



Amirkabir University of Technology  
(Tehran Polytechnic)  
Department of Computer Engineering

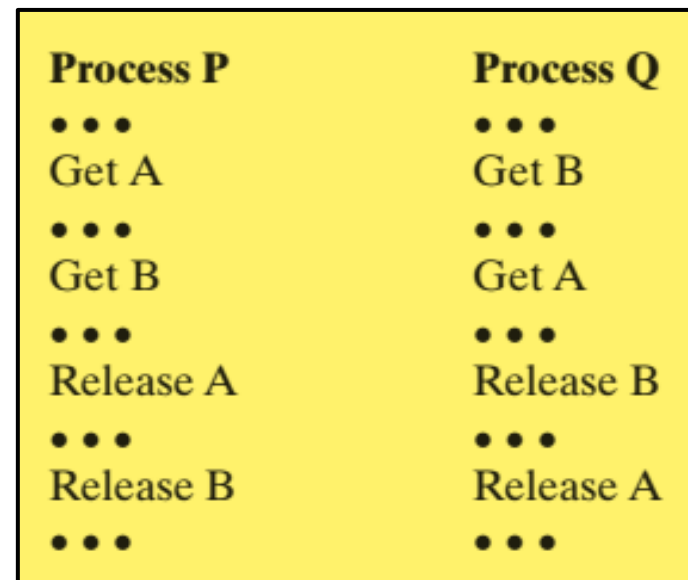
---

# Deadlock, Starvation

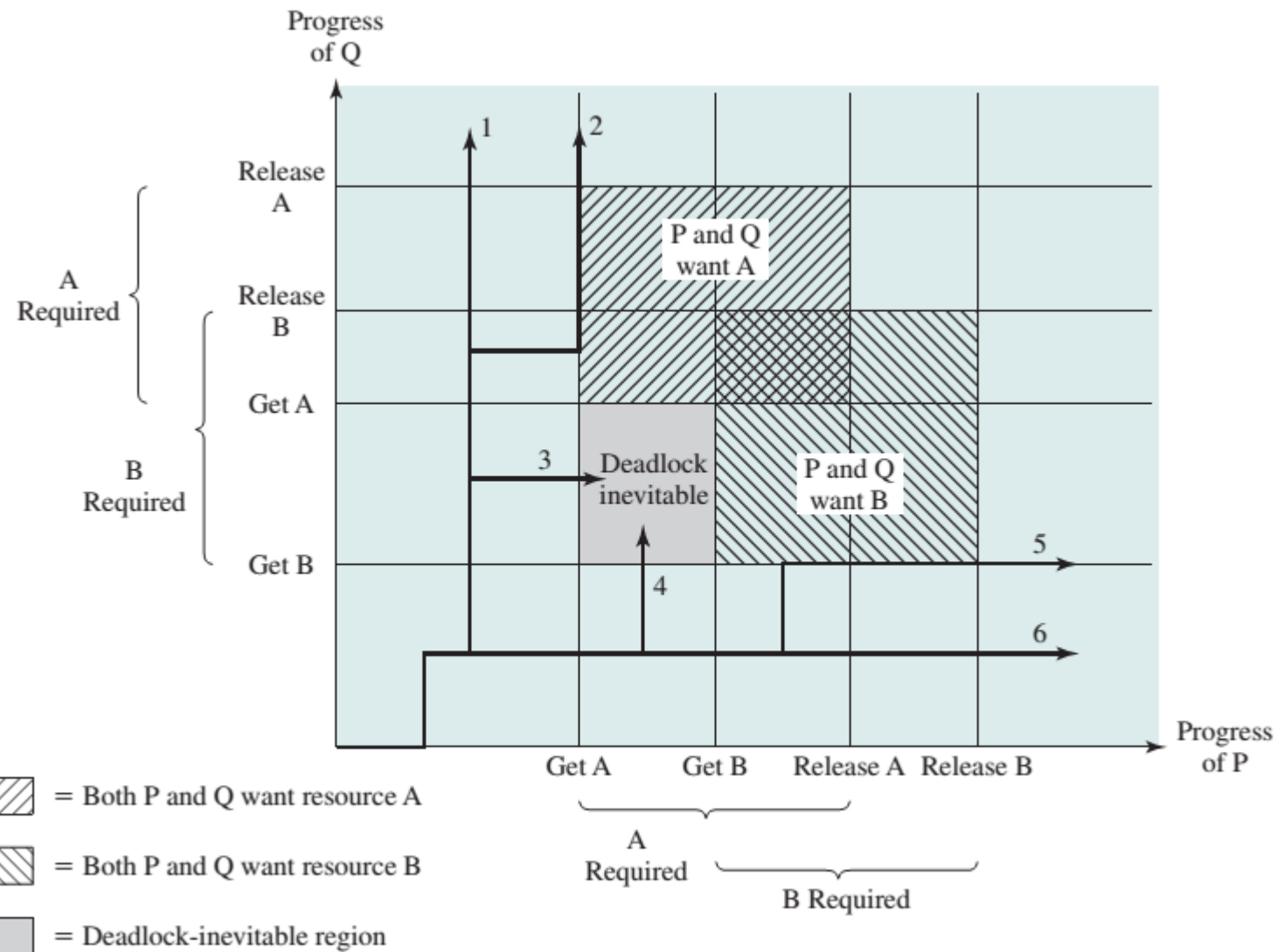
Hamid R. Zarandi  
h\_zarandi@aut.ac.ir

# Why is deadlock handling important?

- Deadlocks prevent sets of concurrent processes from completing their tasks



# Deadlock diagram of 2 processes

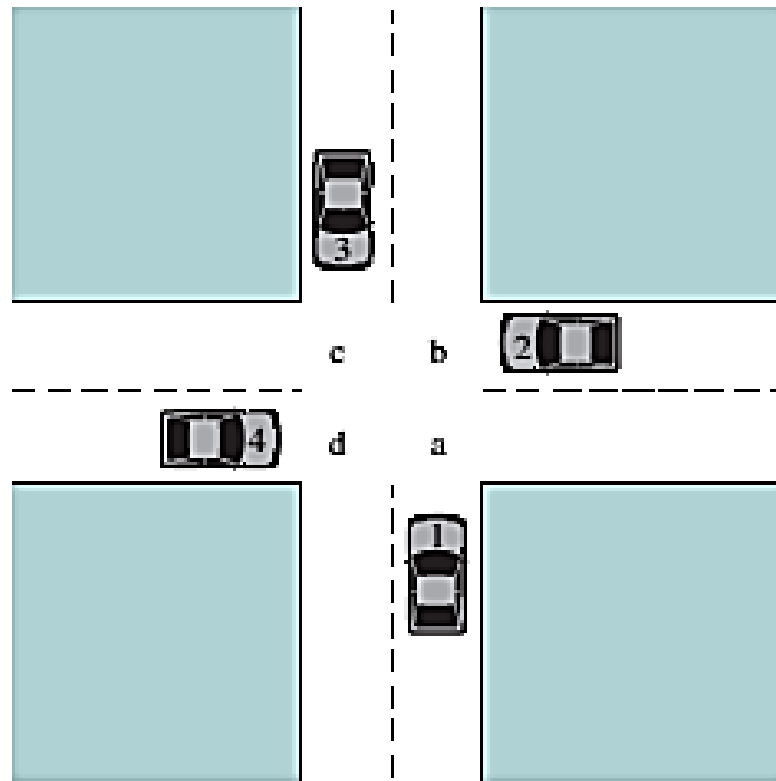


