

CYB 510 OPERATING SYSTEMS

Prepared by Prof. Dr. Lamiaa Fattouh

Lecture 6

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

CYB 510
Deadlock



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

- ❑ The Deadlock Problem
- ❑ System Model
- ❑ Deadlock Characterization
- ❑ Methods for Handling Deadlocks
- ❑ Deadlock Prevention
- ❑ Deadlock Avoidance
- ❑ Deadlock Detection
- ❑ Recovery from Deadlock



Chapter 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



The Deadlock Problem

- A set of **blocked processes** each holding a resource and **waiting** to acquire a resource **held by another** process in the set
- Example
 - System has 2 disk drives
 - P_1 and P_2 each hold one disk drive and each needs another one
- Example
 - **semaphores A** and **B**, initialized to 1

P_1

wait (A);

wait (B);

P_2

wait(B)

wait(A)

Will never reach signal (S);

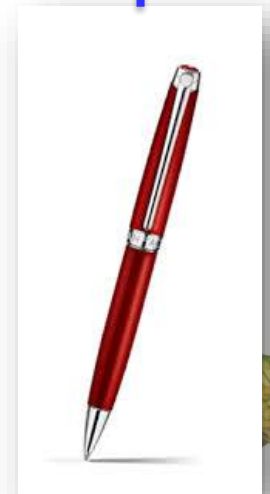


- ❑ To write a note, one needs a **pen** and a **paper**
- ❑ Two persons want to write notes
- ❑ P1 (**Man**) has the pen
- ❑ P2 (**Woman**) has the paper
- ❑ Each is **holding the resource** (non-preemptive)
- ❑ Both will keep **waiting for ever**

Processes

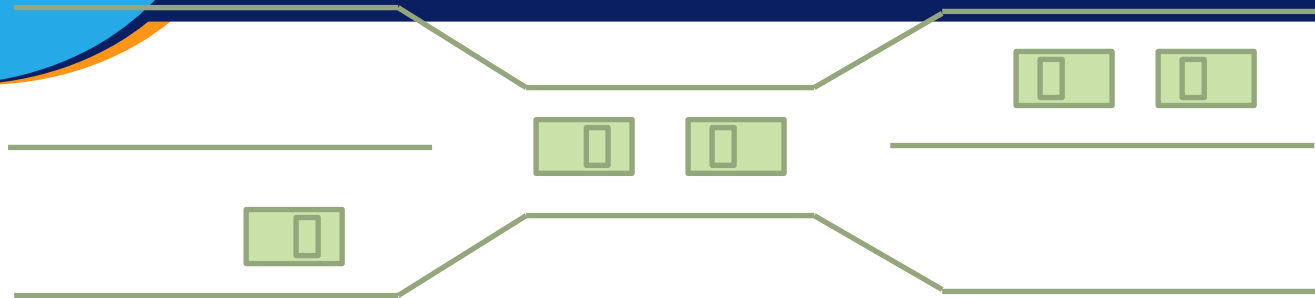


Resources



shutterstock.com • 663404344

Bridge Crossing Example



- ❑ **Traffic** only in one direction
- ❑ **Each section** of a **bridge** can be viewed as a **resource**
- ❑ If a **deadlock occurs**, it can be **resolved if** one car backs up
 - ❑ **preempt resources** and rollback
- ❑ **Several cars** may have to be **backed up** if a **deadlock** occurs



Starvation is **possible**

Note : Most OSes do **not prevent** or **deal** with **deadlocks**



Deadlock Principles

➤ A deadlock is a permanent blocking of a set of threads

- ✓ a deadlock can happen while threads/processes are competing for system resources or communicating with each other

