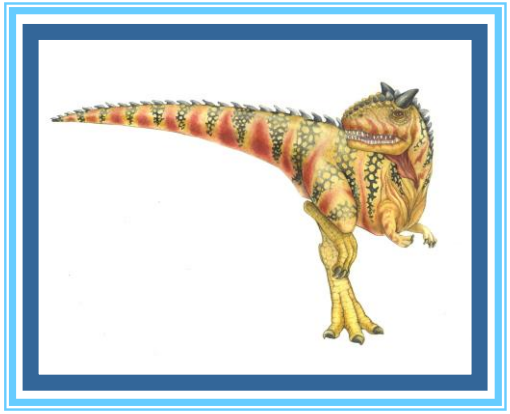
Chapter 7: Deadlocks



Silberschatz, Galvin and Gagne ©2013

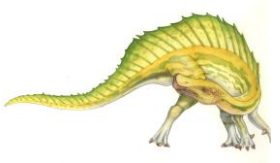
Chapter 7: Deadlocks

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

System Model

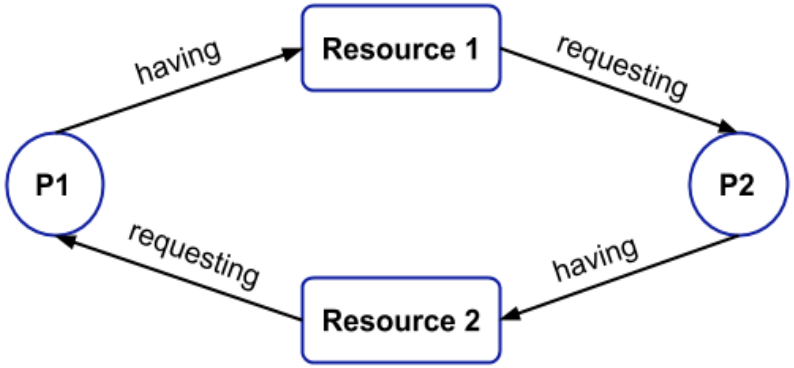
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  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**



What is Deadlock?

- Deadlock is a situation where two or more processes are waiting for each other. For example, let us assume, we have two processes P1 and P2. Now, process P1 is holding the resource R1 and is waiting for the resource R2. At the same time, the process P2 is having the resource R2 and is waiting for the resource R1. So, the process P1 is waiting for process P2 to release its resource and at the same time, the process P2 is waiting for process P1 to release its resource. And no one is releasing any resource. So, both are waiting for each other to release the resource. This leads to infinite waiting and no work is done here. This is called Deadlock.

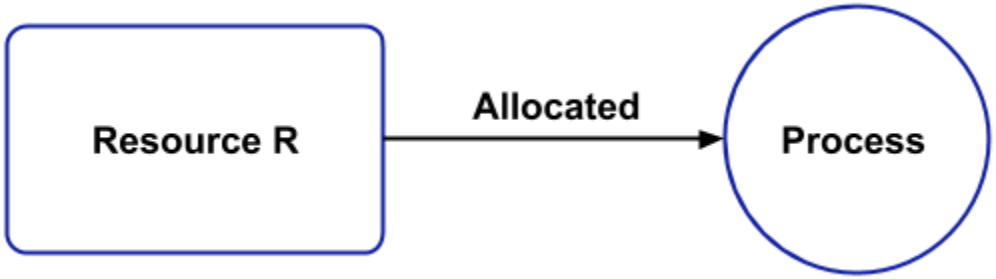


If a process is in the waiting state and is unable to change its state because the resources required by the process is held by some other waiting process, then the system is said to be in Deadlock.