# Deadlock

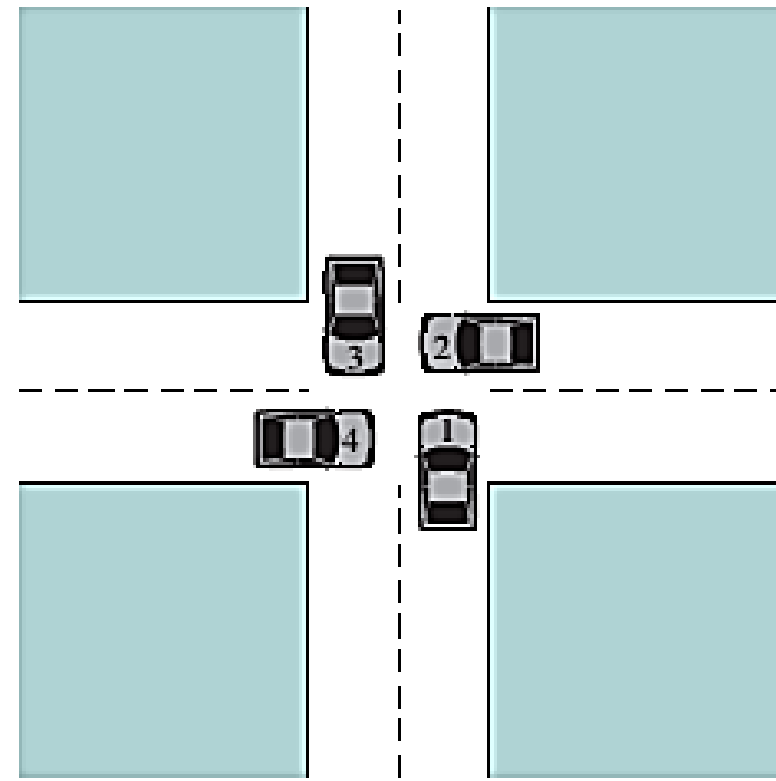
## ➤ Definition

- To wait for a resource which is to acquired by another process that is waited for a resource of requesting process
- Never finishing wait state
- Circular dependencies between processes

# Illustration of deadlock

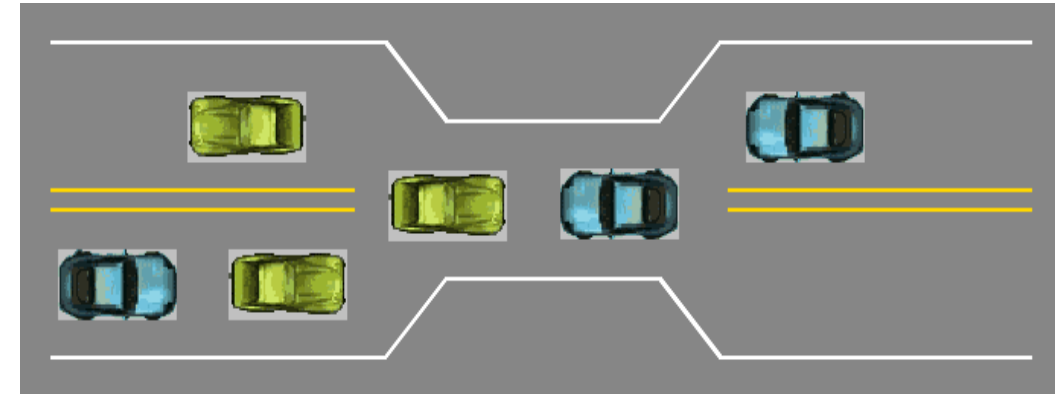


(a) Deadlock possible



(b) Deadlock

# Deadlock



# Necessary conditions

## ➤ 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, after that process has completed its task

## ➤ Circular wait

- there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# How to model deadlock?

- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

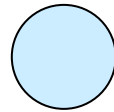


# Resource-allocation graph

## ➤ Directed graph

- Nodes = {Processes, Resources}
- Edges = {Request edges:  $P_i \rightarrow R_j$ , Assignment edges:  $R_j \rightarrow P_i$ }