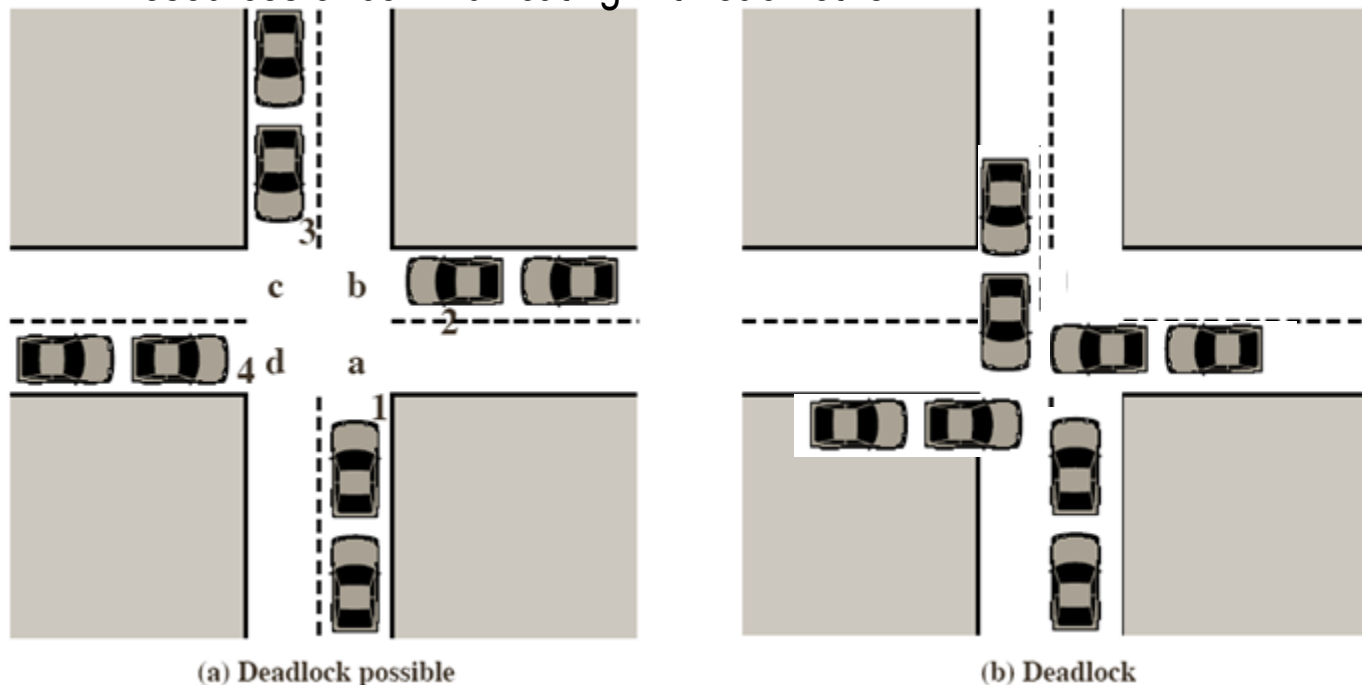


Illustration of a deadlock

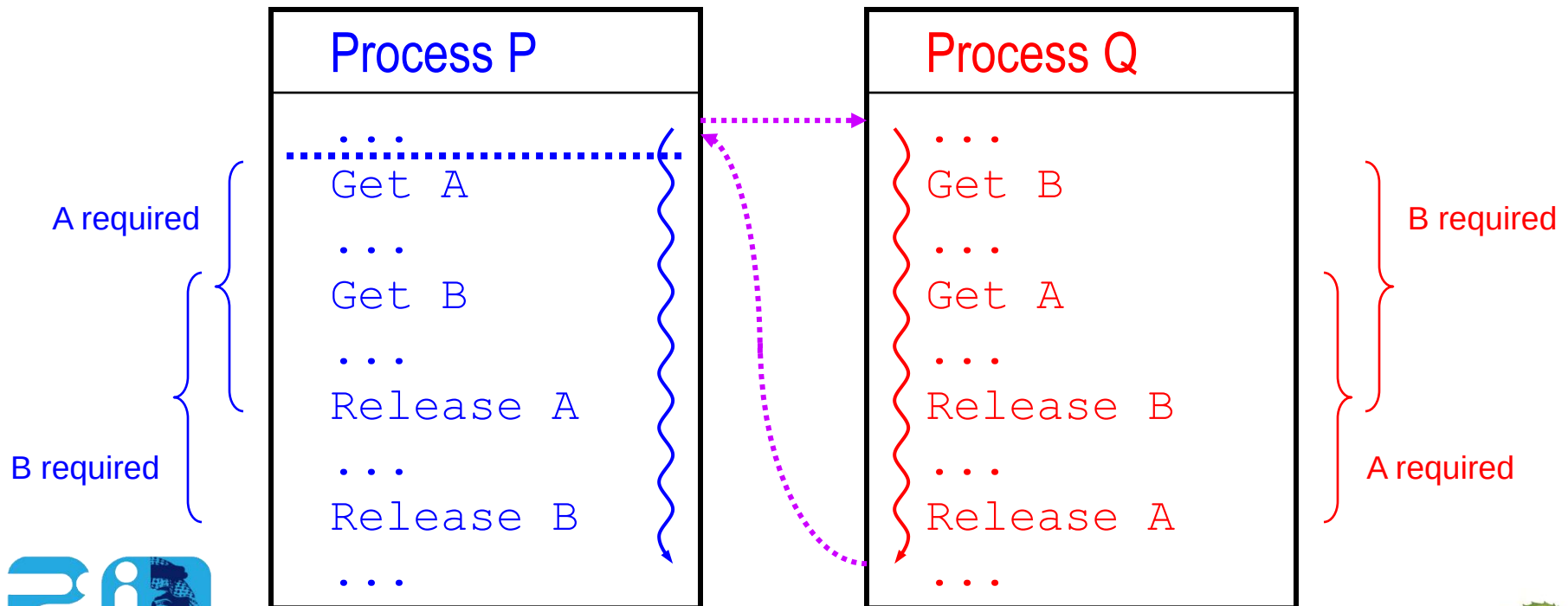
The rule of the road in the United States is that a car at a four-way stop should defer to a car immediately to its right



Deadlock Principles

➤ Illustration of a deadlock — scheduling path 1 😊

- ✓ Q executes everything before P can ever get A
- ✓ when P is ready, resources A and B are free and P can proceed



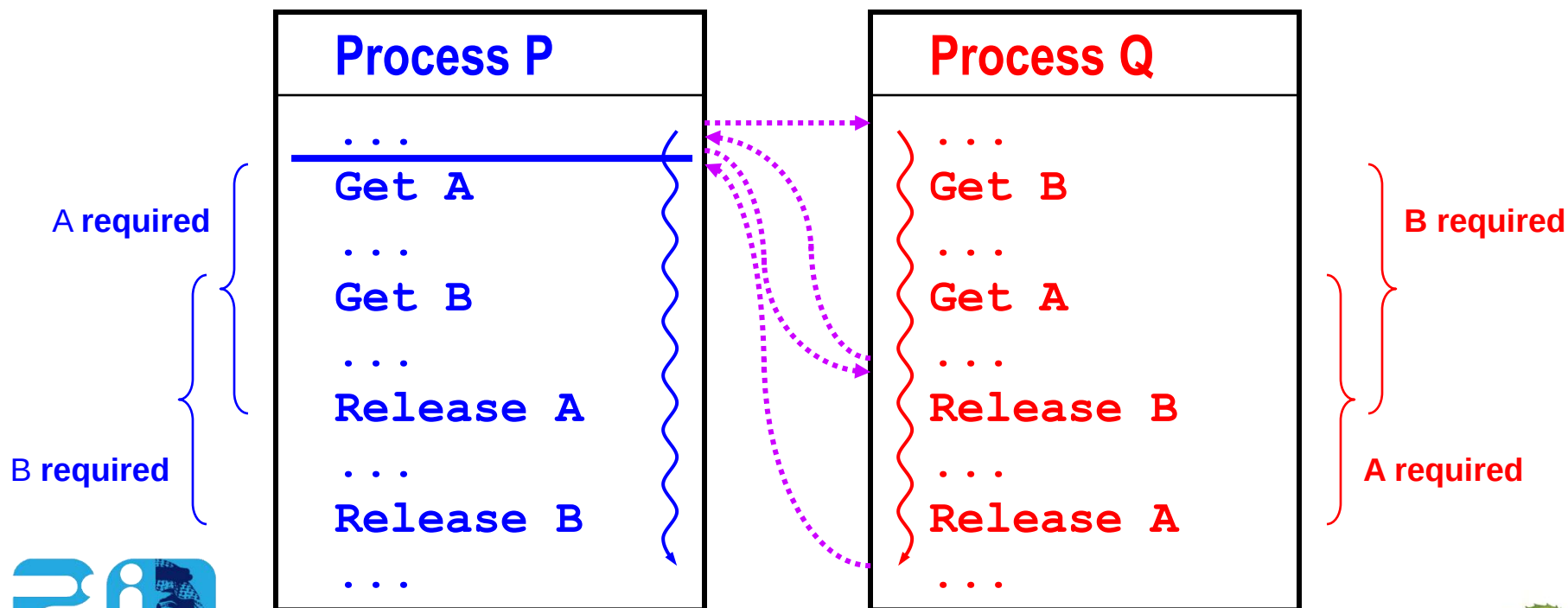
Happy scheduling 1



Deadlock Principles

➤ Illustration of a deadlock — scheduling path 2 😊

- ✓ Q gets B and A, then P is scheduled; P wants A but is blocked by A's mutex; so Q resumes and releases B and A; P can now go



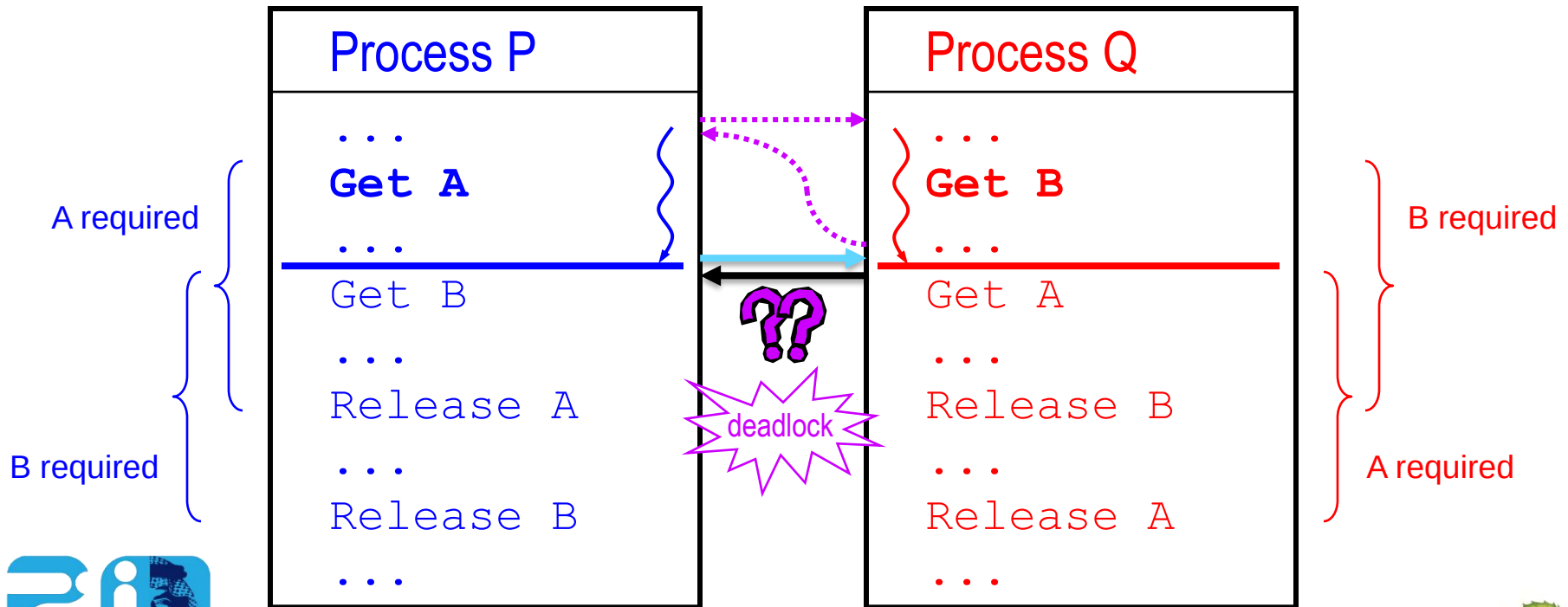
Happy scheduling 2



Deadlock Principles

➤ Illustration of a deadlock — scheduling path 3 😞

- ✓ Q gets only B, then P is scheduled and gets A; now both P and Q are blocked, each waiting for the other to release a resource



