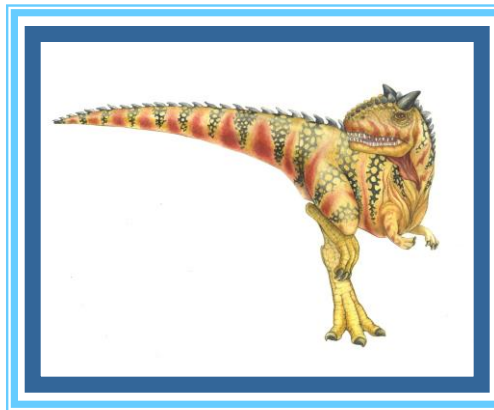


Chapter 7: Deadlocks





Topics to be Covered

Week 1	Introduction
Week 2	Operating System Structures
Week 3	Processes
Week 4	Threads
Week 5	Threads
Week 6	Synchronization
Week 7	Synchronization
Week 8	Midterm
Week 9	Classical Problems
Week 10	CPU Scheduling
Week 11	CPU Scheduling
Week 12	Deadlocks
Week 13	Memory Management
Week 14	Virtual Memory





Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
 - Deadlock **P**revention
 - Deadlock **A**voidance
 - Deadlock **D**etection
- 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





Deadlocks

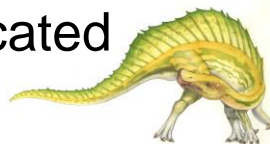
- A law passed by the Kansas legislature early in the 20th century
- It said, in part: «**When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone**»
- In general Operating Systems typically **do not provide deadlock-prevention facilities**. Deadlocks are more common with recent trends (MTP, too many resources etc.).
 - It remains the **responsibility of programmers** to ensure that they design **deadlock-free programs**





System Model

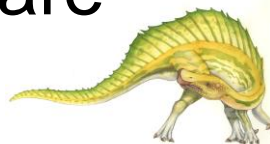
- System consists of resources
 - Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
 - Each resource type R_i has W_i instances.
 - Each process utilizes a resource as follows:
 - ▶ **request**
 - Numerical restrictions
 - ▶ **use**
 - ▶ **release**
- A process must request a resource before using it and must release the resource after using it.
- A system table records whether each resource is free or allocated





System Model

- **If the resources are not available** at that time, the process enters a **waiting state**.
- Sometimes, a **waiting process** is **never again able to change state**, because the resources it has requested are **held by other waiting processes**. This situation is called a **deadlock**.
- **Definition:** A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
- The locking tools presented in Chapter 5 are designed to avoid race conditions





Deadlock with Mutex Locks

- ❑ Deadlocks can occur via system calls, locking, etc.

