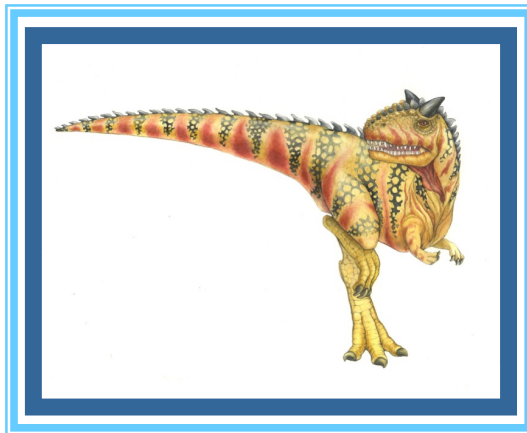


Chapter 8: Deadlocks





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





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**
 - **use**
 - **release**





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

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

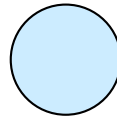
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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



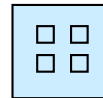


Resource-Allocation Graph (Cont.)

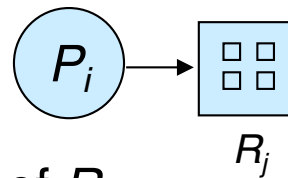
- Process



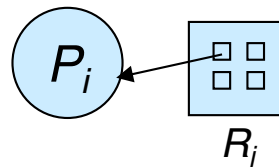
- Resource Type with 4 instances



- P_i requests instance of R_j

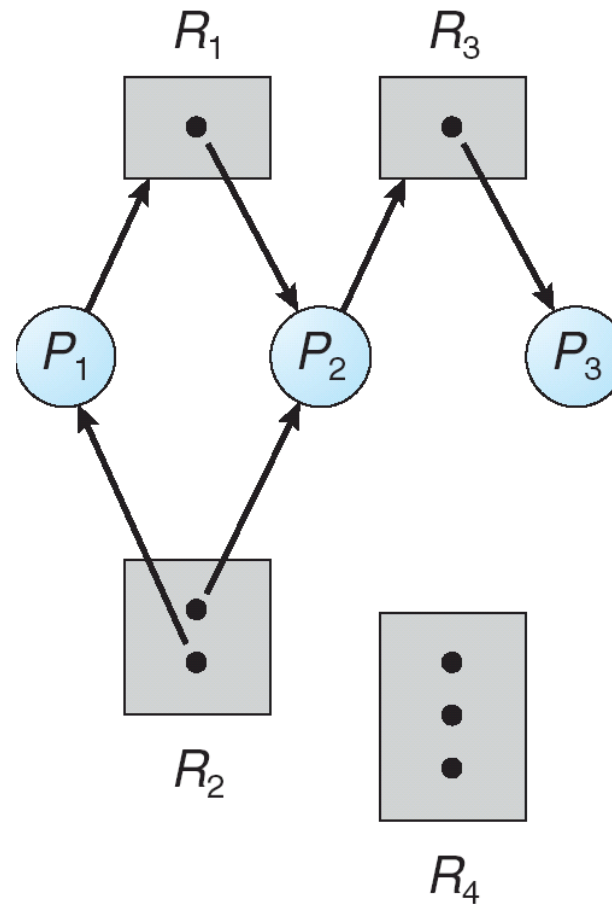


- P_i is holding an instance of R_j



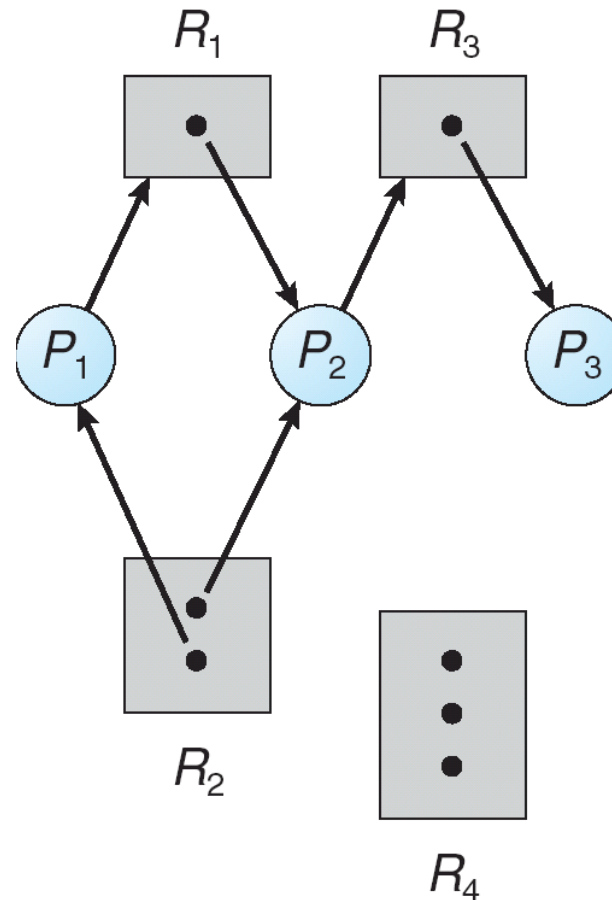


Example of a Resource Allocation Graph





Example of a Resource Allocation Graph



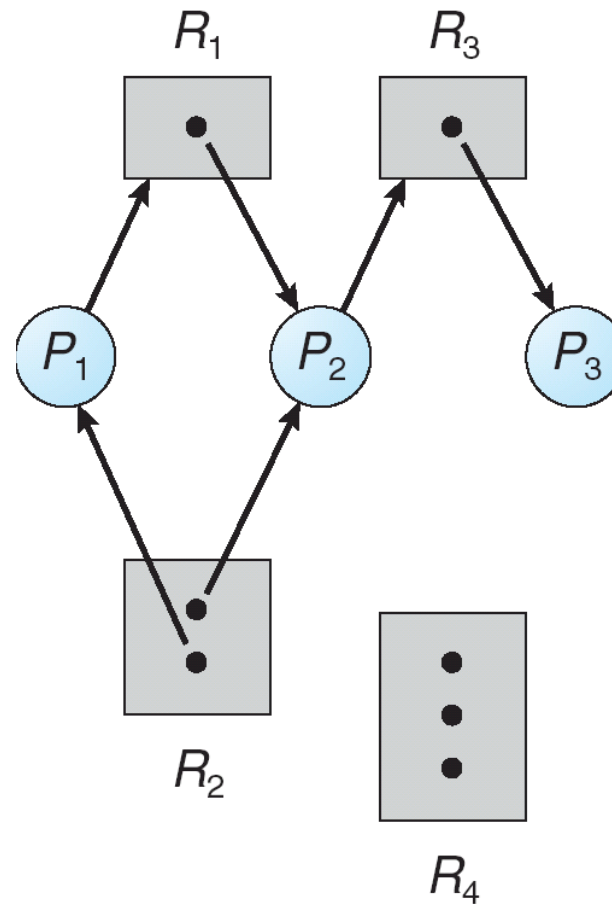
Will there be any deadlock?

Can the system execute the processes in a linear order (in a sequence)?





