

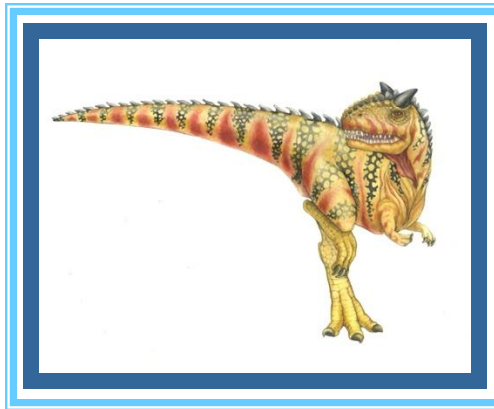
## Table of Contents

Slide 1: Chapter 8: Deadlocks .....	3
Slide 2: Chapter 8: Deadlocks .....	4
Slide 3: Chapter Objectives .....	5
Slide 4: Deadlock in Multithreaded Application .....	6
Slide 5: Deadlock Example with Lock Ordering .....	7
Slide 6: System Model .....	8
Slide 7: Deadlock Characterization .....	9
Slide 8: Resource-Allocation Graph .....	10
Slide 9: Resource-Allocation Graph (Cont.) .....	11
Slide 10: Resource Allocation Graph Example .....	12
Slide 11: Resource Allocation Graph With A Deadlock .....	13
Slide 12: Graph With A Cycle But No Deadlock .....	14
Slide 13: Basic Facts .....	15
Slide 14: Methods for Handling Deadlocks .....	16
Slide 15: Deadlock Prevention .....	17
Slide 16: Deadlock Prevention (Cont.) .....	18
Slide 17: Circular Wait .....	19
Slide 18: Deadlock Avoidance .....	20
Slide 19: Safe State .....	21
Slide 20: Basic Facts .....	22
Slide 21: Avoidance algorithms .....	23
Slide 22: Resource-Allocation Graph Scheme .....	24
Slide 23: Resource-Allocation Graph Algorithm .....	25
Slide 24: Banker ' s Algorithm .....	26
Slide 25: Data Structures for the Banker ' s Algorithm .....	27
Slide 26: Safety Algorithm .....	28

Slide 27: Resource-Request Algorithm for Process $P_i$ .....	29
Slide 28: Example of Banker ' s Algorithm .....	30
Slide 29: Example (Cont.) .....	31
Slide 30: Example (Cont.) .....	32
Slide 31: Example: $P_1$ Request $(1,0,2)$ .....	33
Slide 32: Deadlock Detection .....	34
Slide 33: Single Instance of Each Resource Type .....	35
Slide 34: Resource-Allocation Graph and Wait-for Graph .....	36
Slide 35: Several Instances for a Resource Type .....	37
Slide 36: Detection Algorithm .....	38
Slide 37: Example of Detection Algorithm .....	39
Slide 38: Example (Cont.) .....	40
Slide 39: Detection-Algorithm Usage .....	41
Slide 40: Recovery from Deadlock: Process Termination .....	42
Slide 41: Recovery from Deadlock: Resource Preemption .....	43
Slide 42: End of Chapter 8 .....	44

# Chapter 8: Deadlocks

---





# Chapter 8: Deadlocks

---

- ❑ Deadlock Examples
- ❑ Deadlock Characterization
- ❑ Resource Allocation Graph
- ❑ Methods for Handling Deadlocks
  - ❑ Deadlock Prevention
  - ❑ Deadlock Avoidance
  - ❑ Deadlock Detection and Recovery from Deadlock





# Chapter Objectives

---

- ❑ Illustrate how deadlock can occur when mutex locks are used.
- ❑ 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 banker's algorithm for deadlock avoidance.
- ❑ Apply the deadlock detection algorithm.
- ❑ Evaluate approaches for recovering from deadlock.





# Deadlock in Multithreaded Application

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

- The order in which the threads run depends on how they are scheduled by the CPU scheduler
- This example illustrates the fact that it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.