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two`





```
/* 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 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
- ❑ **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 .



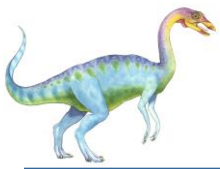


Resource-Allocation Graph (Directed)

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

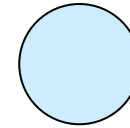
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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



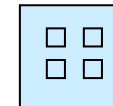


Resource-Allocation Graph (Cont.)

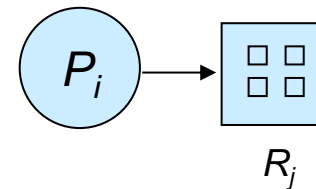
□ Process



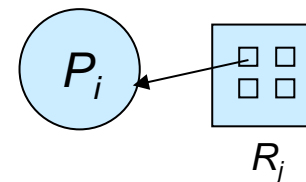
□ Resource Type with 4 instances



□ P_i requests instance of R_j

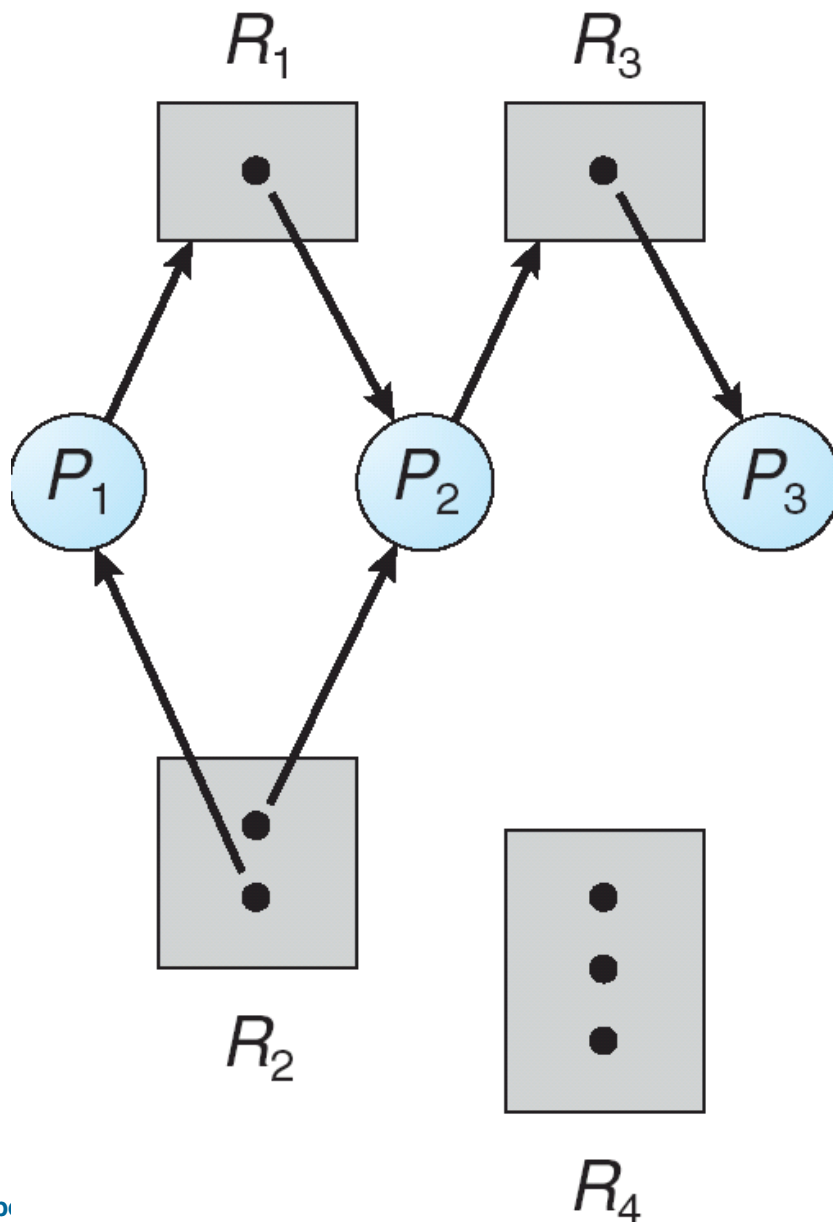


□ P_i is holding an instance of R_j





Example of a Resource Allocation Graph



The sets P , R , and E :

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

When this request can be fulfilled, the request edge is

instantaneously transformed to an assignment edge.

When the process no longer needs access to the resource, it releases the resource. As