Example of a Resource Allocation Graph

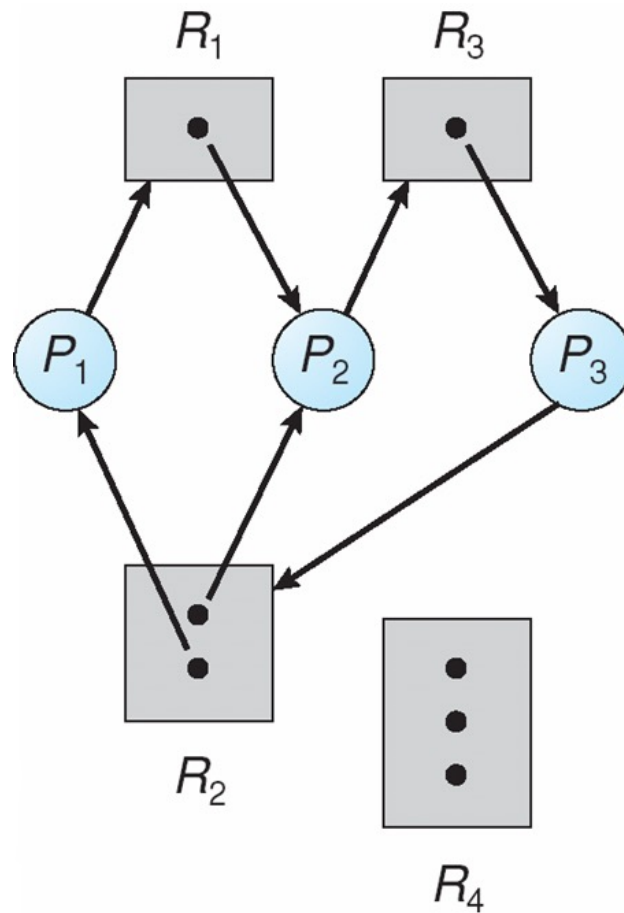


No deadlock: Execute P_3 , and then P_2 , and then P_1



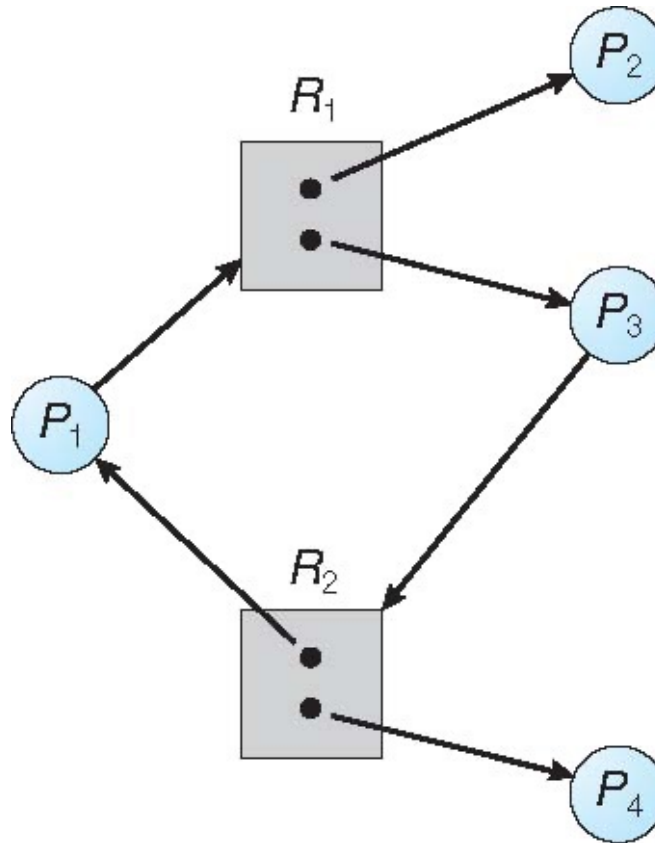


Resource Allocation Graph With A Deadlock



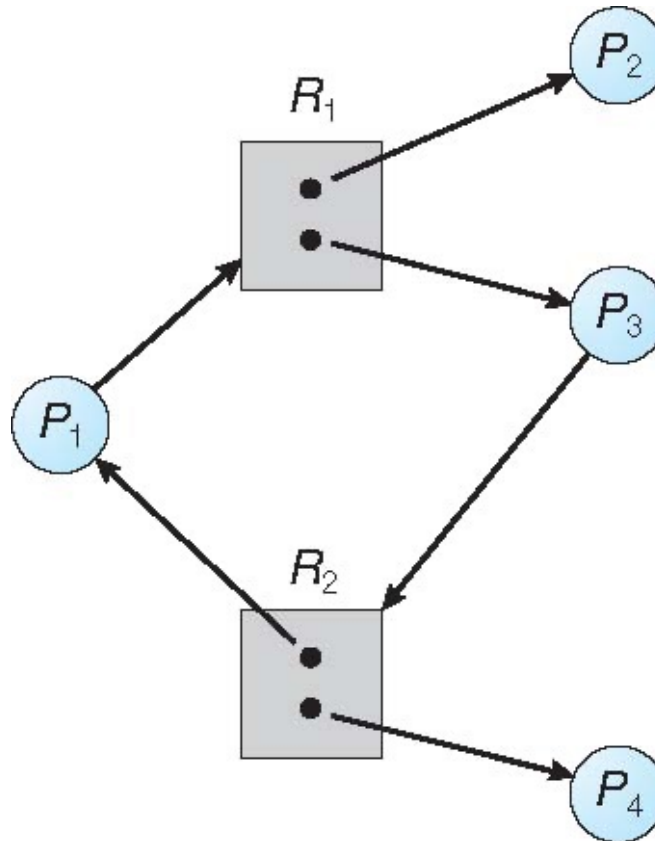


Graph With A Cycle But No Deadlock





Graph With A Cycle But No Deadlock



Order of execution can be:

- P_2, P_1, P_3, P_4
- P_4, P_3, P_1, P_2





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

- Ensure that the system will *never* enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX





Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); 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 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 allocated to it.
 - Low resource utilization; starvation possible





Example on low resources utilization

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