# Deadlock Example with Lock Ordering

```
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);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers \$25 from account A to account B, and Transaction 2 transfers \$50 from account B to account A





# System Model

---

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







# Deadlock Characterization

Deadlock involving multiple processes can arise if the following **four** conditions hold **simultaneously** – they are **necessary** but **not sufficient** 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 resource(s) 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  $P_0$ .





# 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  $P_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow P_i$





# Resource-Allocation Graph (Cont.)

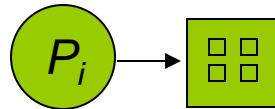
- Process



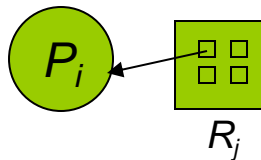
- Resource Type with 4 instances



- $P_i$  requests instance of  $R_j$



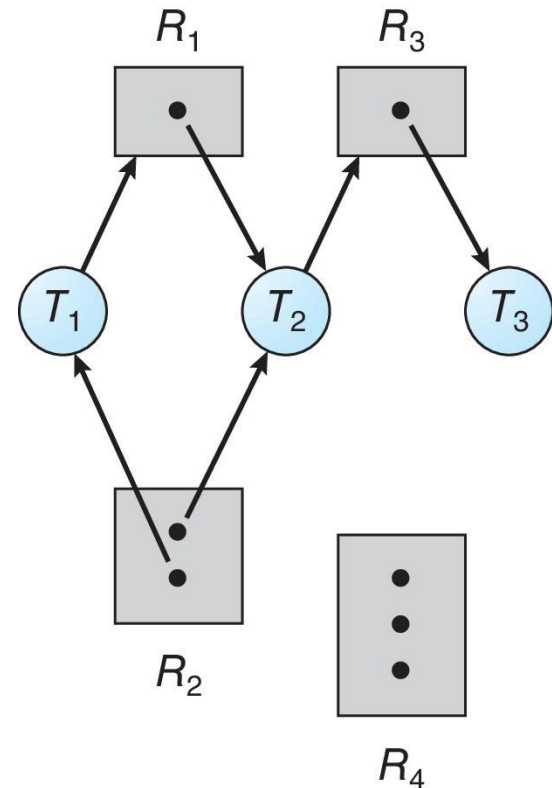
- $P_i$  is holding an instance of  $R_j$





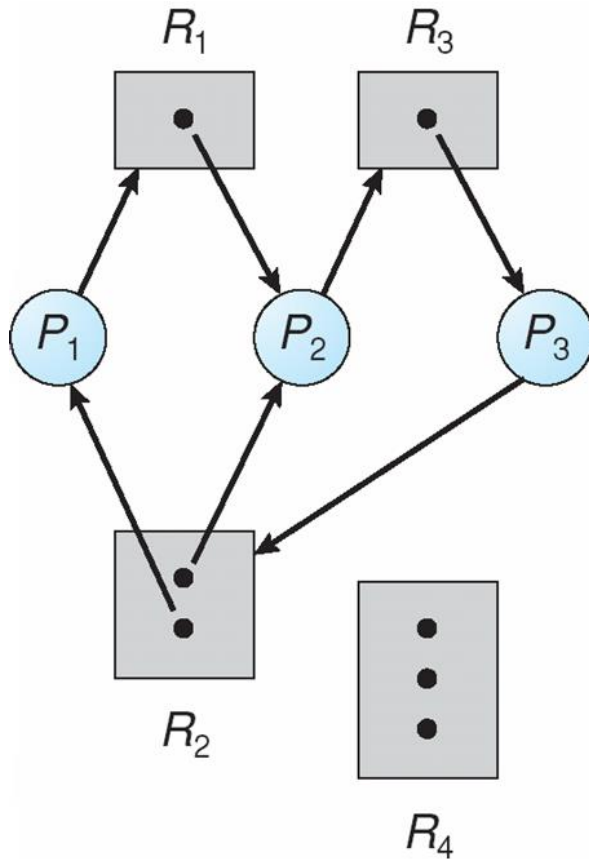
# Resource Allocation Graph Example

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3





# Resource Allocation Graph With A Deadlock



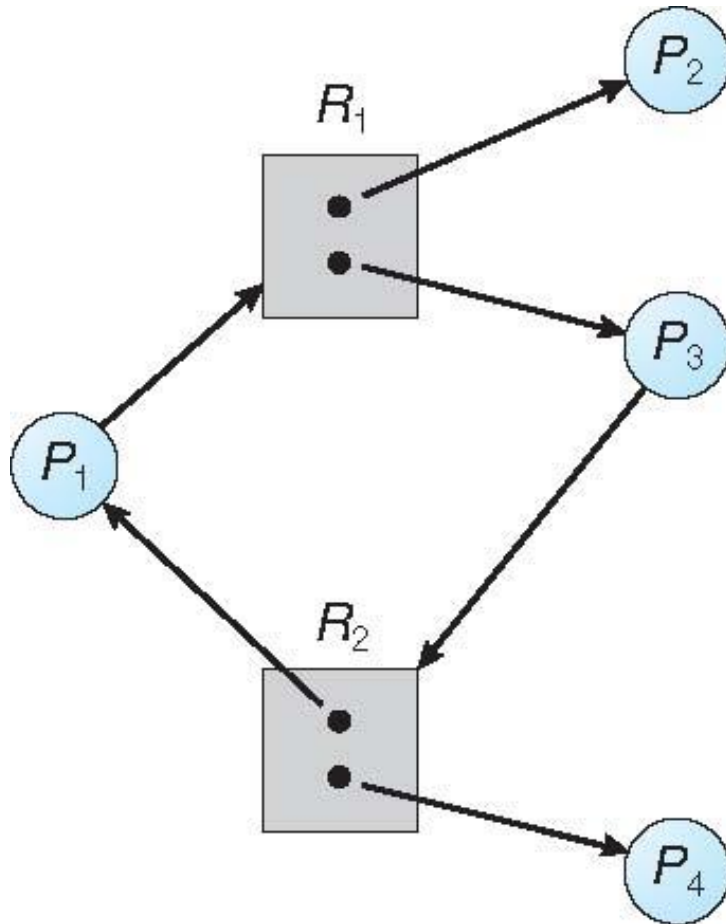
Cycles exist

- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes  $P_1$ ,  $P_2$ , and  $P_3$  are deadlocked





# Graph With A Cycle But No Deadlock



A cycle exists

□  $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

□ However, there is no deadlock. Observe that thread  $P_4$  may release its instance of resource type  $R_2$ . That resource can then be allocated to  $P_3$ , breaking the cycle.





# Basic Facts

---

- If a graph contains **no cycles**  $\Rightarrow$  **no deadlock**
- If a graph contains a cycle  $\Rightarrow$  the system may or may not be in a deadlocked state
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock





# Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - **Deadlock prevention**: it provides a set of methods to ensure at least one of the necessary conditions cannot hold
  - **Deadlock avoidance**: this requires **additional information** given in advance concerning which resources a process will request and use during its lifetime. Within such knowledge, the OS can decide for each resource request whether a process should wait or not
- **Deadlock detection** - allow the system to enter a deadlock state, periodically detect if there is a deadlock and then recover from it