a result, the **assignment edge is deleted.**





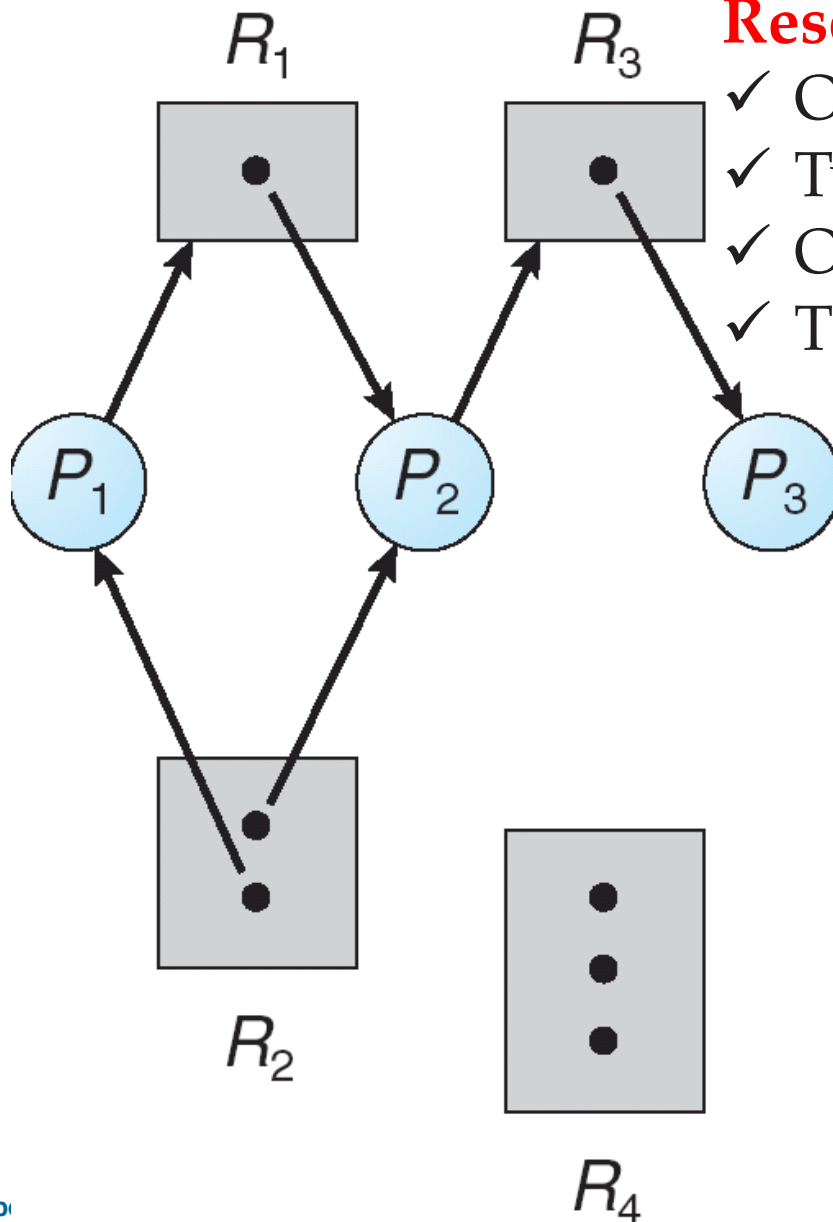
Example of a Resource Allocation Graph

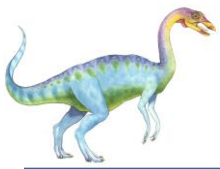
Resource instances:

- ✓ One instance of resource type R_1
- ✓ Two instances of resource type R_2
- ✓ One instance of resource type R_3
- ✓ Three instances of resource type R_4

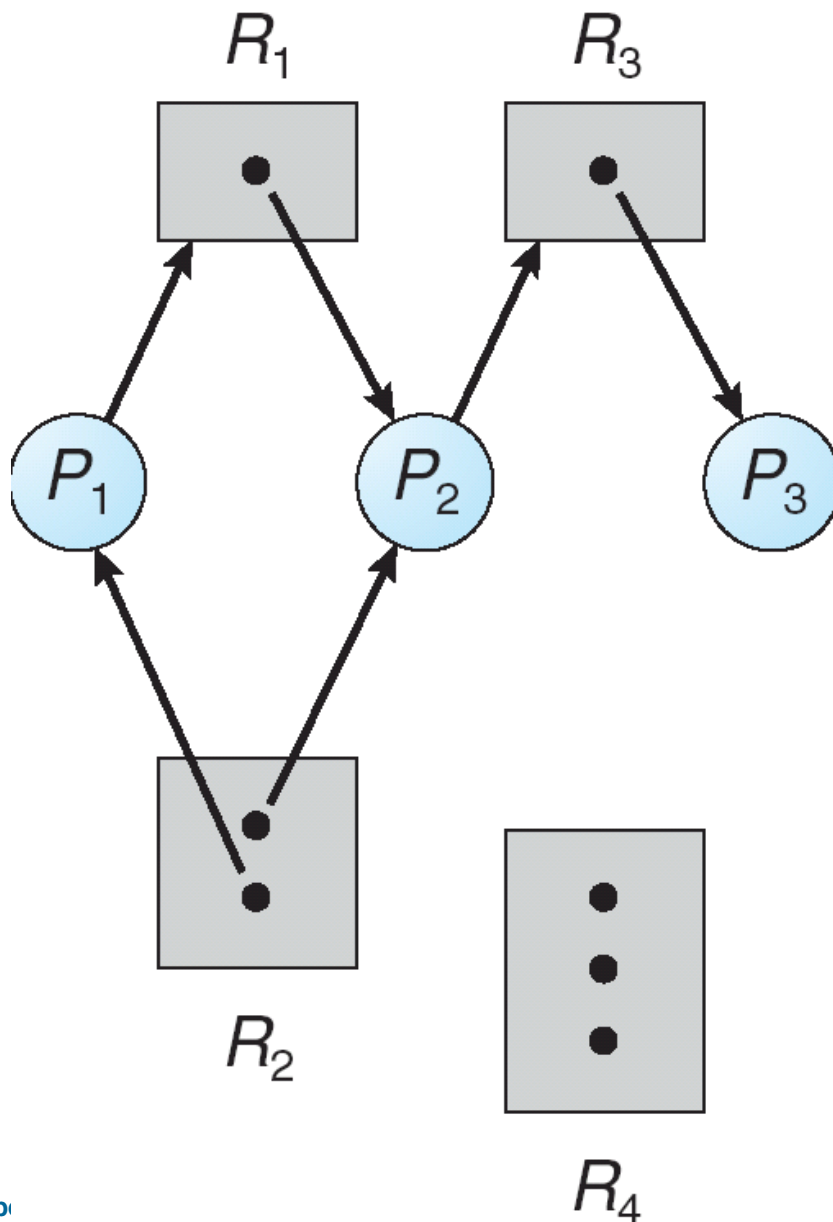
Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1 .
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3 .
- Process P_3 is holding an instance of R_3 .



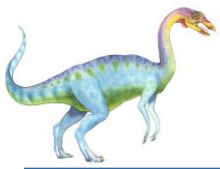


Example of a Resource Allocation Graph

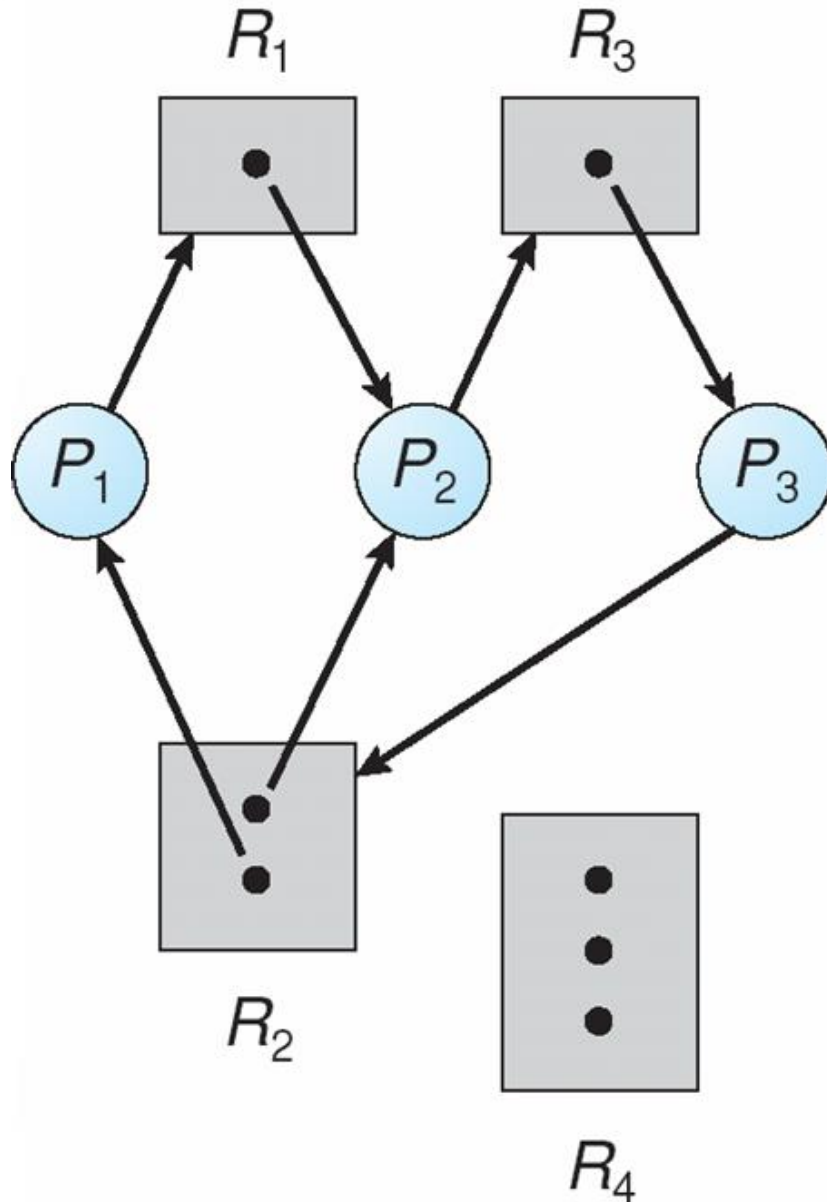


- If the graph contains **no cycles**, then no process in the system is **deadlocked**.
- If the graph does contain a cycle, then a **deadlock may exist**





Resource Allocation Graph With A **Deadlock**



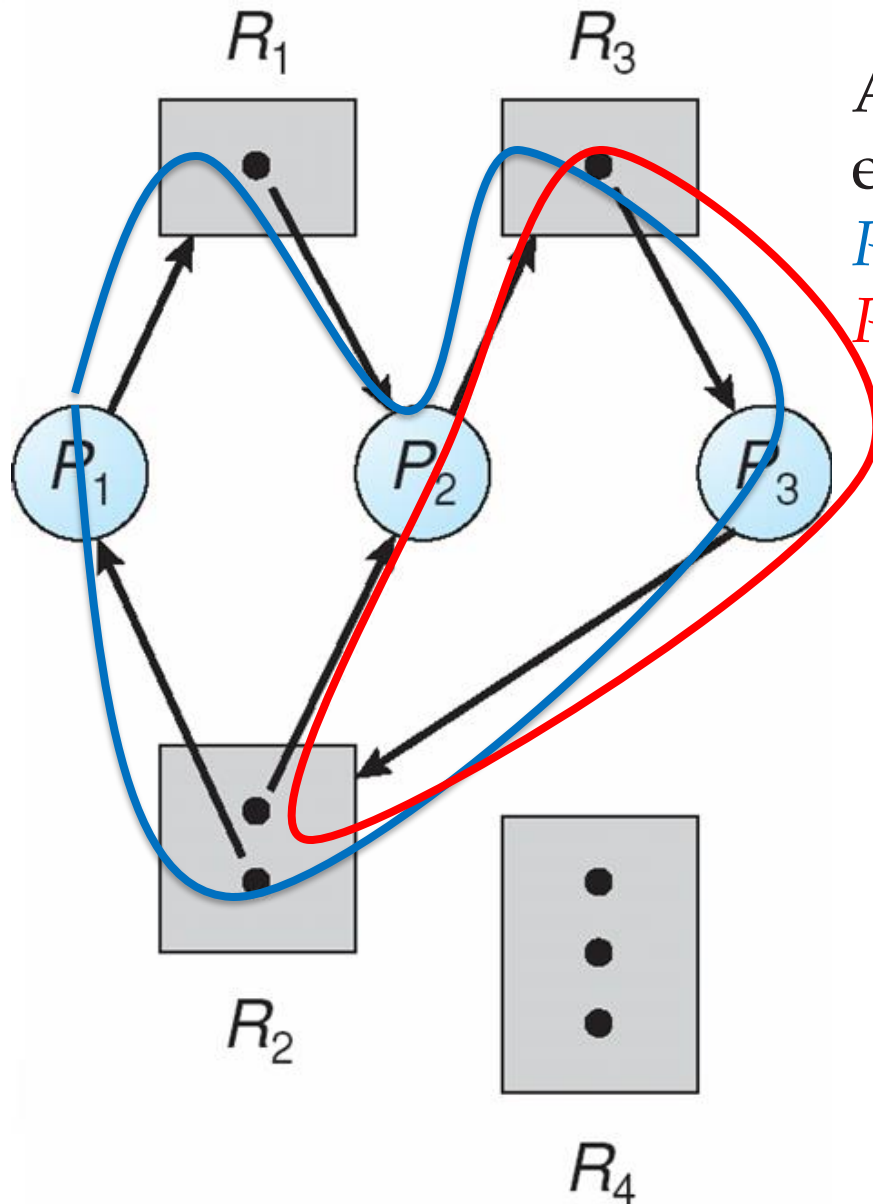
If each resource type has **exactly one instance**, then a cycle implies that a **deadlock has occurred**.

If each resource type has **several instances**, then a cycle **does not necessarily** imply that a deadlock has occurred