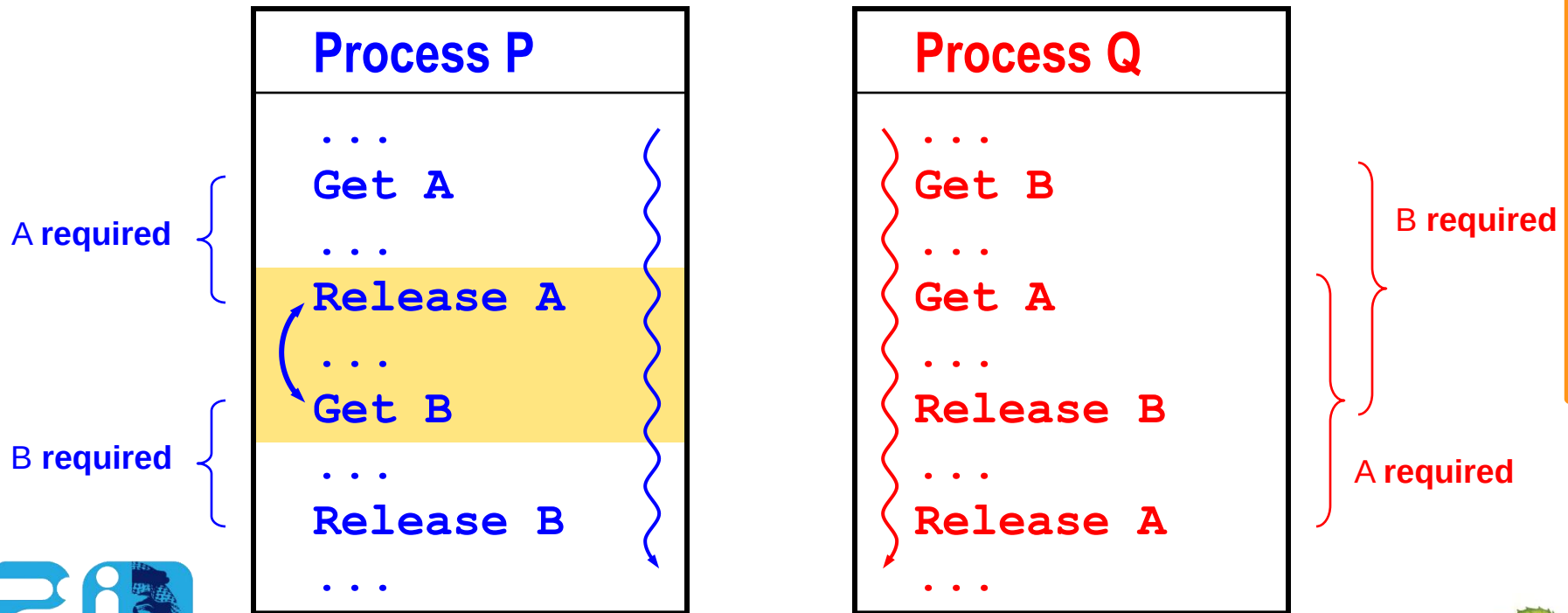
Bad scheduling → deadlock



Deadlock Principles

Changing the program

- ✓ here, P releases A before getting B
- ✓ deadlocks between P and Q are not possible anymore



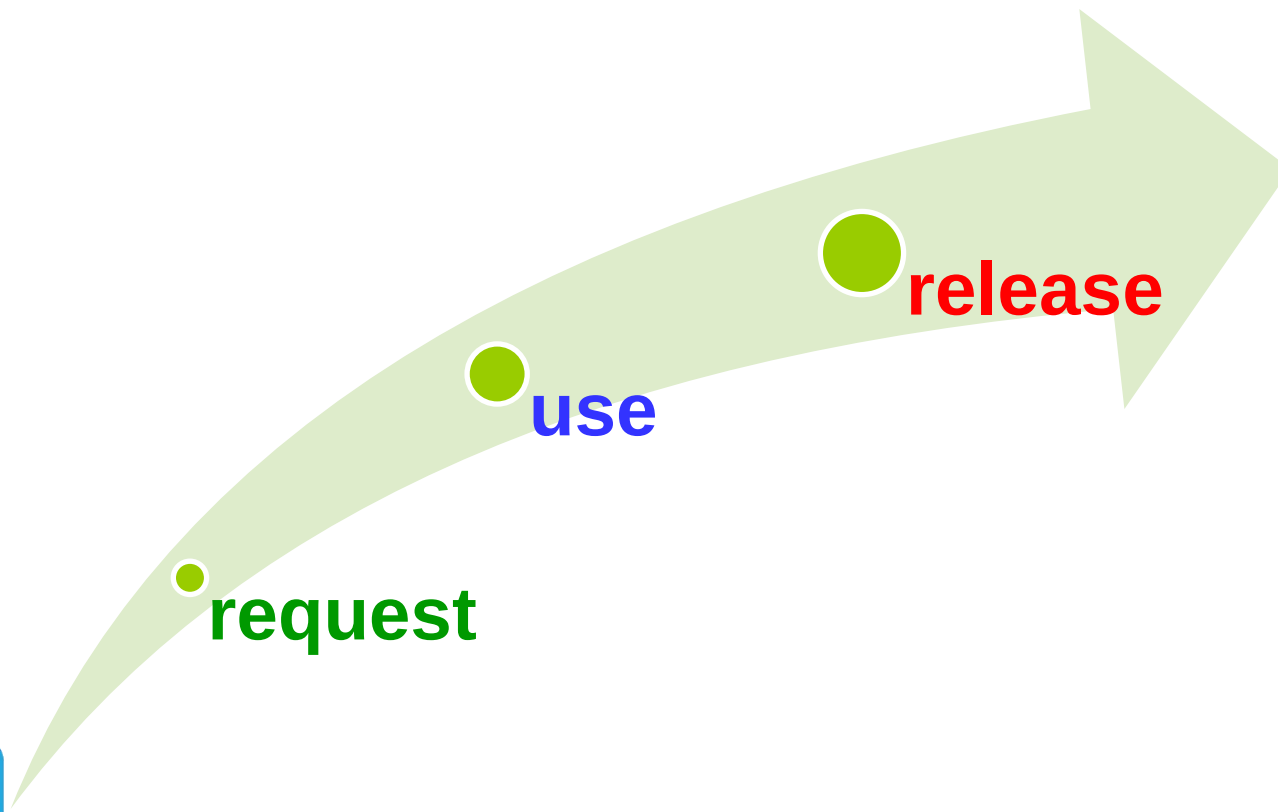
Competing processes



- **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:



Deadlock Characterization

Deadlock can **arise** if **four conditions** hold **simultaneously**.