Deadlock Characterization

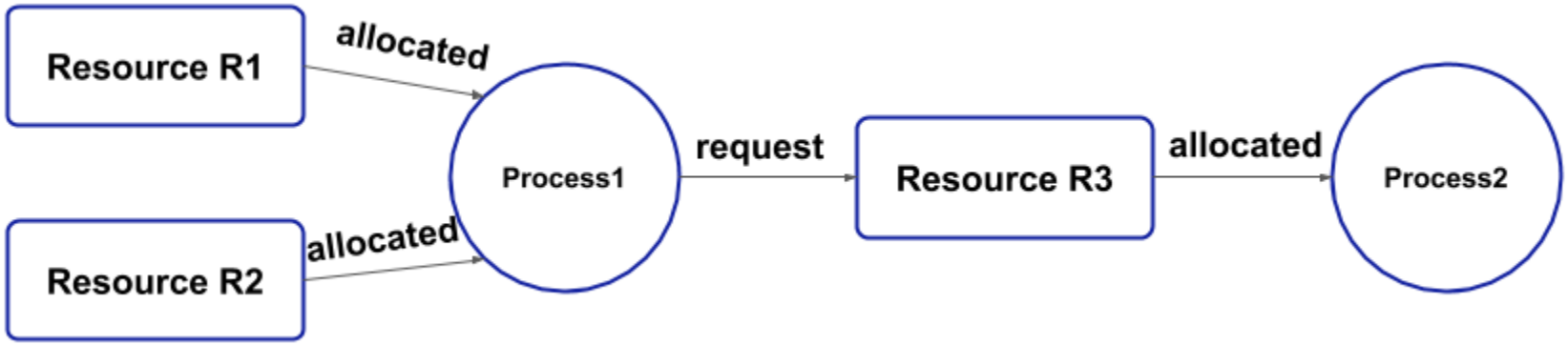
Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource.
- In other words, if a process P1 is using some resource R at a particular instant of time, then some other process P2 can't hold or use the same resource R at that particular instant of time. Process P2 can request that resource R but it can't use that resource simultaneously with process P1.



Deadlock Characterization

- **Hold and wait:** A process holding at least one resource and is waiting to acquire additional resources held by other processes
- For example, a process P1 can hold two resources R1 and R2, and at the same time, it can request some resource R3 that is currently held by process P2.

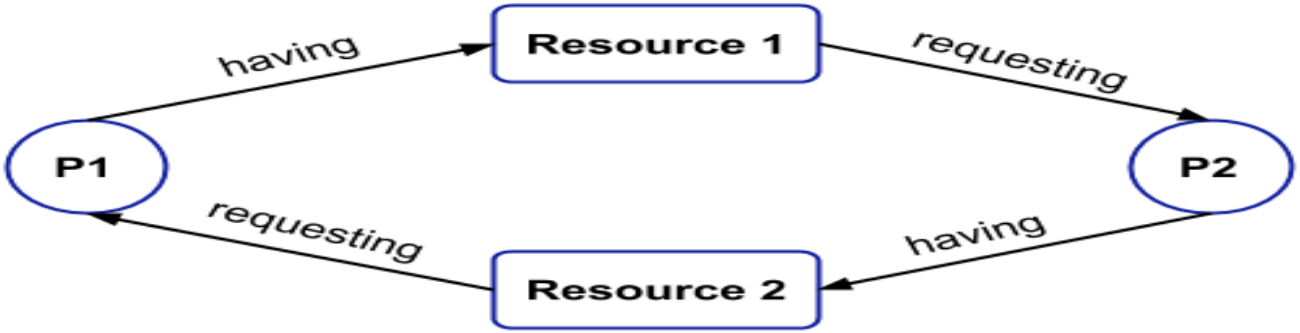


Deadlock Characterization

- **No preemption:** a resource can be released only voluntarily by the process holding it after that process has completed its task. In other words, resources can't be preempted from the process by another process, forcefully.
- For example, if a process P1 is using some resource R, then some other process P2 can't forcefully take that resource.
- process P2 can request the resource R and can wait for that resource to be freed by process P1.

Deadlock Characterization

- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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.
- In other words Circular wait is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on. At last, the last process is waiting for the resource held by the first process. So, every process is waiting for each other to release the resource, and no one is releasing their own resource. Everyone is waiting here for getting the resource. This is called a circular wait.