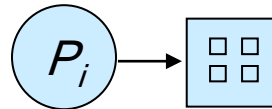
- Process



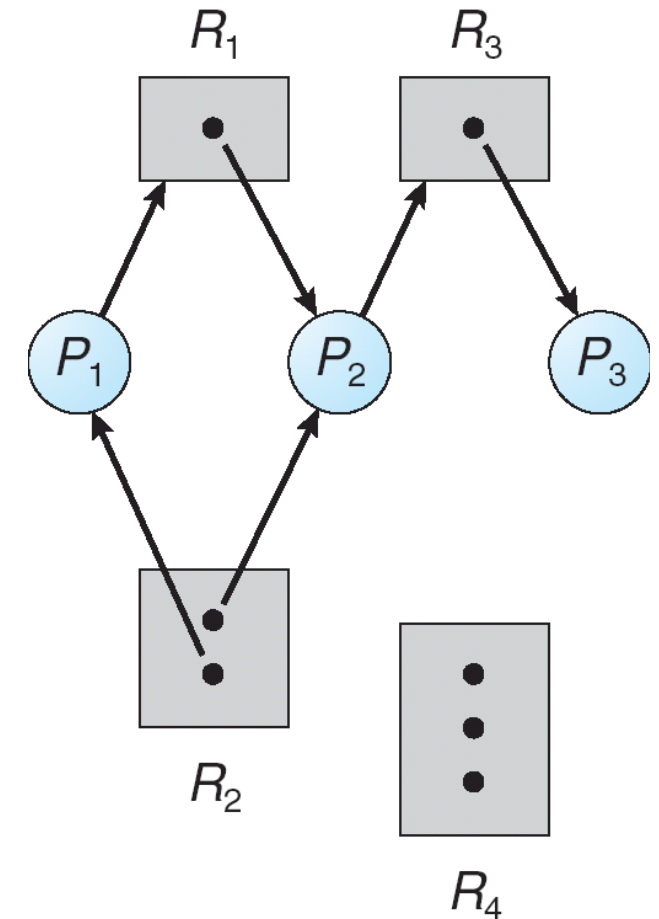
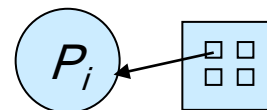
- Resource Type with 4 instances



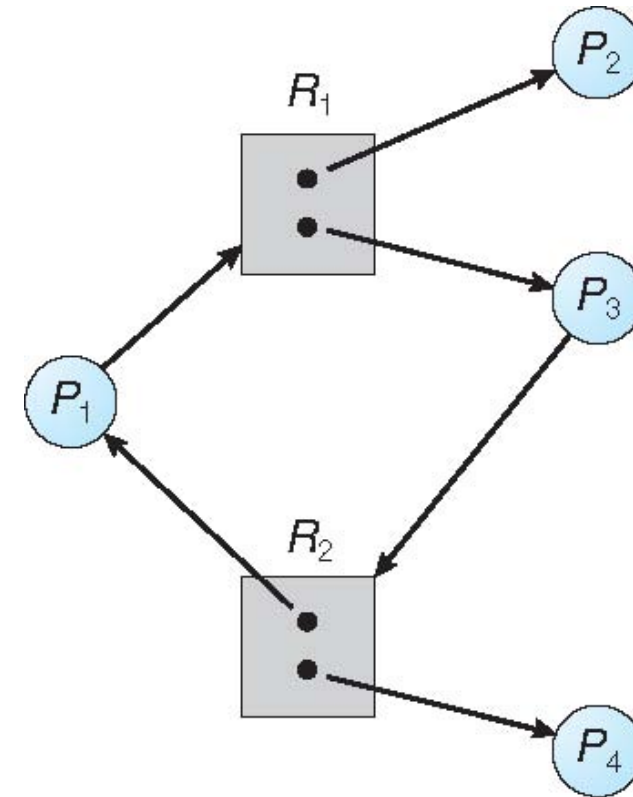
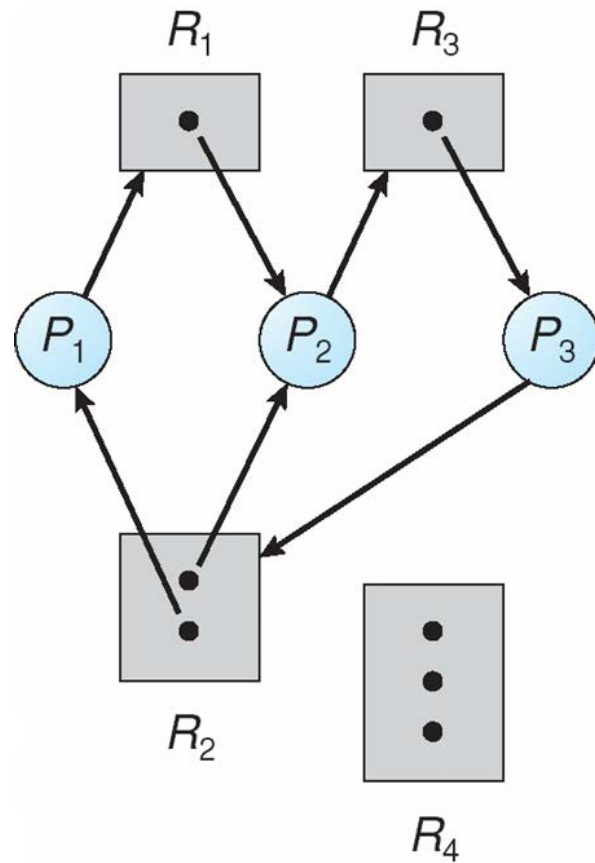
- $P_i$  requests instance of  $R_j$