Resource Allocation Graph With A Deadlock



At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

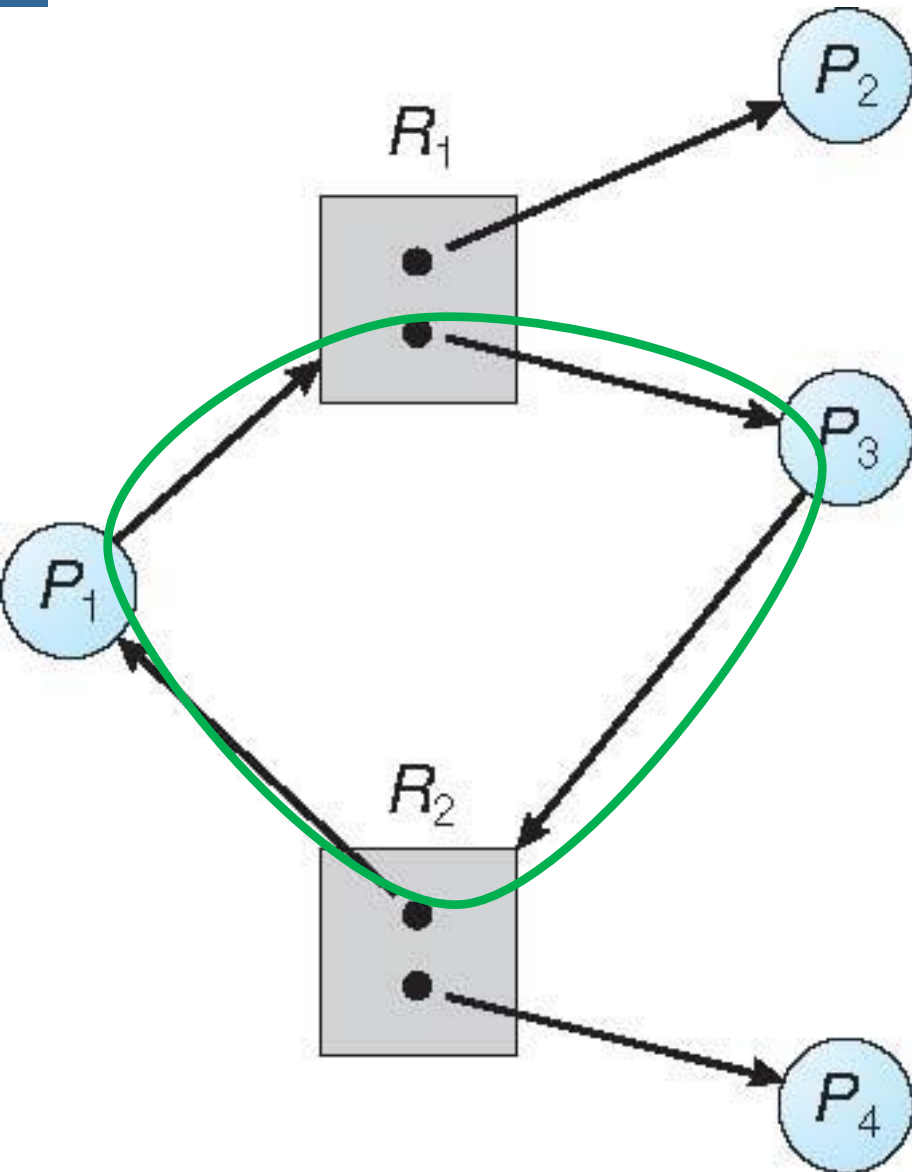
Each process **involved** in the cycle is **deadlocked**.

Processes P_1 , P_2 , and P_3 are **deadlocked**.





Graph With A Cycle But No Deadlock



Cycle:

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

P_4 may release its instance of resource type R_2

If the graph contains **no cycles**, then no process in the system is deadlocked.

If the graph does **contain a cycle**, then a **deadlock may exist**

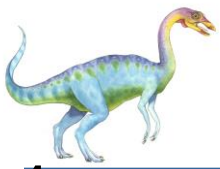




Basic Facts

- 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





Methods for Handling Deadlocks

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

- Deadlock **prevention**

- ▶ a protocol to prevent or avoid deadlocks, ensuring that the system will **never** enter a **deadlocked** state.

- Deadlock **avoidance**

- ▶ operating system be given additional information in advance concerning which resources a process will request and use during its lifetime

2- **Allow** the system to enter a deadlock state, **detect** and then **recover**

3- **Ignore** the problem and pretend that deadlocks never occur in the system; used by most operating systems, including Windows and UNIX





Handling Deadlocks

- **Either** prevent deadlocks using
 - Deadlock-prevention
 - Deadlock avoidance algorithm
- **Or** detect and recover deadlock
- **Otherwise** you are deadlocked
 - Restart your pc.





Deadlock Prevention

Restrain the ways request can be made.

At least one of the **Four** necessary conditions must **not** hold.

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources. **At least one resource must be nonsharable**
- **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, or allow process to request resource **before it begins execution**s only when the process has none allocated to it.
- **Low resource utilization; starvation possible**





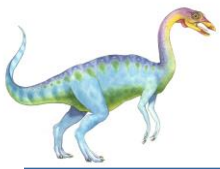
Deadlock Prevention (Cont.)

□ 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

□ Circular Wait – impose a total **ordering** of all resource types, and require that each process requests resources in an **increasing order** of enumeration



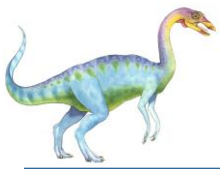


Deadlock Example