- Many commercial operating systems, esp., for desktops, laptops, and smart phones ignore the deadlock problem because of the overhead and pretend that deadlocks never occur in the system
  - It will cause the system's performance to deteriorate, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state – restart the system manually







# Deadlock Prevention

Restrain the ways request can be made

- ❑ **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); but it must hold for non-sharable resources
- ❑ **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - ❑ Require each process to request and be allocated all its resources before it begins execution, or request resources only when the process has none
  - ❑ The **disadvantages** - low resource utilization, and possible starvation
- ❑ **No Preemption** –
  - ❑ If a process that is holding some resources requests another that cannot be immediately allocated to it, then all resources currently being held are released
  - ❑ Preempted resources added to list of resources for which the process is waiting
  - ❑ Process will be restarted only when it can regain all of its old resources, as well as the new ones that it is currently requesting
  - ❑ This can only be applied to resources whose state can be easily saved and restored such as registers, memory space and database transactions. It cannot generally be applied to resources such as locks and semaphores





# Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration –  $R = \{<R_1, R_2, \dots, R_m>\}$ 
  - This requires that a process cannot request a resource  $R_j$  before requesting a resource  $R_i$  if  $j > i$
- This can be proved by contradiction
  - Let the set of processes involved in a circular wait be  $P = \{<P_0, P_1, \dots, P_n>\}$ , where  $P_i$  is waiting for a resource  $R_i$ , which is held by process  $P_{i+1}$ , so that  $P_n$  is waiting for a resource  $R_n$  held by  $P_0$ .
  - Since process  $P_{i+1}$  is holding resource  $R_i$  while requesting resource  $R_{i+1}$ , we must have  $R_i < R_{i+1}$  for all  $i$ .
  - This implies  $R_0 < R_1 < R_2 \dots < R_n < R_0$
  - $R_0 < R_0$ , this is impossible, therefore there can be no circular wait





# Circular Wait

- ❑ Invalidating the circular wait condition is most common.
- ❑ Simply assign each resource (i.e. mutex locks) a unique number.
- ❑ Resources must be acquired **in order**.
- ❑ If:

```
first_mutex = 1  
second_mutex = 5
```

code for `thread_two` could not be written as follows:

```
/* thread_one runs in this function */  
void *do_work_one(void *param)  
{  
    pthread_mutex_lock(&first_mutex);  
    pthread_mutex_lock(&second_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}  
  
/* thread_two runs in this function */  
void *do_work_two(void *param)  
{  
    pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_exit(0);  
}
```





# Deadlock Avoidance

---

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

For instance, with the knowledge of complete sequence of request and release for each process, system can decide for each request whether the process should wait to avoid a possible future deadlock.

- The simplest and most useful model requires that 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 to ensure that a circular-wait condition can never exist
- **Resource-allocation state** is defined by the number of (1) available and (2) allocated resources, and (3) the maximum demands of the processes





# Safe State

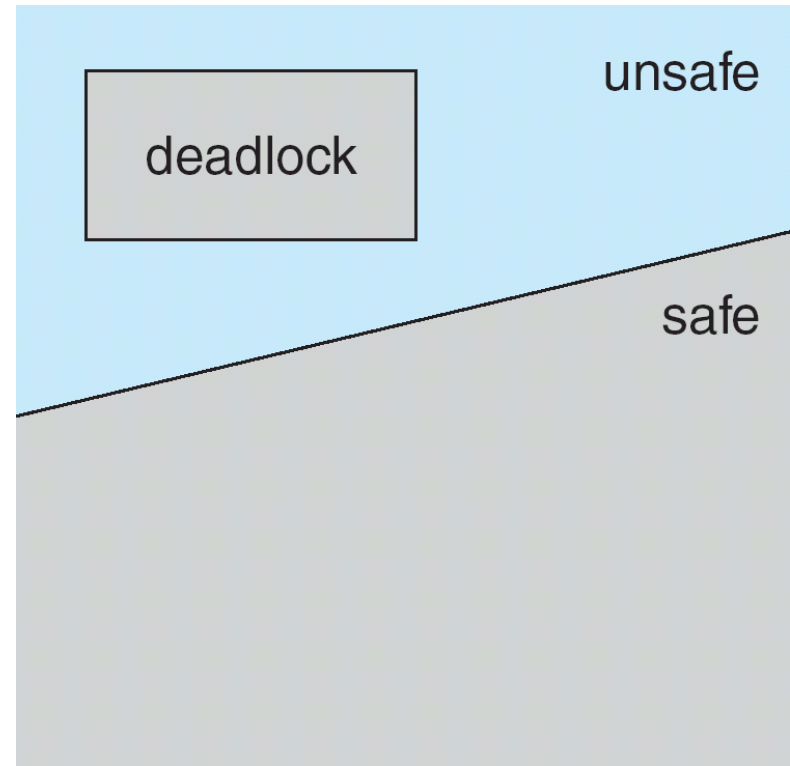
- When a process requests an available resource, system must decide whether such an allocation will leave the system in a **safe state**
- System is in **safe state** if there exists a **safe sequence**  $\langle P_1, P_2, \dots, P_n \rangle$  consisting of **all processes** in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request (based on prior declaration) can be satisfied by currently available resources plus resources held by **all**  $P_j$ , 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
- If no such sequence exists, then the system state is said to be **unsafe**.





# 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**
  - In this scheme, if a process requests a resource that is currently available, it may still have to wait (if the allocation leads to unsafe state).
  - The resource utilization may be lower than it would be otherwise





# Avoidance algorithms

---

- 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**  $P_i \rightarrow R_j$  indicates 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 converts to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to claim edge
- Resources must be claimed a priori in the system

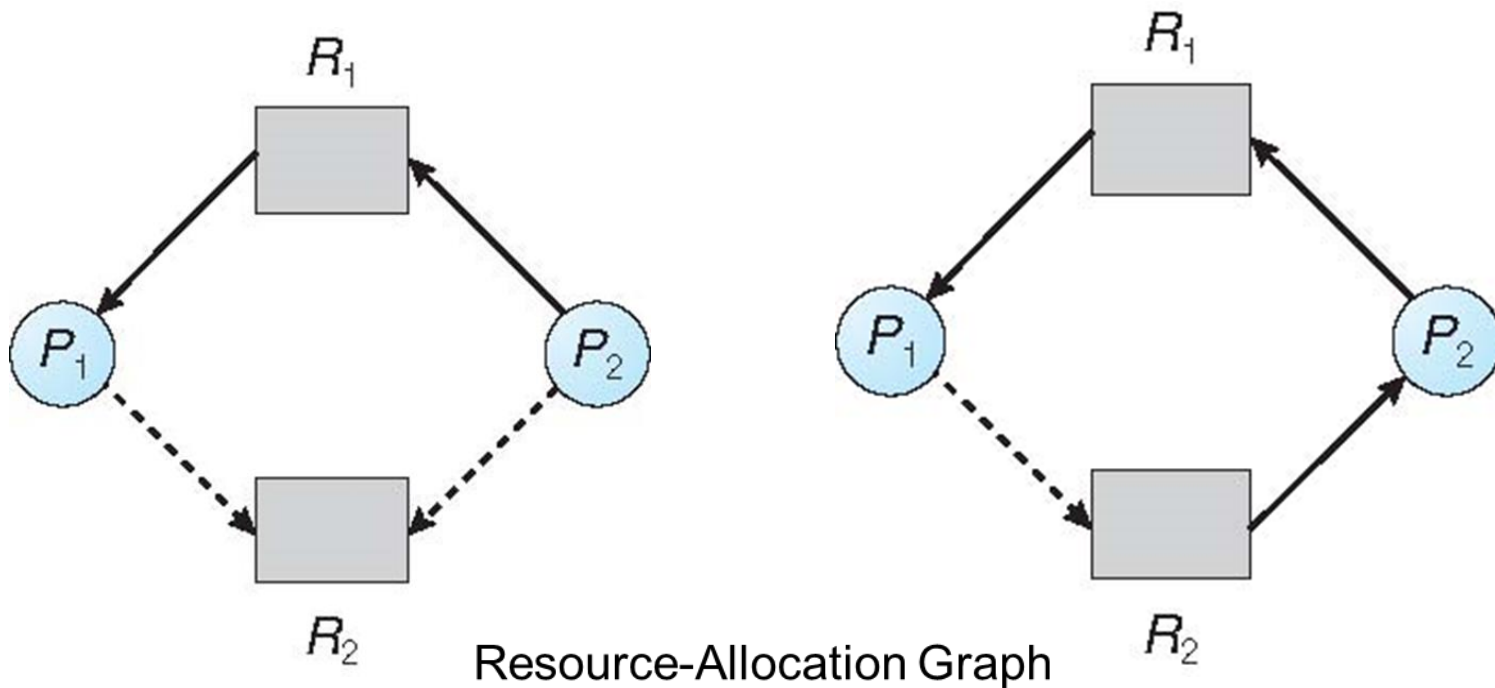






# Resource-Allocation Graph Algorithm

- 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 declare a *priori maximum usage*
- ❑ When a process requests a resource, it may have to wait – check to see if this allocation results in a safe state or not
- ❑ When a process gets all its resources it must return them in a finite amount of time after use