Mutual exclusion

Hold and wait

No preemption

Circular wait



Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

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

Non
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 .



Resource-Allocation Graph

A set of vertices V and a set of edges E .

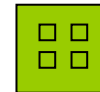
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_1 \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$



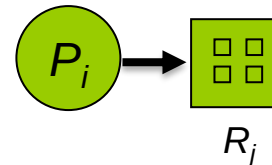
□ **Process**



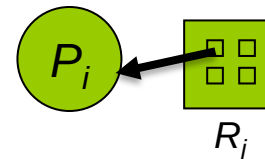
□ **Resource** Type with 4 instances



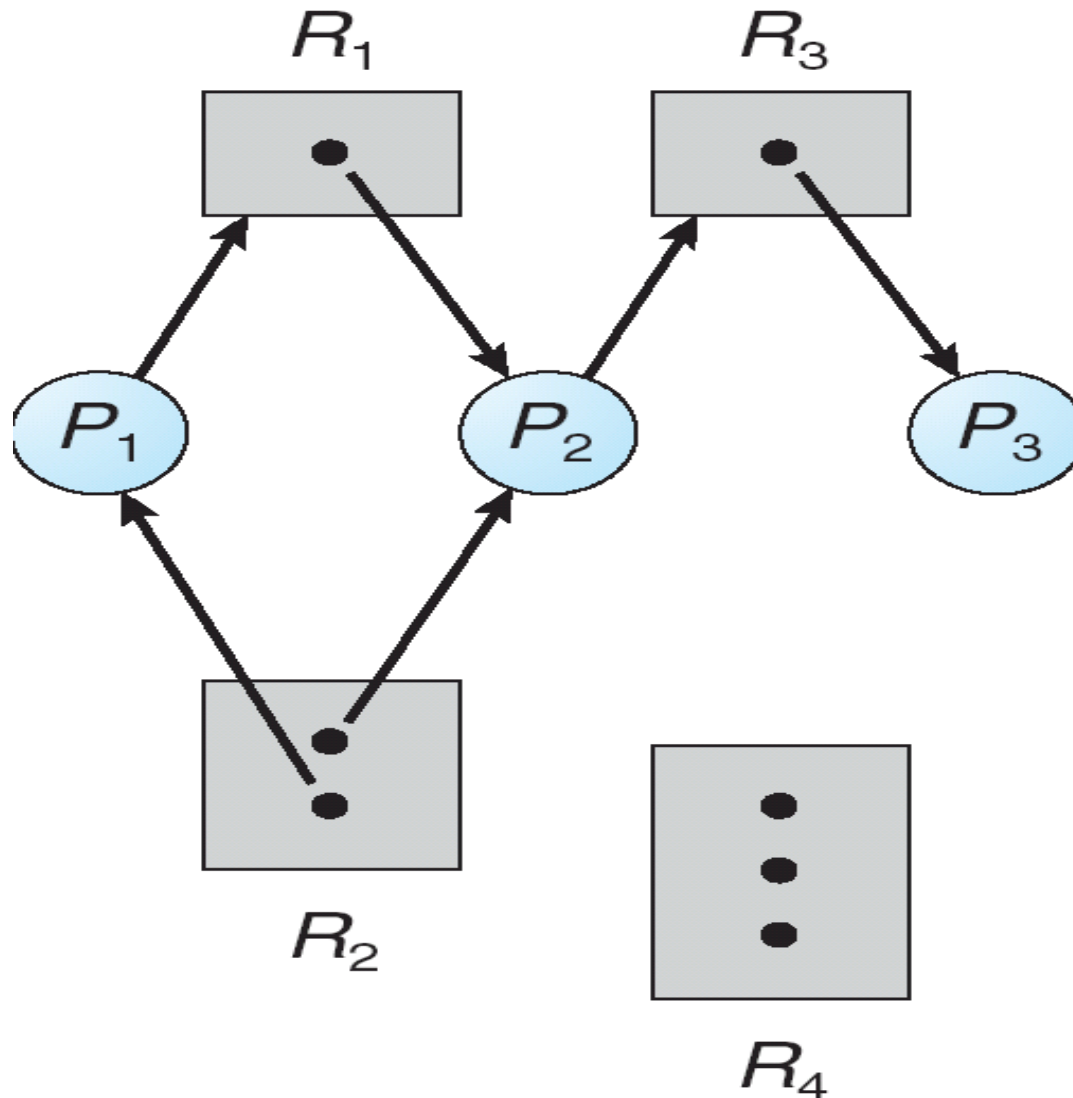
□ P_i **requests** instance of R_j

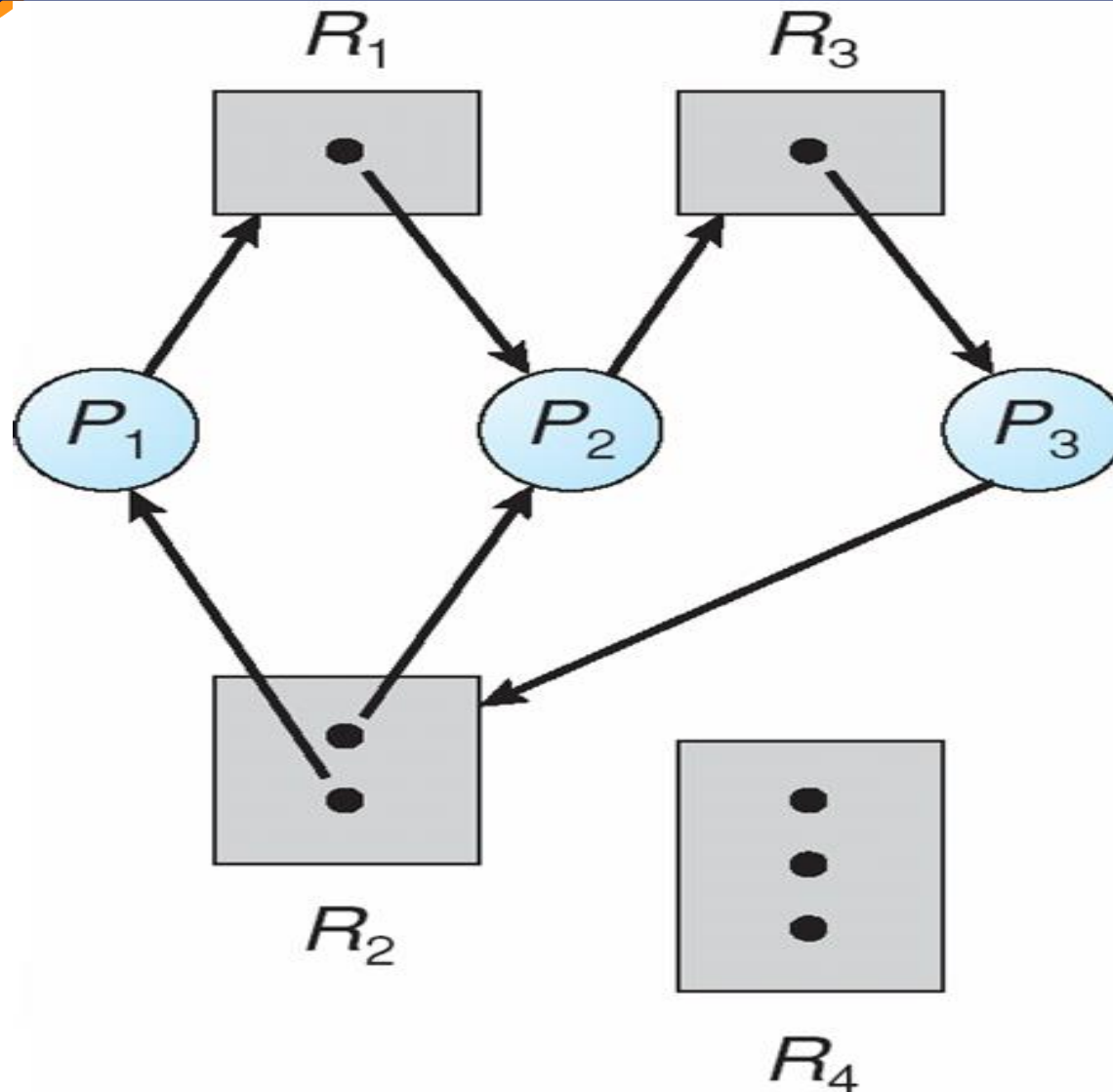


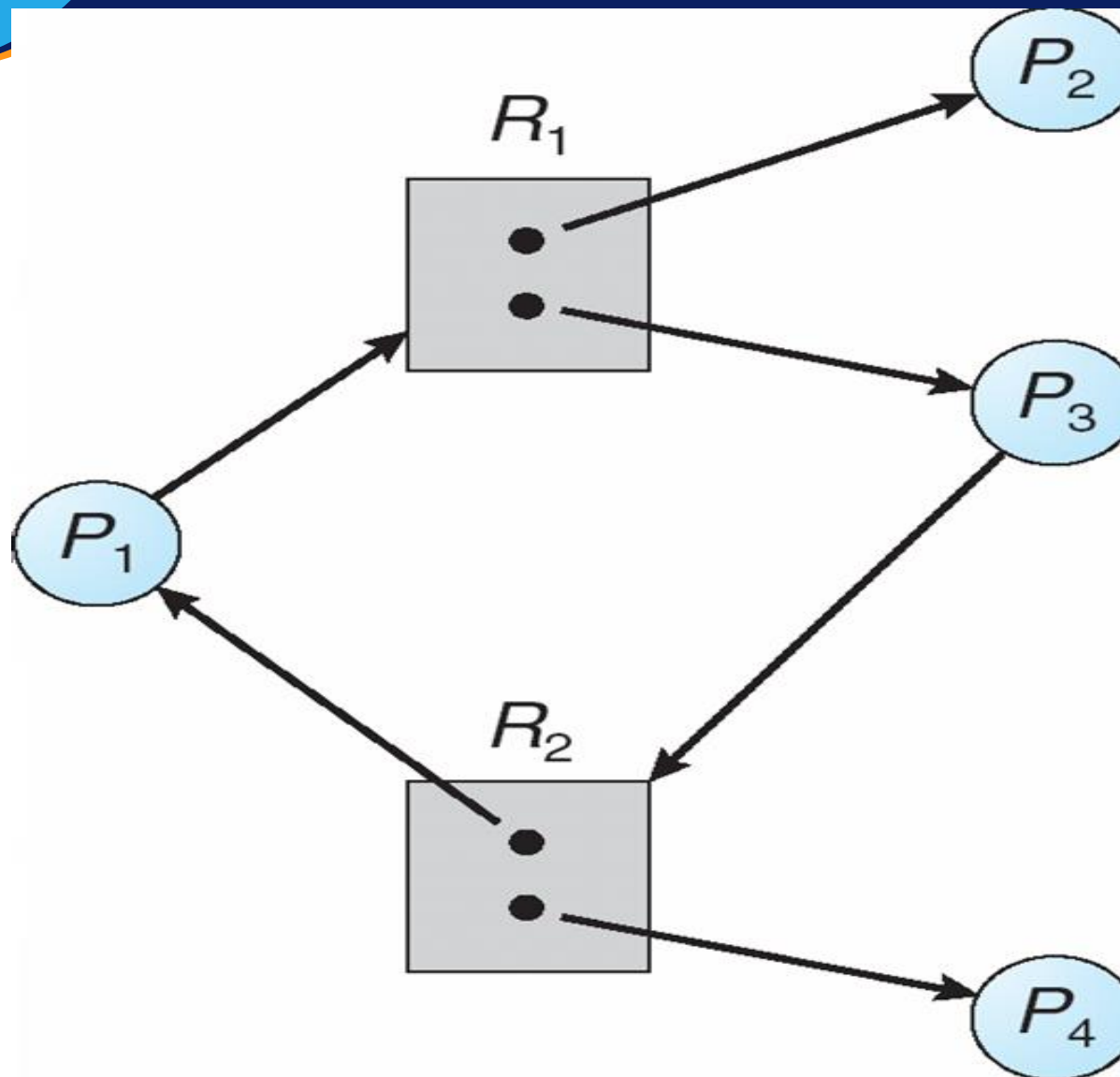
□ P_i is **holding** an instance of R_j



Example of a Resource Allocation Graph

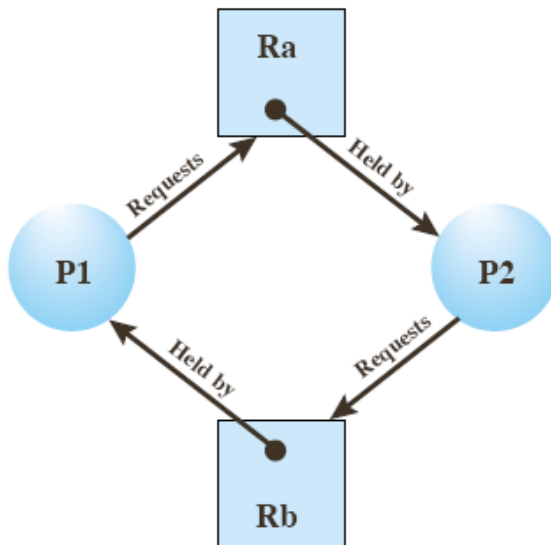




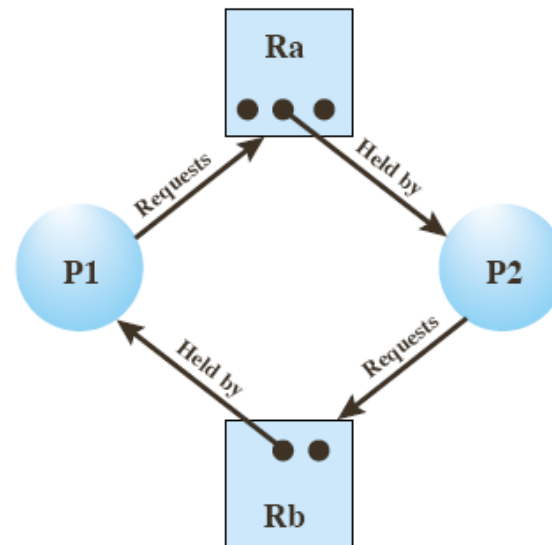


Basic Facts

- If **graph** contains **no cycles** $\Rightarrow$ **no deadlock**
- If graph **contains a cycle** $\Rightarrow$
 - if only **one instance** per **resource type**, then **deadlock**
 - if **several instances** per resource type, **possibility** of deadlock