- Suppose there are 3 resources : **Printer, Scanner and Plotter**
- Process A needs:
 - Printer at the start of execution.
 - Plotter towards the middle of execution.
- Process B needs:
 - Scanner at the start of execution.
 - Printer towards the end of execution.





Example on low resources utilization

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

Scenario:

Process A would request **both the Printer and Plotter** right at the start

Process B would request **both the Scanner and Printer** upfront





Deadlock Prevention (Cont.)

■ No Preemption –

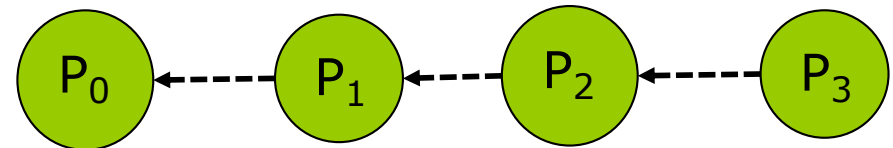
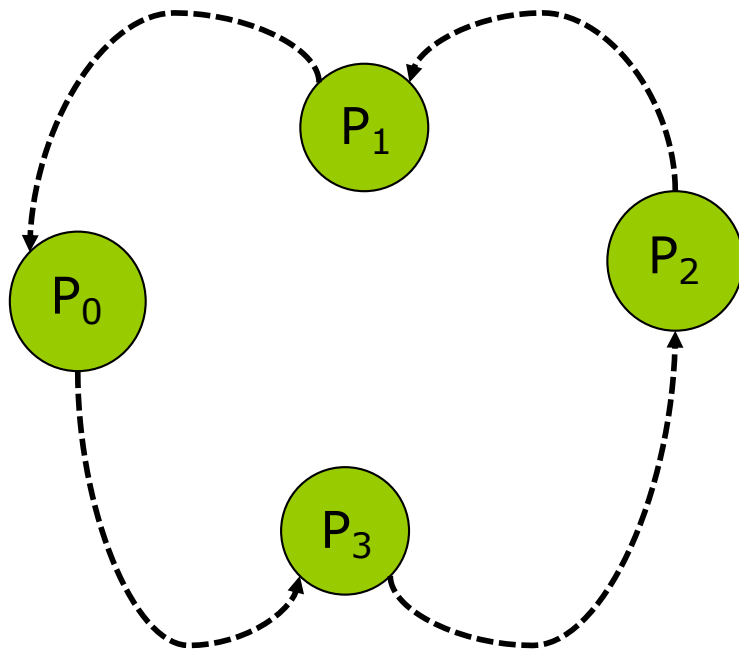
- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can 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





■ Circular wait:





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 resources,
 - the number of allocated resources,
 - the maximum demands of the processes