- ❑ This is analogous to banking loan system, which has a maximum amount, total, that can be loaned at one time to a set of businesses each with a credit line.





# Data Structures for the 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

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





# 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)  **$Need_i \leq Work$**

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, otherwise unsafe





# Resource-Request Algorithm for Process $P_i$

**Request** = request vector for process  $P_i$ . If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

1. If  $\text{Request}_i \leq \text{Need}_i$  go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3. Otherwise,  $P_i$  must wait, since resources are not available
3. Pretend to have allocated requested resources to  $P_i$  by modifying the state as follows:

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

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

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

- Run safety algorithm: If safe  $\Rightarrow$  the resources can be allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored





# Example of Banker's Algorithm

□ 5 processes  $P_0$  through  $P_4$ ;

3 resource types:

$A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)

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	





## Example (Cont.)

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

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

- The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies the safety criteria





## Example (Cont.)

- 5 processes  $P_0$  through  $P_4$ ; 3 resource types:  
A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time  $T_0$ :

	<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	7 4 3
$P_1$	2 0 0	3 2 2		1 2 2
$P_2$	3 0 2	9 0 2		6 0 0
$P_3$	2 1 1	2 2 2		0 1 1
$P_4$	0 0 2	4 3 3		4 3 1

The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies the safety criteria







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

- Check that Request  $\leq$  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? – resource not available
- Can request for (0,2,0) by  $P_0$  be granted? – state is not safe





# Deadlock Detection

---

If a system does not use either a deadlock-prevention, or deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide

- An algorithm that examines the state of the system to determine whether a deadlock can occur
- An algorithm to recover from the deadlock





# Single Instance of Each Resource Type

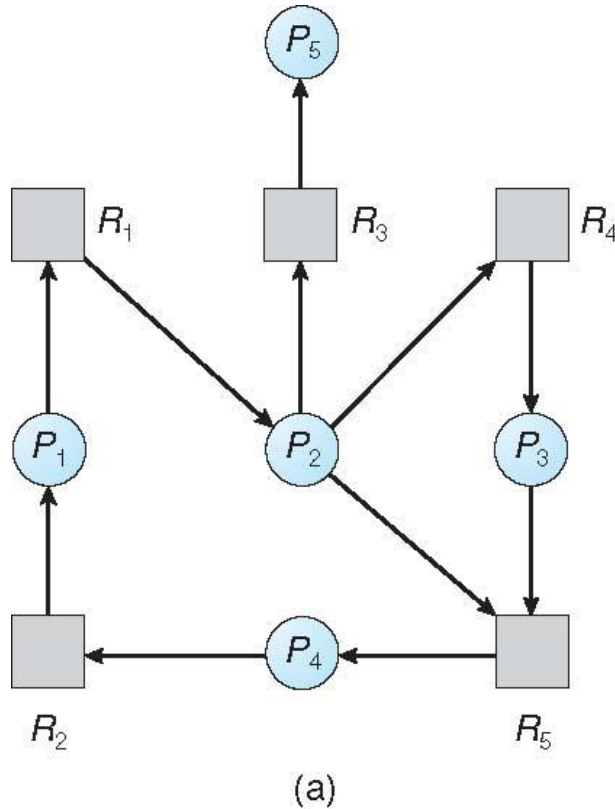
---

- 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
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph
- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances for each resource type

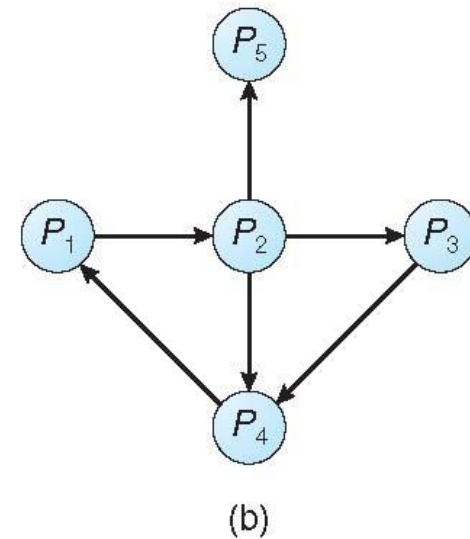




# Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding Wait-for Graph





# Several Instances for a Resource Type

---

- **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  $k$  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  $\text{Allocation}_i \neq 0$ , then **Finish[i] = false**; otherwise, **Finish[i] = true**
2. Find an index **i** such that both:
  - (a) **Finish[i] == false**
  - (b) **Request<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] == false**, for some **i**,  $1 \leq i \leq n$ , then the system is in deadlock state.  
Moreover, if **Finish[i] == false**, then **P<sub>i</sub>** is deadlocked

This algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked 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	

- 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>Request</u>		
	A	B	C
$P_0$	0	0	0
$P_1$	2	0	2
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	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  $P_4$







# Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by a deadlock when it occurs
    - ▶ one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph; we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
- Invoking the deadlock detection algorithm for every resource request will incur considerable overhead in computation.
  - A less expensive alternative is to invoke the algorithm at defined intervals – for example, once per hour, or whenever CPU utilization drops below 40%





# Recovery from Deadlock: Process Termination

---

- ❑ **Abort all deadlocked processes:** This clearly breaks the deadlock cycle, but at great expense
- ❑ **Abort one process at a time until the deadlock cycle is eliminated:** This also incurs considerable overhead, since after each process is aborted, the deadlock-detection algorithm needs to run
- ❑ **In which order should we choose to abort? – many factors:**
  1. Priority of the process
  2. How long process has computed, and how much longer to complete?
  3. Resources the process has used
  4. Resources the process needs to complete
  5. How many processes will need to be terminated?
  6. Is process interactive or batch?





# Recovery from Deadlock: Resource Preemption

---

To successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken

- ❑ **Selecting a victim** – minimize cost (which resources and which processes are to be preempted)
- ❑ **Rollback** – return to some safe state, restart process from that state
- ❑ **Starvation** – the same process may always be picked as victim, including the number of rollback in cost factor might help to reduce the starvation



# End of Chapter 8

---