```
/* 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 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





Deadlock Avoidance

Requires that the system has some additional *a priori* 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**
- **Resource-allocation state** is defined by the number of available and allocated resources, and the maximum demands of the processes





Safe State

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a **safe sequence**
- If no such sequence exists, then the system state is said to be **unsafe**.
- **Not all unsafe states are deadlocks**





Safe State

- 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 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





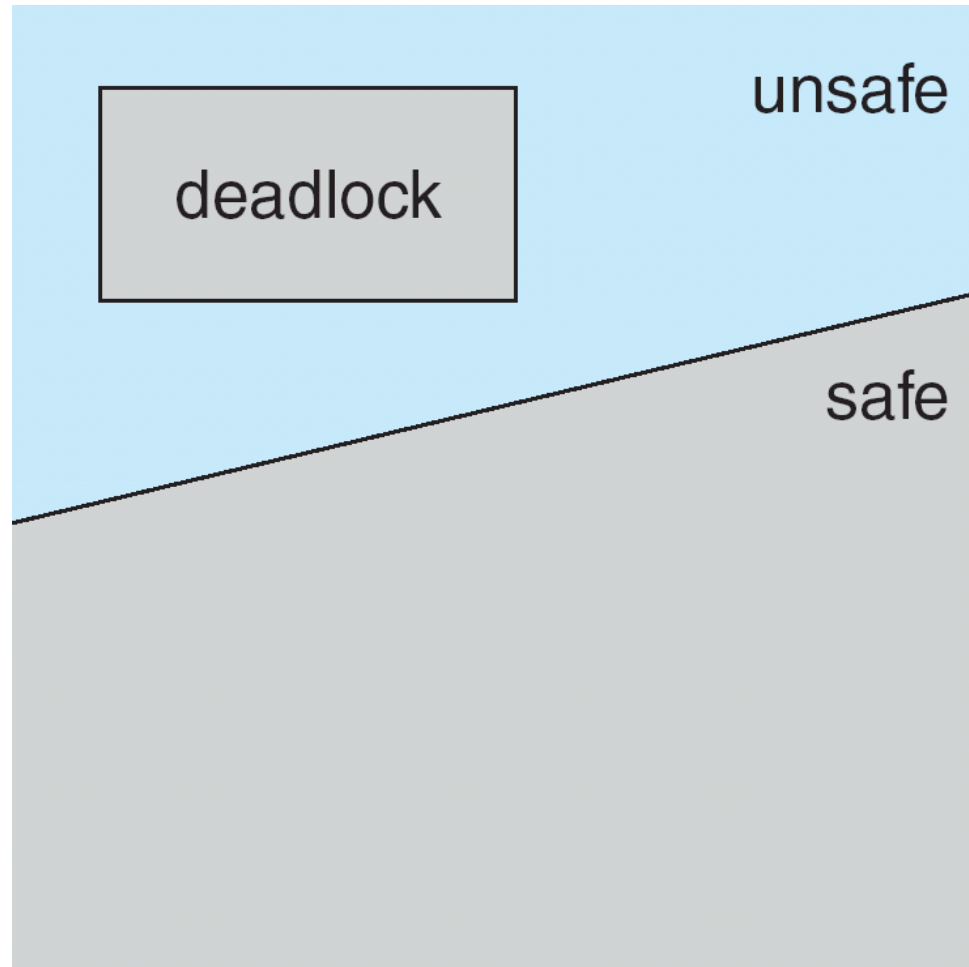
Basic Facts

- 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.





Safe, Unsafe, Deadlock State





Safe State

We consider a system with **twelve magnetic tape** drives and **three processes**: P_0 , P_1 , and P_2 .

Suppose that, at time $t=0$, process P_0 is holding **five tape** drives, process P_1 is holding **two tape** drives, and process P_2 is holding **two tape** drives. (Thus, there are three free tape drives.)

	<u>Maximum Needs</u>	<u>Current Needs</u>	
P_0	10	5	At time t_0 , the system is in a safe state The sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition
P_1	4	2	
P_2	9	2	

Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P_0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P_2 can get all its tape drives and return them (the system will then have all twelve tape drives available).





Unsafe State

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

A system can go from a safe state to an unsafe state. Suppose that, at time t_1 , process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have **only four available** tape drives

Since process P_0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P_2 may request six additional tape drives and have to wait, resulting in a deadlock.

In this scheme, if a process requests a resource that is currently available, it may still have to **wait**. Thus, **resource utilization may be lower** than it would otherwise be.