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



# Is there a deadlock?



# Deadlock illustration in RAG

- Having 1 resource in each resource type
  - Deadlock  $\Leftrightarrow$  existing a cycle
  
- Having multiple resources for at least one resource type
  - Deadlock is possible (not necessary) when existing a cycle
  
- No Cycle  $\rightarrow$  No deadlock
- One cycle  $\rightarrow$ 
  - (one instance per resource type): deadlock
  - (multiple instances per resource type): possible of deadlock

# How to handle deadlocks?

## ➤ 1) Prevent or 2) avoid deadlocks

### ○ Deadlock prevention

- To ensure at least one of necessary condition cannot hold.

### ○ Deadlock avoidance

- Conservative: May current req make a deadlock in future (non safe state)?

## ➤ 3) Detect & recover deadlocks

- 1st detect, next recover (how?)

## ➤ 4) Do nothing: Ostrich algorithm!

- Modern OS: Windows, Linux

# 1) Deadlock prevention (جلوگیری)

## ➤ Prevent deadlock by missing one of:

### ○ ~ **Mutual exclusion**

- Make resources sharable: read-only files
- Is not possible in all cases

### ○ ~ **Hold & wait**

- How?
  - ✓ Request all resources before execution
  - ✓ Request a resource if no others it have
- Drawbacks
  - ✓ Underutilization of resources
  - ✓ Starvation

### ○ ~ **No preemption**

- 3 solutions exist = {self preemption, dest process preemption, save & switch resource status}

### ○ ~ **Circular wait**

- Request a resource in an increasing order of enumeration

# Example of prevention of circular wait

## ➤ Example of ~circular wait

- $F(\text{tape drive}) = 1$
- $F(\text{disk drive}) = 5$
- $F(\text{printer}) = 12$
- After having resource type  $R_i$  request to resource type  $R_j$  is possible if  $F(R_j) > F(R_i)$
- Otherwise, release all resource types  $R_k$  where  $F(R_k) \geq F(R_j)$

## 2) Deadlock avoidance (اجتناب)

### ➤ Prevention is bad (why?)

- Resource underutilization
- Reduced throughput

### ➤ Avoidance is good; How to avoid?

- Required extra info
  - Available resources
  - Resources allocated to processes
  - Future request of processes (!)
- Definition:
  - Safe state
- Solutions
  - 1) Resource-allocation-graph algorithm (single instance resource type)
  - 2) Banker's algorithm (multiple instance resource type)