(c) Circular wait



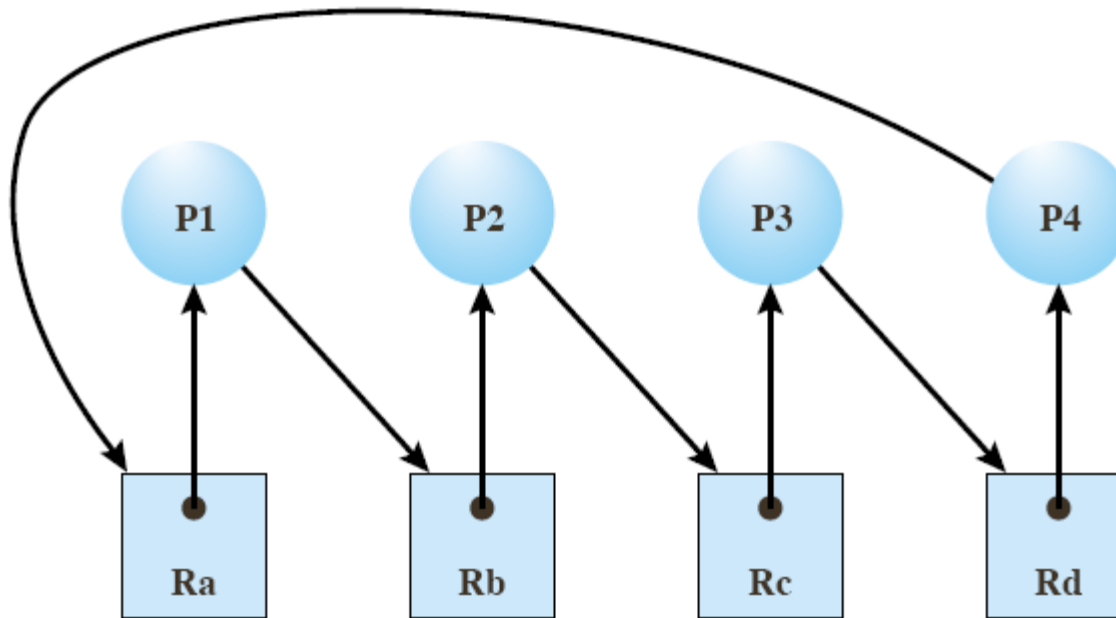
(d) No deadlock



Chapter 7

Resource allocation graphs & deadlocks

- ✓ there is **deadlock** when a **closed chain of processes** exists
- ✓ **each process holds at least one resource** needed **by the next** process
- ✓ a **cycle** illustrates deadlock



A deadlock



- **Ensure** that the system will *never* enter a **deadlock** state

deadlock prevention

- **design** decision

deadlock avoidance

- **runtime** decision

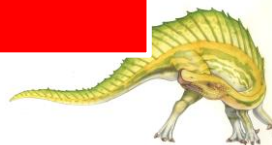


Allow the system to **enter a deadlock** state and then **recover**

- **deadlock recovery**
 - periodic runtime decision

Ignore the **problem** and pretend that deadlocks **never occur** in the system;

- used by **most operating systems**, including UNIX



Methods for Handling Deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock Recovery

Ignoring Deadlock



Deadlock Prevention

Deadlock Avoidance

Deadlock recovery

Ignoring Deadlock



1 **Request**

2 **Use**

3 **Release**



- ❑ **Deadlock** may happen in **4 conditions**
 - ❑ **Each** of the four necessary conditions **must hold**. By ensuring that **at least one of** these conditions cannot hold, we can **prevent** the occurrence of a deadlock.



Mutual exclusion
Non preemption
Hold and wait
Circular wait



Deadlock Prevention

Stop the ways **request** can **be made**

- ❑ **Mutual Exclusion** – **not** required for **sharable** resources;
 - ❑ **must hold** for **nonsharable** resources
 - ❑ **Sharable resources**, in contrast, do **not require mutually exclusive** access and thus **cannot** be involved in a **deadlock**.
 - ❑ **Read-only files** are a good example of a **sharable resource**.
 - ❑ ***not possible to control***

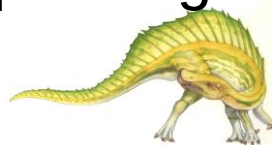


- **Hold and Wait** – **must guarantee** that **whenever** a **process requests a resource**, it does **not hold** any **other** resources
- **Require** process to **request and be allocated** all its resources **before** it begins **execution**,
 - ▶ **or allow** process to **request** resources **only** when the **process has none**
 - ▶ **Low resource utilization**; **starvation** possible
- **inefficient** and **impractical**:
 - ▶ creates **long waits**, **cannot predict** all resource **needs**



□ No Preemption –

- If a **process** that is **holding some resources** **requests another resource** that cannot be **immediately allocated** to it,
 - ▶ then **all resources** currently **being held** are **released**
- **Preempted resources** are added to the **list of resources** for which the process is **waiting**
- **Process** will be **restarted** only when it can **regain its old resources**,
 - ▶ **as well as the new ones** that it is requesting
- *possible starvation*



- ❑ **Circular Wait** – apply a **total ordering** of all **resource types**,
 - ❑ require that **each process requests** resources in an **increasing order of enumeration**
 - ❑ *inefficient*



Methods for Handling Deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock recovery

Ignoring Deadlock



Requires that the system has **some additional information** available

- **Simplest** and most **useful model** requires that **each process declare** the **maximum number** of **resources** of each type that it may **need**
- The **deadlock-avoidance** algorithm **dynamically examines** the **resource-allocation state** to ensure that there **can never be a circular-wait** condition
 - mainly a **scheduling problem**



- **Resource-allocation state** is defined by the number of **available** and **allocated** resources, and **the maximum demands** of the processes
 - **do not start** a process **if its demands** might **lead to deadlock**
 - **do not grant** an **incremental resource request** to a **running process** if this allocation might **lead to deadlock**



□ Safe State.

A **state** is **safe** if the system can **allocate all resources** requested by **all processes** (up to their stated maximums) **without** entering a **deadlock state**.

□ All **safe states** are **deadlock free**, but **not all unsafe states** lead to **deadlocks**.

□ **When a process requests** an **available resource**, system **must decide**

□ if **immediate allocation** leaves the system in a **safe state**

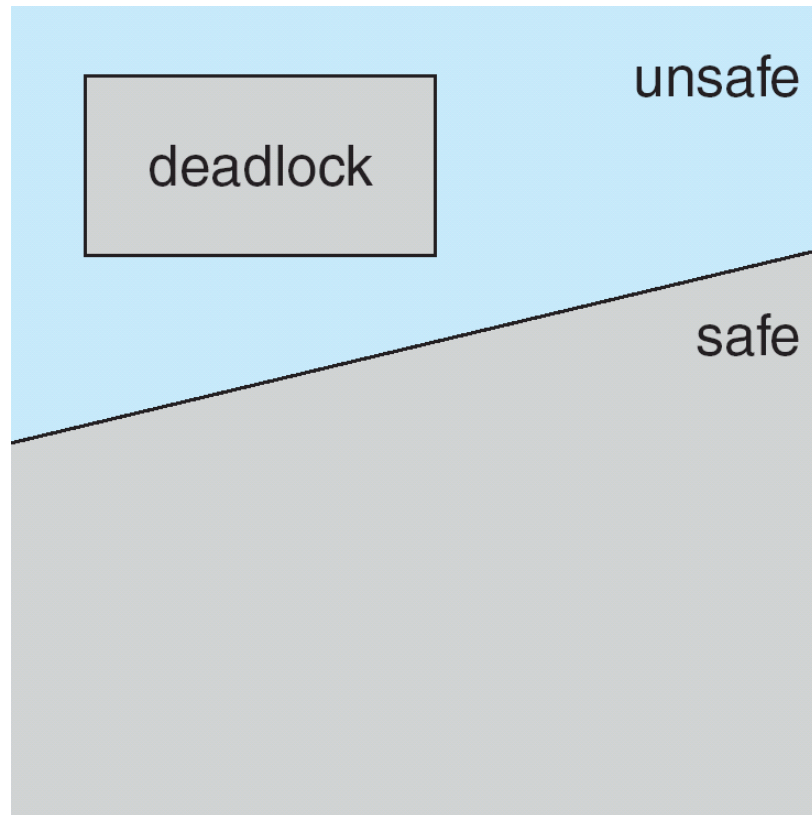


- System is **in safe state** if there exists a **sequence** $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the system such that
 - for **each** P_i , the **resources** that P_i can still **request** can be **satisfied** by currently **available resources** + **resources held** by **all 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



Basic Facts

- 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.