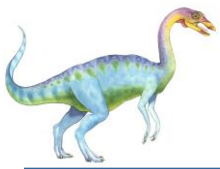


Avoidance Algorithms

- **Single instance** of a resource type
 - Use a **resource-allocation graph**

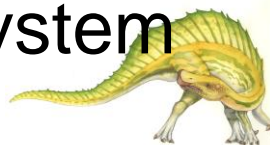
- **Multiple instances** of a resource type
 - Use the **banker's algorithm**





Resource-Allocation Graph Scheme

- **(Request + Assignment + Claim) Edge**
- **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 *a priori* in the system





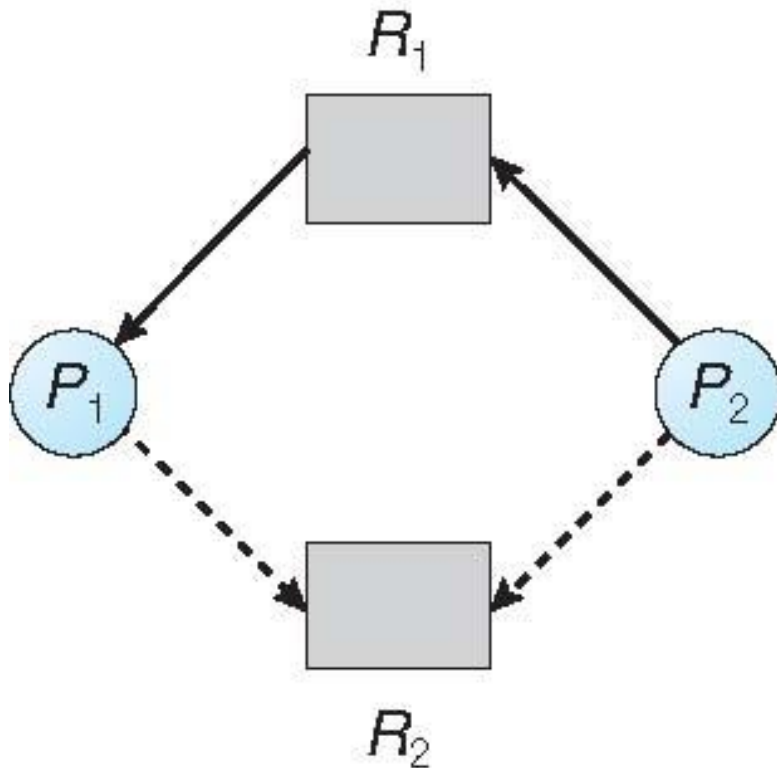
Resource-Allocation Graph Algorithm

- 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
- The resource-allocation-graph algorithm **is not applicable to** a resource allocation system with **multiple instances** of each resource type.





Resource-Allocation Graph



P_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle in the graph

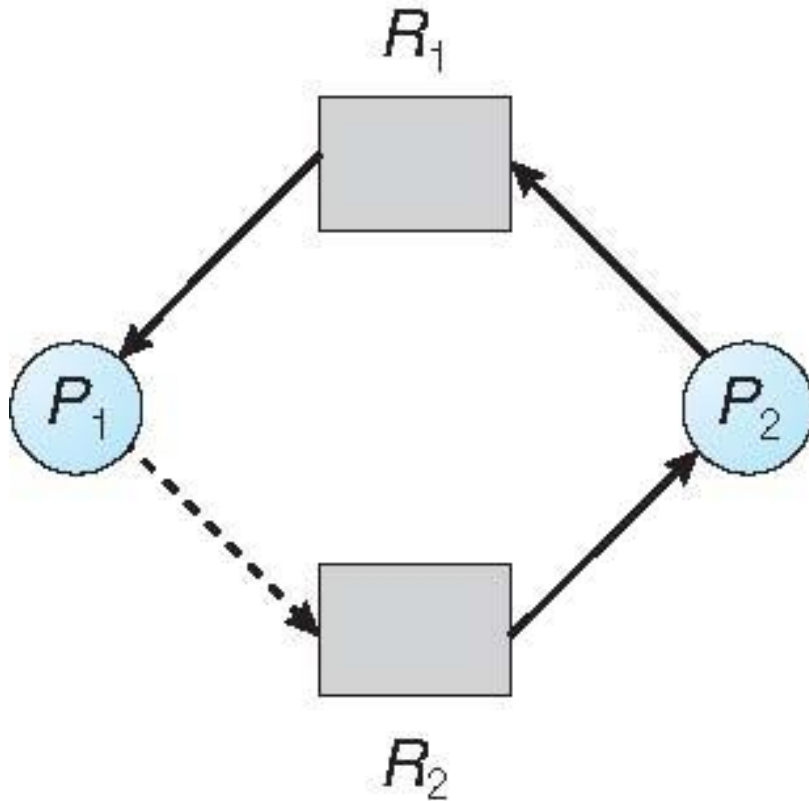
If P_1 requests R_2 , and P_2 requests R_1 , then a deadlock will occur.