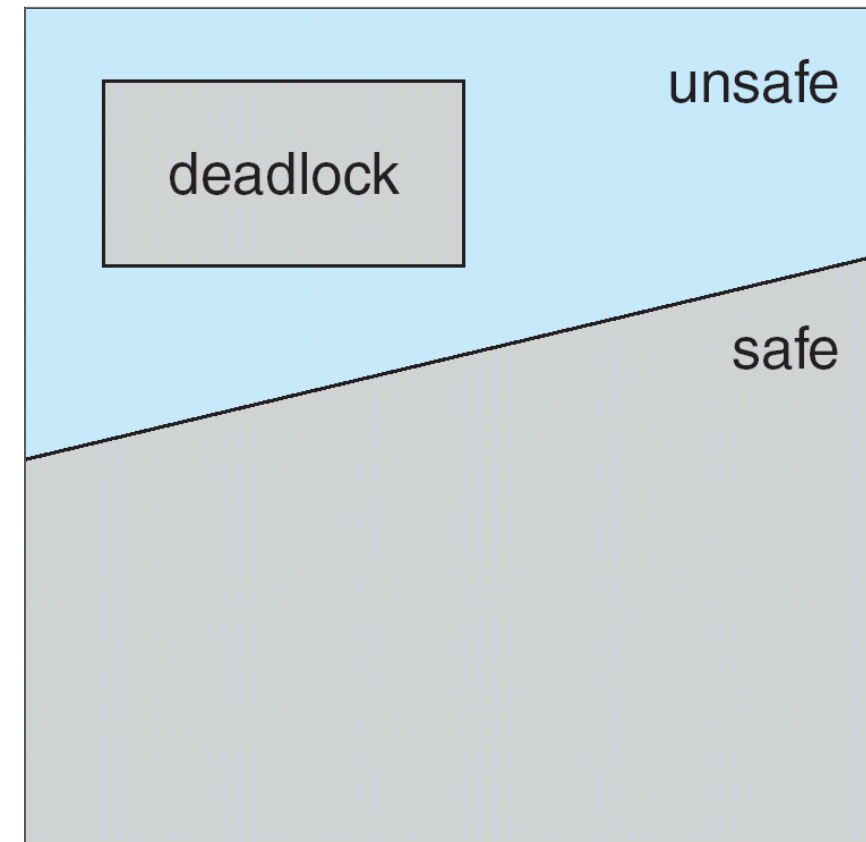
# Safe state definition

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- **Safe state:** there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of **ALL** the processes in the systems 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 the  $P_j$  with  $j < i$**
- That is:
  - If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
  - 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



# Safe state definition

- If a system is in **safe state**  $\Rightarrow$  **no deadlocks**
- If a system is in **unsafe state**  $\Rightarrow$  **possibility of deadlock**
- Avoidance  $\Rightarrow$  **ensure that a system will never enter an unsafe state.**



# Safe mode example (1)

- 3 processes:  $P_0$ ,  $P_1$ ,  $P_2$
- 1 resource type:  $A$  (12)
- Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- Safe mode sequence?  $\langle P_1, P_0, P_2 \rangle$

# Safe mode example (2)

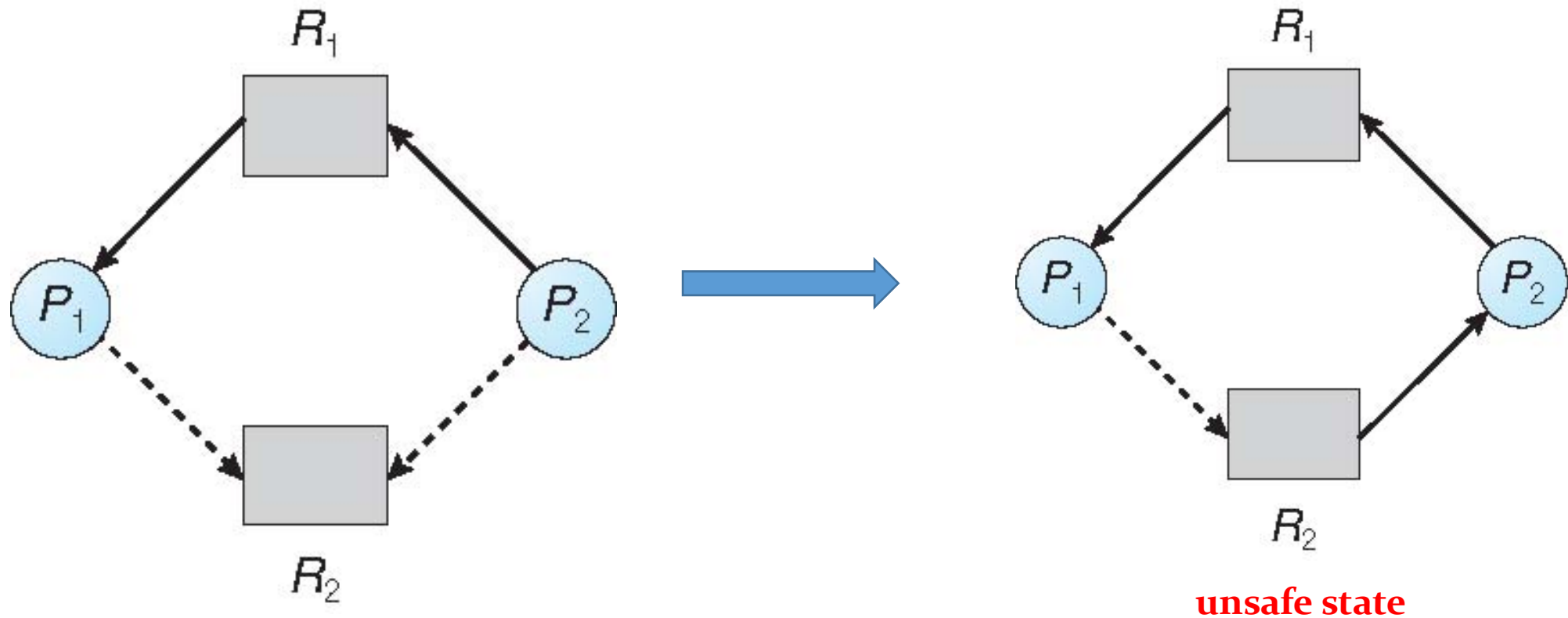
- 3 processes:  $P_0, P_1, P_2$
- 1 resource type:  $A$  (12)
- Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- Suppose that, at time  $T_1$ , process  $P_2$  requests and is allocated one more resource.
- Safe mode sequence? **no safe**