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





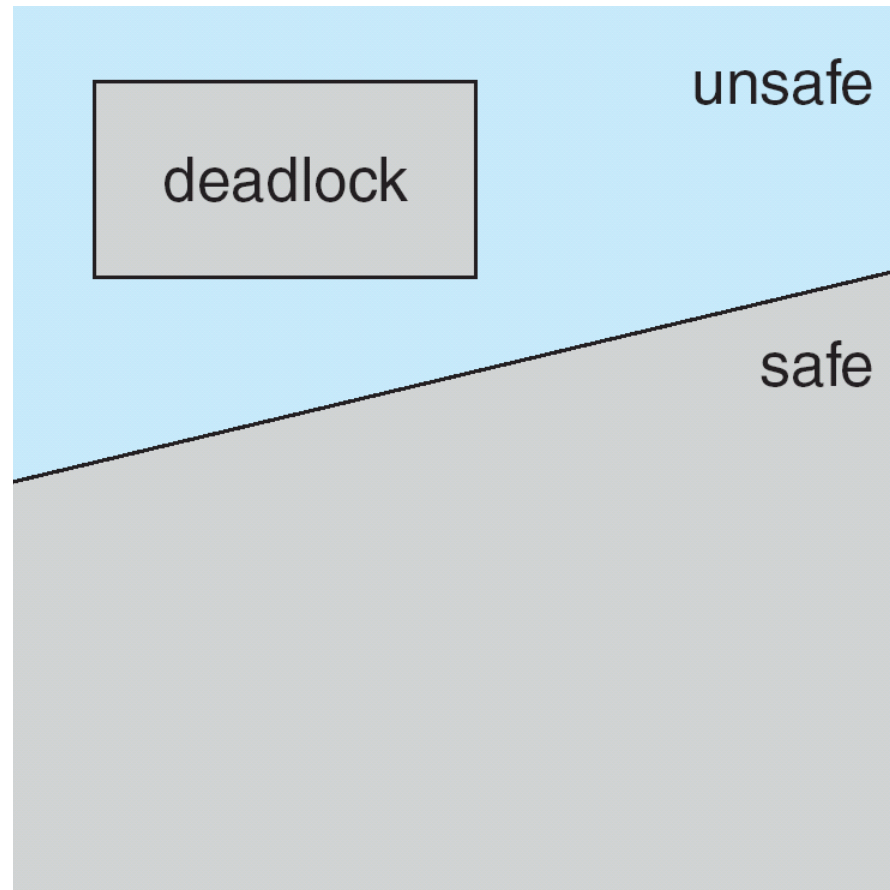
Basic Facts

- 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, Unsafe, Deadlock State

12 available resources

	<u>Maximum Needs</u>	<u>Current Needs</u>
T_0	10	5
T_1	4	2
T_2	9	2

Predict whether the processes are in the safe or unsafe state





Safe, Unsafe, Deadlock State

12 total resources

	Maximum Needs	Current Needs	Remaining Needs
T0	10	5	5
T1	4	2	2
T2	9	2	7

Available resources = $12 - (5+2+2) = 3$

With 3 available resources which process can be completed first?





Safe, Unsafe, Deadlock State

12 total resources

	Maximum Needs	Current Needs	Remaining Needs	Order of resource allocation
T0	10	5	5	
T1	4	2	2	1
T2	9	2	7	

Once T1 done, available resources become $3 + 2 = 5$





Safe, Unsafe, Deadlock State

12 total resources

	Maximum Needs	Current Needs	Remaining Needs	Order of resource allocation
T0	10	5	5	2
T1	4	2	2	1
T2	9	2	7	

Once T0 done, available resources become $5 + 5 = 10$





Safe, Unsafe, Deadlock State

12 total resources

	Maximum Needs	Current Needs	Remaining Needs	Order of resource allocation
T0	10	5	5	2
T1	4	2	2	1
T2	9	2	7	3

Once T2 done, available resources become $10 + 2 = 12$





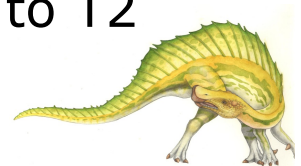
Safe, Unsafe, Deadlock State

12 total resources

Now suppose the current needs of T2 is 3,

	Maximum Needs	Current Needs	Remaining Needs	Order of resource allocation
T0	10	5	5	
T1	4	2	2	1
T2	9	2 3	6	