Avoidance algorithms

According to the **number** of the **instances** of **resource type**

Single instance of a resource type

- Use a **resource-allocation graph**

Multiple instances of a resource type

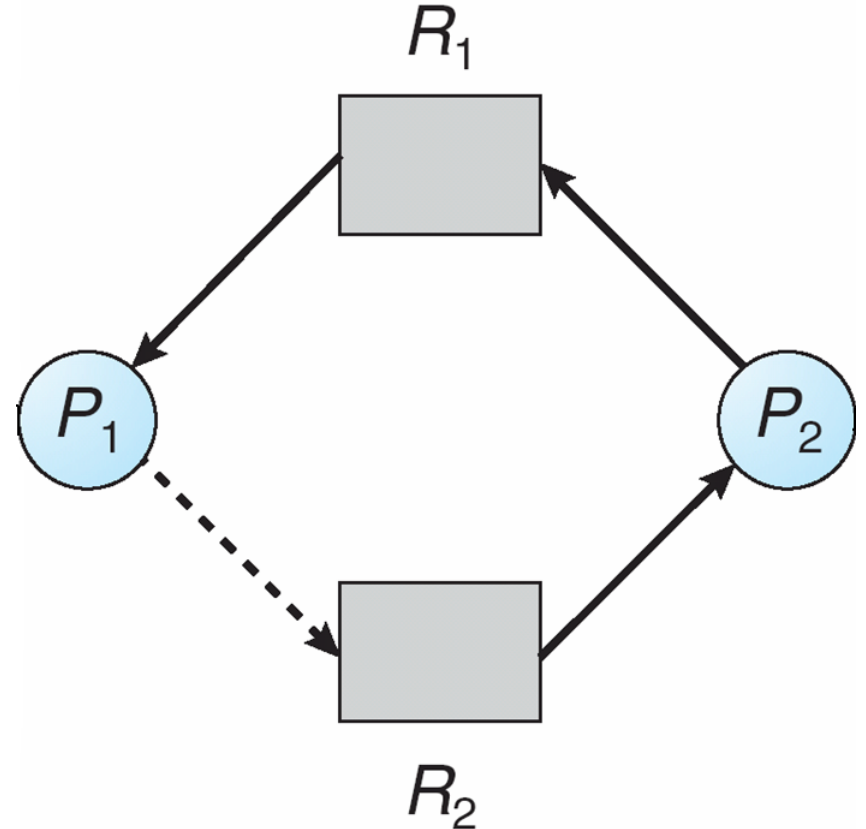
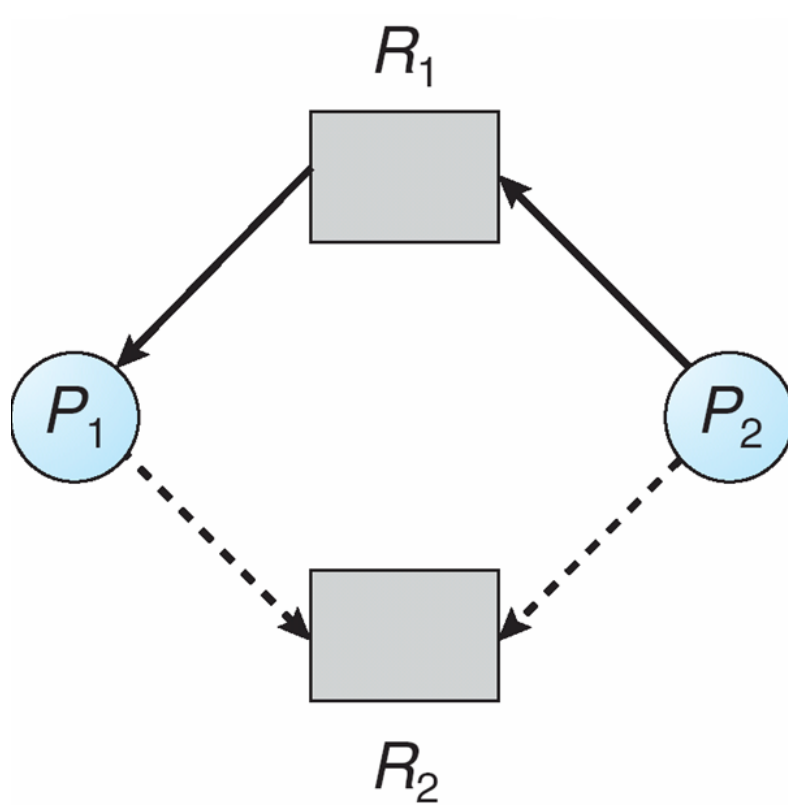
- Use the **banker's** algorithm



Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that **process P_j 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**
- **Resources must be claimed** in the system





Unsafe State



Resource-Allocation Graph 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**



Banker's Algorithm

- ❑ **Multiple instances**
- ❑ 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 a **finite** amount of **time**



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



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



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** _{i} ≤ **Work**

If no such i exists, go to step 4

3. **Work** = **Work** + **Allocation** _{i}
Finish [i] = **true**
go to step 2

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



Chapter 7

Example of Banker's algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
2		Allocation				Max				Available				Need						
3	Process	A	B	C		A	B	C		A	B	C		A	B	C		Need = Max(A4)-Allocation(A4)		
4	P0	0	1	0		7	5	3						7	4	3				
5	P1	2	0	0		3	2	2						1	2	2				
6	P2	3	0	2		9	0	2						6	0	0				
7	P3	2	1	1		2	2	2						0	1	1				
8	P4	0	0	2		4	3	3						4	3	1				
9																				



Chapter 7

Example of Banker's algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
2		Allocation					Max					Available				Need				
3	Process	A	B	C		A	B	C		A	B	C		A	B	C		Need = Max(A4)-Allocation(A4)		
4	P0	0	1	0		7	5	3						7	4	3		7	7	0
5	P1	2	0	0		3	2	2						1	2	2		1	3	2
6	P2	3	0	2		9	0	2						6	0	0		6	9	3
7	P3	2	1	1		2	2	2						0	1	1		0	2	2
8	P4	0	0	2		4	3	3						4	3	1		4	4	0
9																				
10	Total Allocation	7	2	5		= 0+0+2+1+2														

13	instances	A	B	C																
14	Available	10	5	7		in the beginning														
15	Total allocation	7	2	5																
16																				
17	Available	3	3	2		Available(A) - Max (A)														
18						Available(B) - Max (B)														
19						Available(C) - Max (C)														



Chapter 7

Example of Banker's algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
2		Allocation				Max				Available				Need						
3	Process	A	B	C		A	B	C		A	B	C		A	B	C		Need = Max(A4)-Allocation(A4)		
4	P0	0	1	0		7	5	3		3	3	2		7	4	3		7	7	0
5	P1	2	0	0		3	2	2						1	2	2		1	3	2
6	P2	3	0	2		9	0	2						6	0	0		6	9	3
7	P3	2	1	1		2	2	2						0	1	1		0	2	2
8	P4	0	0	2		4	3	3						4	3	1		4	4	0
9																				
10	Total Allocation	7	2	5		= 0+0+2+1+2														
11																				
12	P0																			
13	instances	A	B	C																
14	Available	10	5	7		in the beginning														
15	Total allocation	7	2	5																
16																				
17	Available	3	3	2		Available(A) - Max (A)														
18						Available(B) - Max (B)														
19						Available(C) - Max (C)														