## 2.1) Resource-allocation graph (RAG) algorithm

- **Claim edge**  $P_i \rightarrow R_j$  indicated that process  $P_i$  **may** request resource  $R_j$  represented by a dashed line



## 2.1) Resource-allocation graph (RAG) algorithm

Suppose that process  $P_i$  requests a resource  $R_j$

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

## 2.2) Banker's algorithm

Let  $n$  = number of processes, and  $m$  = number of resources types.

**Available:** Vector of length  $m$ .

If  $available[j] = k$ , there are  $k$  instances of resource type  $R_j$  available

**Max:**  $n \times m$  matrix.

If  $Max[i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$

**Allocation:**  $n \times m$  matrix.

If  $Allocation[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$

**Need:**  $n \times m$  matrix.

If  $Need[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

$Need[i,j] = Max[i,j] - Allocation[i,j]$

## 2.2) Banker's algorithm: Safety algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:  
**Work = Available**  
**Finish [i] = false for  $i = 0, 1, \dots, n-1$**
2. Find an  $i$  such that both:
  - (a) **Finish [i] = false**
  - (b) **Need<sub>i</sub> ≤ Work**If no such  $i$  exists, go to step 4
3. **Work = Work + Allocation<sub>i</sub>**  
**Finish[i] = true**  
go to step 2
4. If **Finish [i] == true** for all  $i$ , then the system is in a **safe state**

## 2.2) Banker's algorithm: Resource-request algorithm for process $P_i$

**$Request_i$**  = 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  **$Request_i \leq Need_i$**

go to step 2

else

raise error condition (process has exceeded its maximum claim)

2. If  **$Request_i \leq Available$**

go to step 3

else

$P_i$  must **wait** (resources are not available)

3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

If **safe**  $\Rightarrow$  the resources are allocated to  $P_i$

If **unsafe**  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored



# Banker's algorithm example

5 processes  $P_0$  through  $P_4$

3 resource types:  $A = 10$     $B = 5$     $C = 7$

Snapshot at time  $T_0$ :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
$P_0$	0 1 0	7 5 3	3 3 2
$P_1$	2 0 0	3 2 2	
$P_2$	3 0 2	9 0 2	
$P_3$	2 1 1	2 2 2	
$P_4$	0 0 2	4 3 3	

# Banker's algorithm example

The content of the matrix *Need* is defined to be *Max* – *Allocation*

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>		<u>Need</u>
	A B C	A B C	A B C		A B C
$P_0$	0 1 0	7 5 3	3 3 2	$P_0$	7 4 3
$P_1$	2 0 0	3 2 2		$P_1$	1 2 2
$P_2$	3 0 2	9 0 2		$P_2$	6 0 0
$P_3$	2 1 1	2 2 2		$P_3$	0 1 1
$P_4$	0 0 2	4 3 3		$P_4$	4 3 1

Is the system safe?

Yes. The sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria

# Example: $P_1$ Request (1,0,2)

Check that  $\text{Request} \leq \text{Available}$  (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$ )

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

Executing **safety algorithm** shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement

Can request for (3,3,0) by  $P_4$  be granted?

Can request for (0,2,0) by  $P_0$  be granted?