Deadlock Avoidance Algorithm will never assign 3 resources to T2 because it will cause Unsafe State





Avoidance Algorithms

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

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





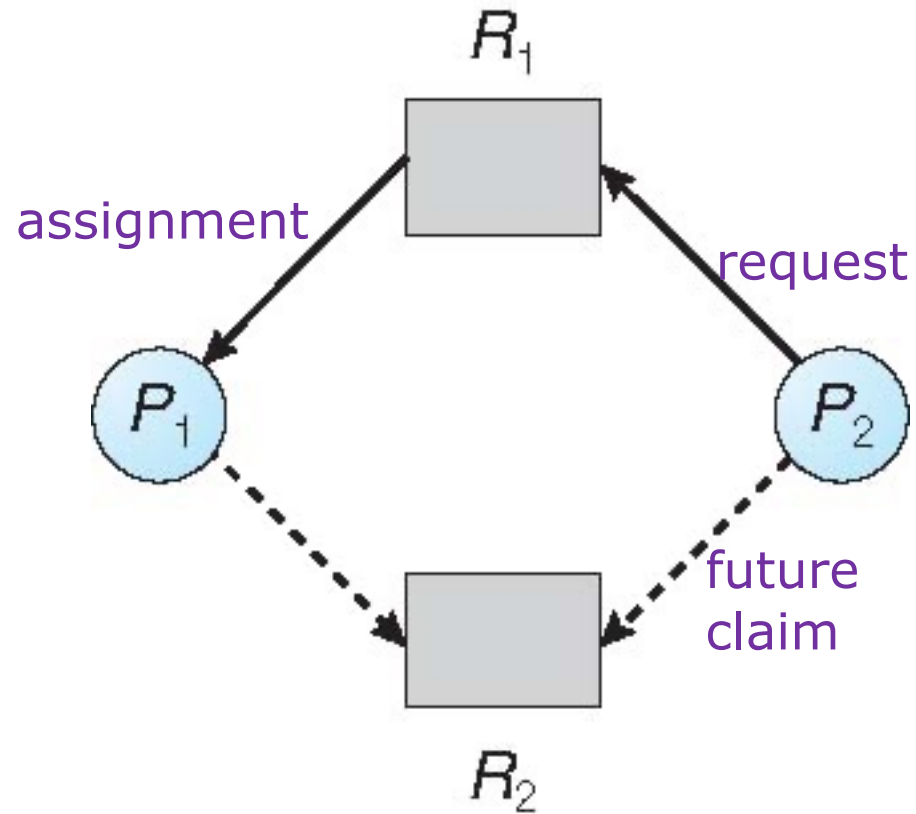
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line
 - Represented the maximum needs
 - Resources must be claimed *a priori* in the system
- **Claim edge** converts to **request edge** when a process requests a resource
- **Request edge** converted to an **assignment edge** when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge



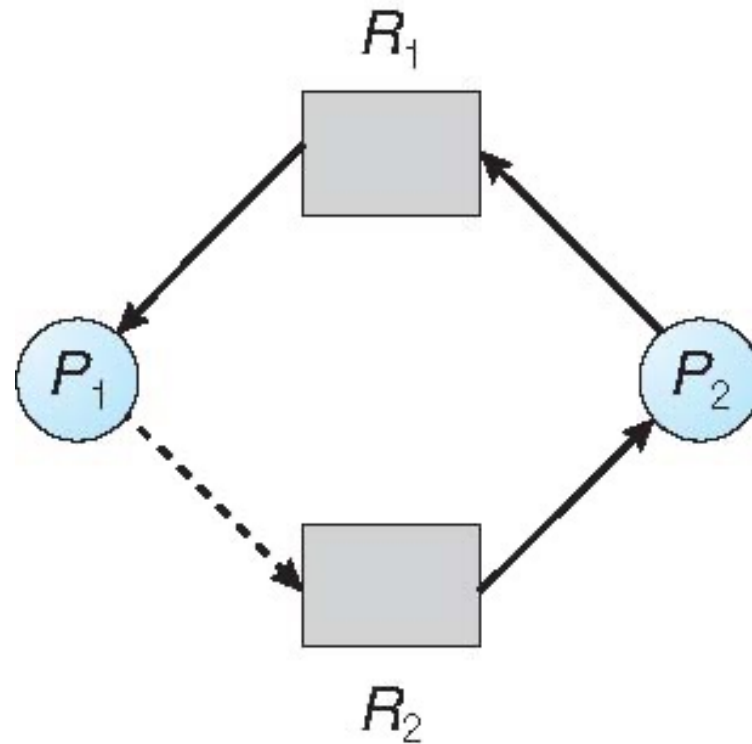


Resource-Allocation Graph





Unsafe State In Resource-Allocation Graph





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





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

1	3	3	6	...	2
---	---	---	---	-----	---

- **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

	Resource Type			
Process				





Data Structures for the Banker's Algorithm

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

- **Allocation:** $n \times m$ matrix. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j

	Resource Type			
Process				

- **Need:** $n \times m$ matrix. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$





Banker's: Safety Algorithm

Run this algorithm before allocating any resources to processes

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





Banker's: Safety Algorithm: Example

- 5 processes P_0 through P_4 ;

3 resource types:

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

- Suppose that the following snapshot represents the current state of the system:

	Allocation	Max	Available
	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: Safety Algorithm: Example (Con't)

- Determine whether the current state of the system is safe or unsafe
 - Use the safety algorithm

	Allocation	Max	Need = (Max - Allocation)	Work = Available
	A B C	A B C	A B C	A B C
P ₀	0 1 0	7 5 3	7 4 3	3 3 2
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 if there is a linear sequence in which the processes can be executed.





Let's Trace Banker's Safety Algorithm

- Determine whether the current state of the system is safe or unsafe
- Use the safety algorithm

	Allocation	Max	Need = (Max - Allocation)	Work = Available	Sequence of resource allocation	
	A B C	A B C	A B C	A B C		
P ₀	0 1 0	7 5 3	7 4 3	3 3 2	1	P ₁
P ₁	2 0 0	3 2 2	1 2 2	5 3 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			

SEQUENCE:





Banker's: 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





(1) Banker's Algorithm Example: P_1 Request (1,0,2)

- 3 resource types:
 - A (10 instances)
 - B (5 instances)
 - C (7 instances)

	Allocation	Max	Available
	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	

current state of the system

Suppose: P_1 requests 1 A, 0 B, and 2 C

Can request for (1,0,2) be granted?

**Use: Resource-Request Algorithm
for Process P_i**





(2) Banker's Algorithm Example: P_1 Request (1,0,2)

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
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	

$Request_i \leq Need_i$

$Request_i \leq Available$





(3) Banker's Algorithm Example: P_1 Request (1,0,2)

Simulate as if P_1 's request is granted

	Allocation	Max	Need	Available
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2 2 3 0
P_1	2 0 0 3 0 2	3 2 2	1 2 2 0 2 0	
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	

**Determine whether the pretend or simulated
state is safe or unsafe
Use Safety Algorithm**





(4) Let's Trace: Banker's Algorithm Example: P_1 Request (1,0,2)

	Allocation	Max	Need = (Max - Allocation)	Work = Available	Sequence of resource allocation	
	A B C	A B C	A B C	A B C		
P_0	0 1 0	7 5 3	7 4 3	2 3 0		
P_1	3 0 2	3 2 2	0 2 0			
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			

Read this note after we are done tracing:

P_1 's request (1,0,2) is granted since the simulation shows that there is a sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ that ensure the system remains in safe state





Now check your understanding on Banker's Algorithm

- 3 resource types:
 - A (10 instances)
 - B (5 instances)
 - C (7 instances)

	Allocation	Max	Available
	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	

current state of the system

- Can request for (3,3,0) by **P₄** be granted? Show your work
- Can request for (0,2,0) by **P₀** be granted? Show your work