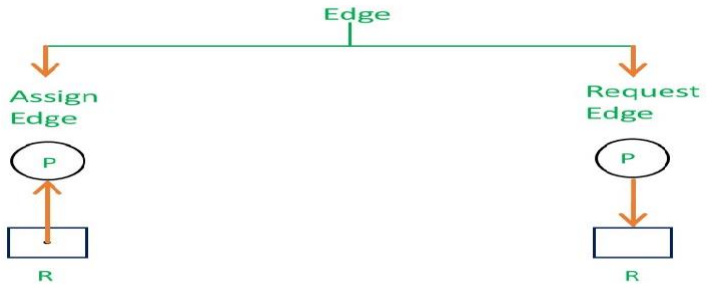
agne ©2013

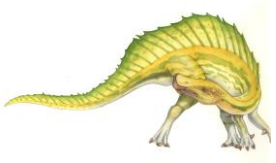
Resource-Allocation Graph

- Resource Allocation Graph (RAG) is a graph that represents the state of a system pictorially.

There are two components of RAG-A set of vertices *V* and a set of edges *E*.

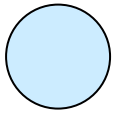
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system **(Denoted by a circle)**
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system **(Denoted by Square)**
- **Request edge–directed** edge $P_i \rightarrow R_j$
- **Assignment edge–directed** edge $R_j \rightarrow P_i$



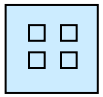


Resource-Allocation Graph (Cont.)

