  $\text{Request}[4] = (3,3,0)$  is asked later, it must be rejected

If  $\text{Request}[0] = (0,2,0)$  is asked later, it must be rejected because it results in an unsafe state



# Deadlock Detection

- ▶ Approach:
  - Allow system to enter deadlock state
- ▶ Thus, we need:
  - Detection algorithm
  - Recovery scheme



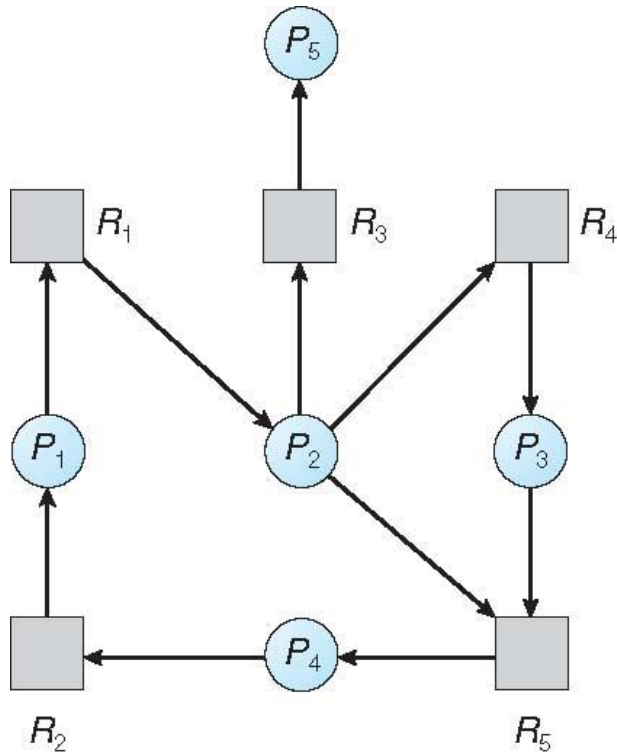


# Single Instance of Each Resource Type (1 / 2)

- ▶ Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- ▶ Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, there exists a deadlock

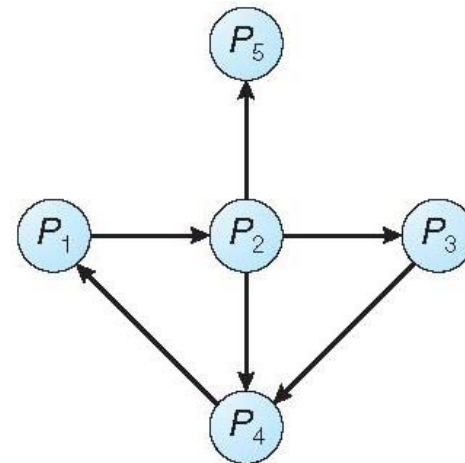


# Single Instance of Each Resource Type (2/2)



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph



# Multiple Instances of Each Resource Type (1 / 2)

$n$ : number of processes,  $m$ : number of resource types

## ► Data Structures

- Available[1..m]: number of available resource instances
- Allocation[1..n, 1..m]: current resource allocation to each process
- Request[1..n, 1..m]: the current request of each process
- If Request[i,j] = k,  $P_i$  is now requesting k more instances of resource type  $R_j$



# Multiple Instances of Each Resource Type (2 / 2)

1.  $Work[1..m] \leftarrow Available[1..m]$   
 $Finish[1..n] \leftarrow False$
2. Find a process  $P_i$  such that both
  - a.  $Finish[i] = False$
  - b.  $Request[i] \leq Work$**If** no such  $i$ , **goto** Step 4
3.  $Work \leftarrow Work + Allocation[i]$   
 $Finish[i] := True$   
**goto** Step 2
4. **If**  $Finish[i] = False$  for some  $P_i$ , **then** the system is in a deadlock state  
**If**  $Finish[i] = False$ , then process  $P_i$  is deadlocked



# Deadlock Detection Example

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	2	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			

- ▶ Find a sequence  $\langle P0, P2, P3, P1, P4 \rangle$  such that  $Finish[i] = \text{True}$  for all  $i$
- ▶ If  $Request[2] = (0,0,1)$  is issued, then P1, P2, P3, and P4 are deadlocked



# Deadlock Detection Demo

		Allocation			Request		
		A	B	C	A	B	C
1	P0						
4	P1						
2	P2						
3	P3						
	P4	0	0	2	0	0	2

Available		
A	B	C
0	2	0
0	3	0
3	3	3
5	4	4
7	4	4



# Detection-Algorithm Usage

- ▶ When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- ▶ If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the deadlocked processes “caused” the deadlock



# Recovery from Deadlock: Process Termination

- ▶ Abort all deadlocked processes
- ▶ Abort one process at a time until the deadlock cycle is eliminated
- ▶ In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?





# Recovery from Deadlock: Resource Preemption

- ▶ **Selecting a victim** – minimize cost
- ▶ **Rollback** – return to some safe state, restart process for that state
- ▶ **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