Deadlock Detection

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





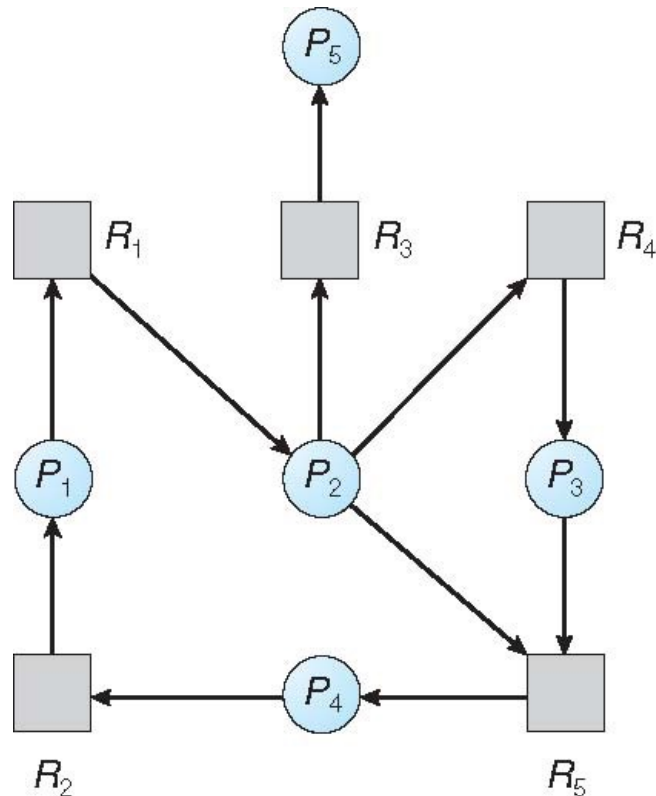
Single Instance of Each Resource Type

- 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



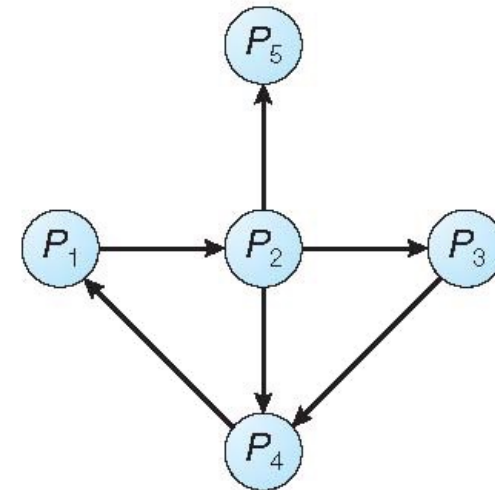


Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(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

n = number of processes or threads, and m = number of resources types.

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





Deadlock Detection Algorithm: Example 1

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- The following snapshot represents the current state of the system:

	Allocation	Request	Available
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
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	

Run Detection Algorithm to check whether there is a deadlock





Let's Trace Deadlock Detection Algorithm: Example 1

	Allocation	Request	Work = Available	Finish
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>	
P₀	0 1 0	0 0 0	0 0 0	F
P₁	2 0 0	2 0 2		F
P₂	3 0 3	0 0 0		F
P₃	2 1 1	1 0 0		F
P₄	0 0 2	0 0 2		F

SEQUENCE:





Detection Algorithm Example 2

- 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





Example (Cont. 2)

	Allocation	Request	Work = Available	Finish
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>	
P₀	0 1 0	0 0 0	0 0 0	T
P₁	2 0 0	2 0 2	0 1 0	F
P₂	3 0 3	0 0 1		F
P₃	2 1 1	1 0 0		F
P₄	0 0 2	0 0 2		F

Deadlock exists, consisting of processes **P₁**, **P₂**, **P₃**, and **P₄**





Detection-Algorithm Usage

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





Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
 1. Priority of the process
 2. How long process has computed, and how much longer to completion
 3. Resources the process has used (less)
 4. Resources process needs to complete (more)
 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 8