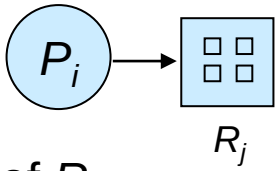
- Process



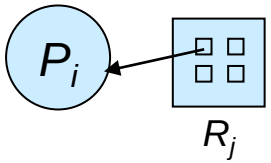
- Resource Type with 4 instances



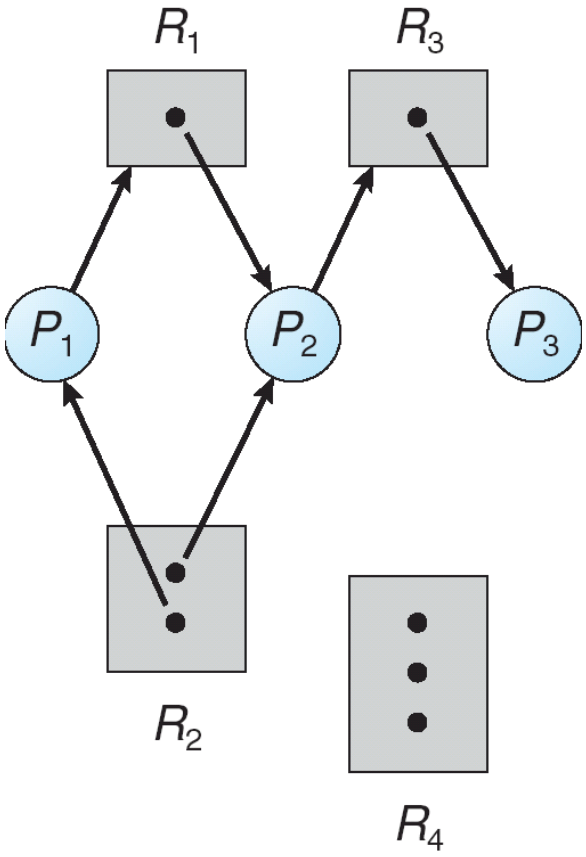
- P_i requests instance of R_j



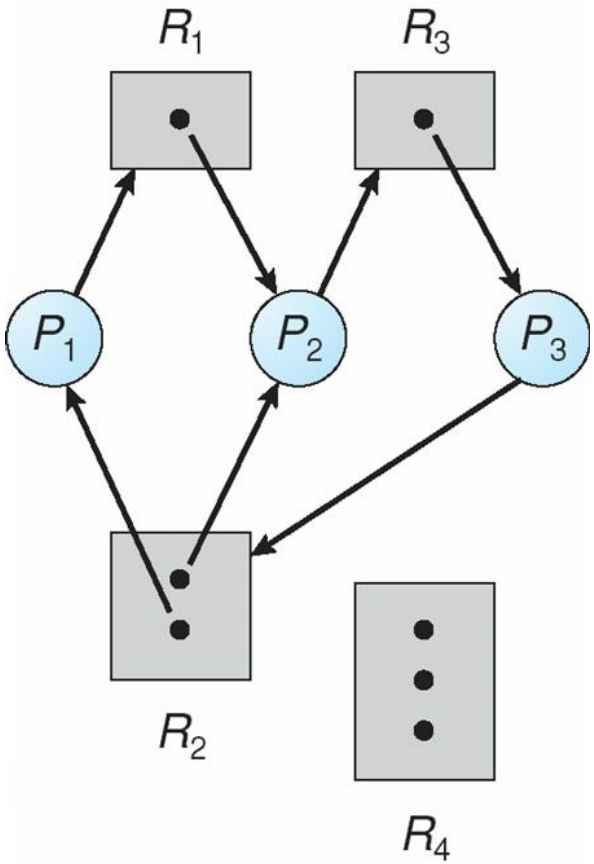
- P_i is holding an instance of R_j

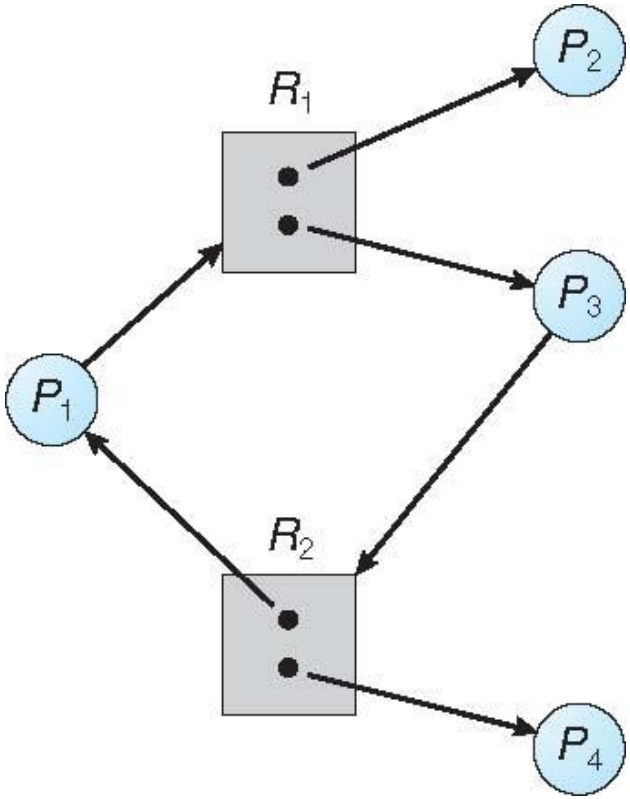


Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



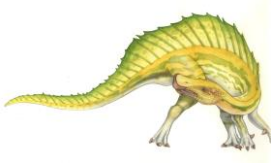

P_2
R_1
P_3
P_1
R_2
P_4

Basic Facts

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

Methods for Handling Deadlocks

- We can use a protocol to prevent or avoid deadlocks to ensure that the system will ***never*** enter a deadlock state:
- **Deadlock Prevention:** design such a system that violates at least one of four necessary conditions of deadlock and ensures independence from deadlock.
- **Deadlock Avoidance**: The system maintains a set of data using which it decides whether to entertain a new request or not, to be in a safe state.
- **Deadlock Detection and Recovery:** Allow the system to enter a deadlock state and then recover.
- **Deadlock Ignorance:** We ignore the problem as if it does not exist. The Ostrich algorithm is used in deadlock Ignorance.