Note that **request edge converted** to an **assignment edge** when the resource is allocated to the process





Unsafe State In Resource-Allocation Graph



A cycle, as mentioned, indicates that the system is in an **unsafe** state

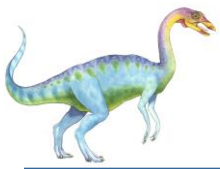




Banker's Algorithm

- Supports Multiple instances
- **Less efficient** than the **resource-allocation graph** scheme
- When a new process enters the system, it must declare the **maximum** number of instances of each resource type that it **may need**.
 - This number **may not exceed** the total number of resources in the system.
- 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**





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**

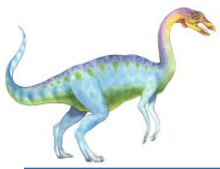
(b) **Need** $_i \leq$ **Work**

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





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. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If **$Request_i \leq Available$** , go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request_i;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i - Request_i;$

- 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





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 of Banker's Algorithm

□ The content of the matrix **Need** is defined to be

$$\text{Need} = \text{Max} - \text{Allocation}$$

<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	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**

□ $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies 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?
- Can request for (0,2,0) by P_0 be granted?

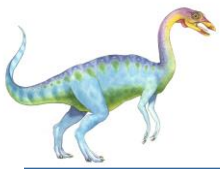




Deadlock Detection

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

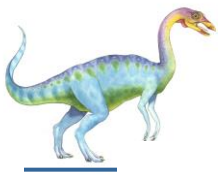




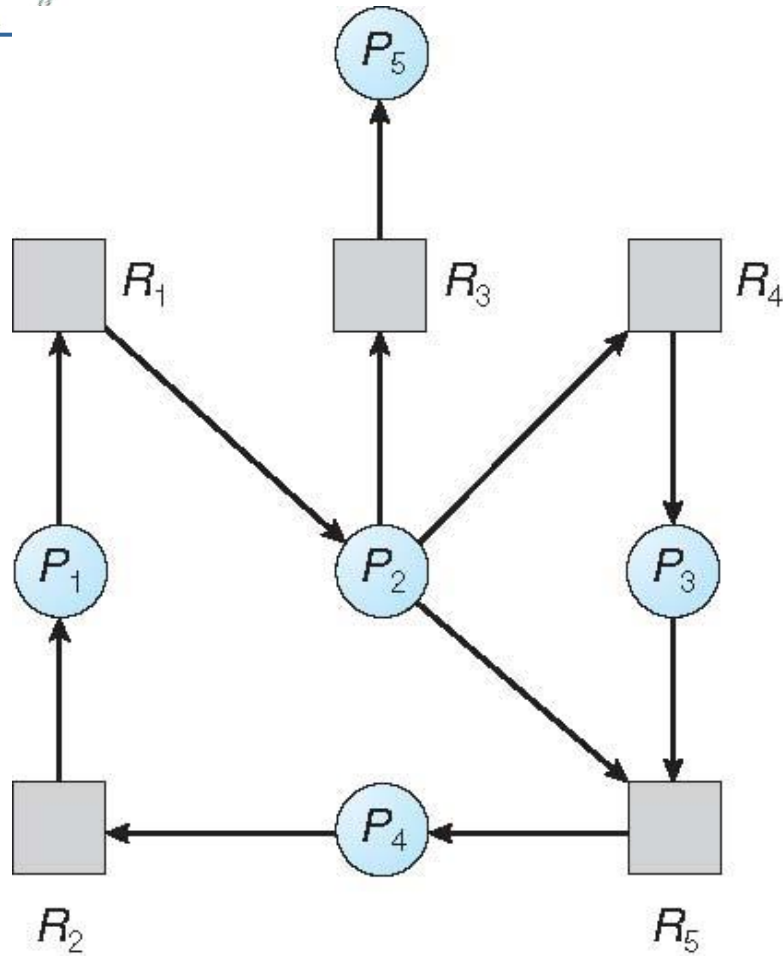
Single Instance of Each Resource Type

- 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





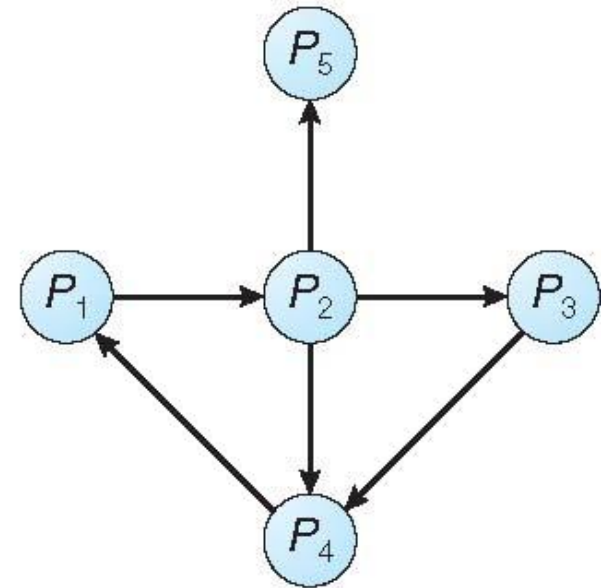
Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph

We obtain this graph from the **resource-allocation graph** by removing the resource nodes and collapsing the appropriate edges



(b)

Corresponding wait-for graph





Several Instances of 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 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] = false; otherwise, **Finish[i] = true**

2. Find an index **i** such that both:

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

(b) **Request_i \leq Work**

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





Detection Algorithm (Cont.)

3. **$Work = Work + Allocation_i$**
 $Finish[i] = true$
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_i is deadlocked

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





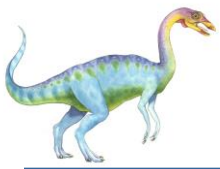
Example (Cont.)

- 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





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



End of Chapter 7