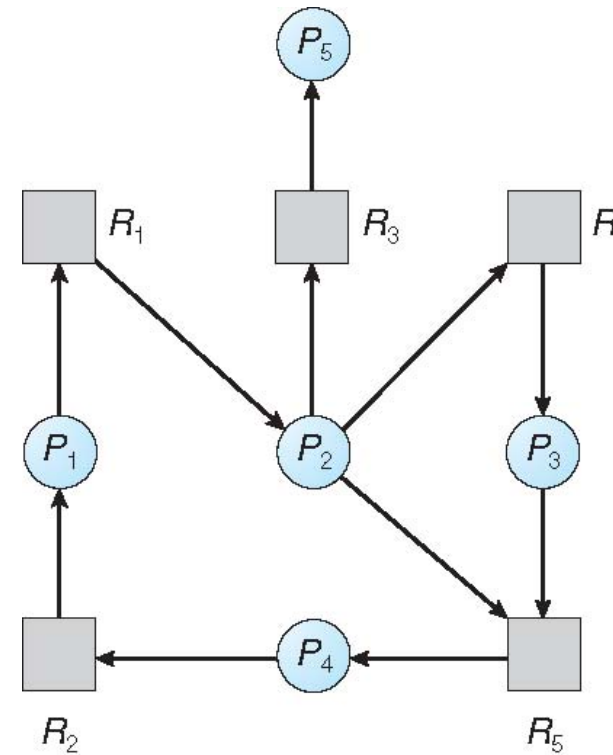
### 3) Deadlock detection & recovery

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

# 3.1) Resource-allocation graph and wait-for graph

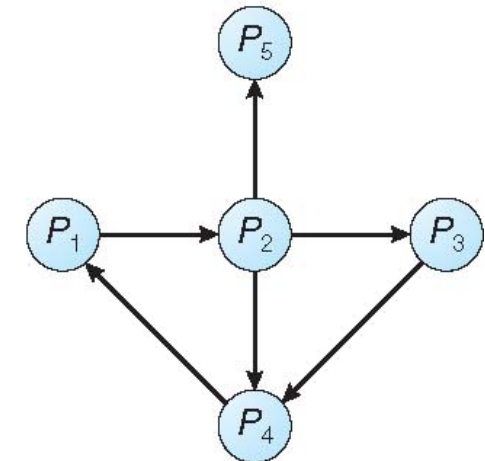
➤ For single instance resource type:

- Create wait-for graph from RAG
- Periodically run cycle detector



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

## 3.2) Several instances of a resource type

Let  $n$  = number of processes, and  $m$  = number of resources types.

**Available:** Vector of length  $m$ .

If  $\text{available}[j] = k$ , there are  $k$  instances of resource type  $R_j$  available

**Allocation:**  $n \times m$  matrix.

If  $\text{Allocation}[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$

**Request:**  $n \times m$  matrix.

If  $\text{Request}[i,j] = k$ , then  $P_i$  is requesting  $k$  more instances of  $R_j$

## 3.2) Several instances of a resource type

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:  
 $\text{Work} = \text{Available}$   
 for  $i = 0, 1, \dots, n-1$  if  $\text{Allocation}_i \neq 0$  then  $\text{Finish}[i] = \text{false}$   
                                   else  $\text{Finish}[i] = \text{true}$
2. Find an  $i$  such that both:
  - (a)  $\text{Finish}[i] = \text{false}$
  - (b)  $\text{Request}_i \leq \text{Work}$
 If no such  $i$  exists, go to step 4
3.  $\text{Work} = \text{Work} + \text{Allocation}_i$   
 $\text{Finish}[i] = \text{true}$   
 go to step 2
4. If  $\text{Finish}[i] == \text{false}$  for some  $i$ , then the system is in a **deadlock state**

# Example of detection algorithm

Five processes  $P_0$  through  $P_4$ ; three resource types  
A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time  $T_0$ :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	0 0 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	

Deadlock?

Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = true$  for all  $i$



# Example (Cont.)

$P_2$  requests an additional instance of type  $C$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>		<u>Request</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$		$A \ B \ C$
$P_0$	0 1 0	0 0 0	0 0 0	$P_0$	0 0 0
$P_1$	2 0 0	2 0 2		$P_1$	2 0 2
$P_2$	3 0 3	0 0 0		$P_2$	0 0 1
$P_3$	2 1 1	1 0 0		$P_3$	1 0 0
$P_4$	0 0 2	0 0 2		$P_4$	0 0 2

## State of system?

Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes; requests

**Deadlock exists**, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$

# 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

# Questions?