If **Need** <= **Available** -> **Available** = **Available** + **Allocation**



Chapter 7

Example of Banker's algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
2		Allocation				Max				Available				Need						
3	Process	A	B	C		A	B	C		A	B	C		A	B	C				
4	P0	0	1	0		7	5	3		3	3	2		7	4	3				
5	P1	2	0	0		3	2	2						1	2	2				
6	P2	3	0	2		9	0	2						6	0	0				
7	P3	2	1	1		2	2	2						0	1	1				
8	P4	0	0	2		4	3	3						4	3	1				
9																				
10		Need			<=	Available			-->	Available			=	Available			+	Allocation		
11	P0	7	4	3		3	3	2		X										



Example of Banker's algorithm

< P1, >

Example of Banker's algorithm

< P1, P3, >

Chapter

Example of Banker's algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1																					
2		Allocation				Max					Available				Need						
3	Process	A	B	C		A	B	C			A	B	C		A	B	C				
4	P0	0	1	0		7	5	3			5	3	2		7	4	3				
5	P1	2	0	0		3	2	2			7	4	3		1	2	2				
6	P2	3	0	2		9	0	2			7	4	5		6	0	0				
7	P3	2	1	1		2	2	2			7	5	5		0	1	1				
8	P4	0	0	2		4	3	3							4	3	1				
9																					
10		Need			<=	Available			-->		Available			=	Available			+	Allocation		
11	P0	7	4	3		5	3	2		✗											
12	P1	1	2	2		3	3	2		✓	5	3	2	=	3	3	2	+	2	0	0
13	P2	6	0	0		5	3	2		✗											
14	P3	0	1	1		5	3	2		✓	7	4	3		5	3	2	+	2	1	1
15	P4	4	3	1		7	4	3		✓	7	4	5		7	4	3	+	0	0	2
16	P0	7	4	3		7	4	5			7	5	5		7	4	5	+	0	1	0
17																					
18																					
19																					
20						< P1, P3, P4, P0 >															



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1																					
2		Allocation				Max					Available				Need						
3	Process	A	B	C		A	B	C			A	B	C		A	B	C				
4	P0	0	1	0		7	5	3			5	3	2		7	4	3				
5	P1	2	0	0		3	2	2			7	4	3		1	2	2				
6	P2	3	0	2		9	0	2			7	4	5		6	0	0				
7	P3	2	1	1		2	2	2			7	5	5		0	1	1				
8	P4	0	0	2		4	3	3			10	5	7		4	3	1				
9																					
10		Need			<=	Available			-->		Available			=	Available			+	Allocation		
11	P0	7	4	3		5	3	2	✗												
12	P1	1	2	2		3	3	2	✓		5	3	2	=	3	3	2	+	2	0	0
13	P2	6	0	0		5	3	2	✗												
14	P3	0	1	1		5	3	2	✓		7	4	3		5	3	2	+	2	1	1
15	P4	4	3	1		7	4	3	✓		7	4	5		7	4	3	+	0	0	2
16	P0	7	4	3		7	4	5	✓		7	5	5		7	4	5	+	0	1	0
17	P2	6	0	0		7	5	5	✓		10	5	7		7	5	5	+	3	0	2
18																					
19																					
20		< P1, P3, P4, P0, P2 >																			



Methods for Handling Deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock recovery

Ignoring Deadlock



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

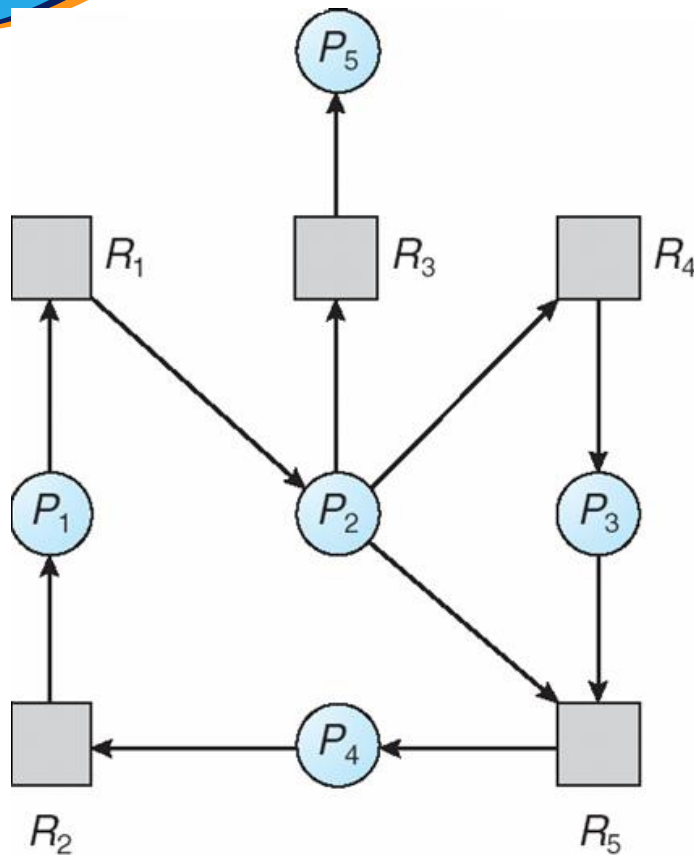


- Maintain **wait-for graph**
 - **Nodes** are processes
 - $P_i \rightarrow P_j$ if P_i is **waiting** for P_j
- Periodically invoke an algorithm that **searches** for a **cycle in the graph**. If there is a cycle, there exists a deadlock (**Resource-Allocation Graph**)

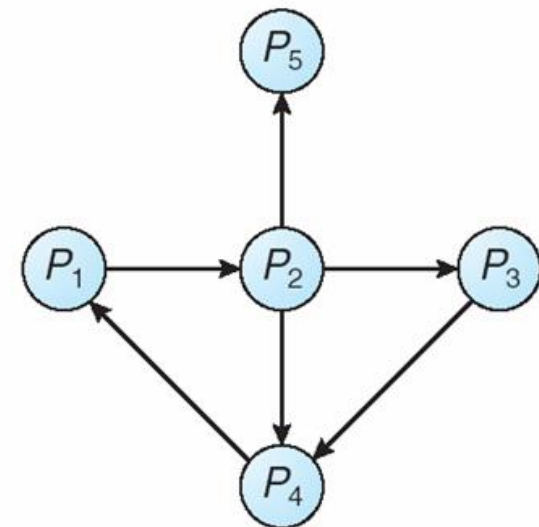


Chapter 7

Resource-Allocation Graph and Wait-for Graph



(a)



(b)



- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $Request[i_j] = k$, then process P_i is requesting k more instances of resource type R_j .



Detection Algorithm

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

- (a) *Work* = *Available*

- (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] = \text{false}$; otherwise, $Finish[i] = \text{true}$

2. Find an index i such that both:

- (a) $Finish[i] == \text{false}$

- (b) $Request_i \leq Work$

requires an order of
 $O(m \times n^2)$ operations

If no such i exists, go to step 4

3. $Work = Work + Allocation_i$
 $Finish[i] = \text{true}$
go to step 2

4. If $Finish[i] == \text{false}$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == \text{false}$, then P_i is **deadlocked**



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, there **may be many cycles** in the **resource graph** and so we would not be able to tell which of the **many deadlocked processes** “caused” the deadlock



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



- ❑ Selecting a **victim** – minimize cost
 - ❑ **Rollback** – return to **some safe state**, restart process for that state
 - ❑ Requires rollback and restarting capabilities
 - ❑ No **guarantee** that the original deadlock **will not occur** again
 - ❑ **Starvation** – **same process** may always **be picked** as **victim**, include number of rollback in cost factor



Methods for Handling Deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock recovery

Ignoring Deadlock



Ignore Deadlocks

- ❑ **Ignore** the problem altogether and **pretend** that **deadlocks never occur** in the system.
- ❑ The **user** has to terminate the process **manually**.
- ❑ The **last solution** is the one **used** by most **operating systems**, including **Linux** and **Windows**.



End of Lecture 6