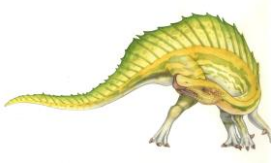


Deadlock Prevention

- This strategy involves designing a system that violates one of the four necessary conditions required for the occurrence of deadlock.
- This ensures that the system remains free from the deadlock.

Mutual Exclusion –

- To violate this condition, all the system resources must be such that they can be used in a shareable mode.
- In a system, there are always some resources that are mutually exclusive by nature.
- So, this condition can not be violated.



Deadlock Prevention

Hold and Wait –

- **Conservative Approach:** The process is allowed to start execution if and only if it has acquired all the resources.
  - Less Efficient.
  - Not Implementable.
  - Easy
- **Do not Hold:** The process will acquire only desired resources but before making any request it must release all the resources that it holds currently.
  - Efficient
  - Implementable
- **Wait for Timeout:** We place a maximum time up to which a can process after which it must release all the holding resources and exit.

Deadlock Prevention (Cont.)

No Preemption –

- **Forceful Preemption:** We allow a process to forcefully preempt the resources held by other processes.
- This method may be used by high-priority processes or system processes.
- The process which is waiting must be selected as a victim, instead of the process in the running state.

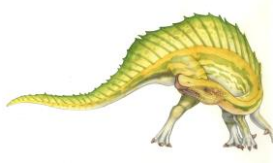
Deadlock Prevention (Cont.)

Circular Wait:

- This condition can be violated by not allowing the processes to wait for resources in a cyclic manner.
- To violate this condition, the following approach is followed-

Approach-

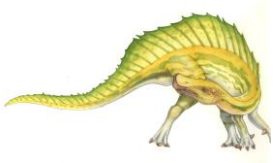
- A natural number is assigned to every resource.
- Each process is allowed to request the resources either in only increasing or decreasing order of the resource number.
- In case increasing order is followed, if a process requires a lesser number of resources, then it must release all the resources having a larger number and vice versa.
- This approach is the most practical approach and implementable.
- However, this approach may cause starvation but will never lead to deadlock.



Deadlock Avoidance

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

- The 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.



Deadlock Avoidance

	Max Need		
	A	B	C
P0			
P1			
P2			
P3			

	Allocation		
	A	B	C
P0			
P1			
P2			
P3			

	Current Need		
	A	B	C
P0			
P1			
P2			
P3			

System Max		
A	B	C

Available		
A	B	C

Deadlock Avoidance

	Max Need		
	A	B	C
P0	4	3	1
P1	2	1	4
P2	1	3	3
P3	5	4	1

	Allocation		
	A	B	C
P0	1	0	1
P1	1	1	2
P2	1	0	3
P3	2	0	0

	Current Need		
	A	B	C
P0			
P1			
P2			
P3			

System Max		
A	B	C
8	4	6

Available		
A	B	C

Deadlock Avoidance

	Max Need		
	A	B	C
P0	4	3	1
P1	2	1	4
P2	1	3	3
P3	5	4	1

	Allocation		
	A	B	C
P0	1	0	1
P1	1	1	2
P2	1	0	3
P3	2	0	0

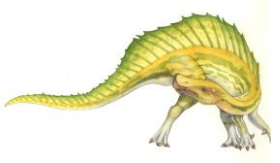
	Current Need		
	A	B	C
P0	3	3	0
P1	1	0	2
P2	0	3	0
P3	3	4	1

System Max		
A	B	C
8	4	6

Available		
A	B	C
3	3	0

Deadlock Avoidance

- A deadlock-avoidance algorithm dynamically examines the resource-allocation state.
- The resource- allocation state is defined by the number of available and allocated resources and the maximum demands of the processes before allowing that request first
- We check, if there exist "some sequence in which we can satisfies demand of every process without going into deadlock, if yes, this sequence is called safe sequence" and request can be allowed. Otherwise there is a possibility of going into deadlock.



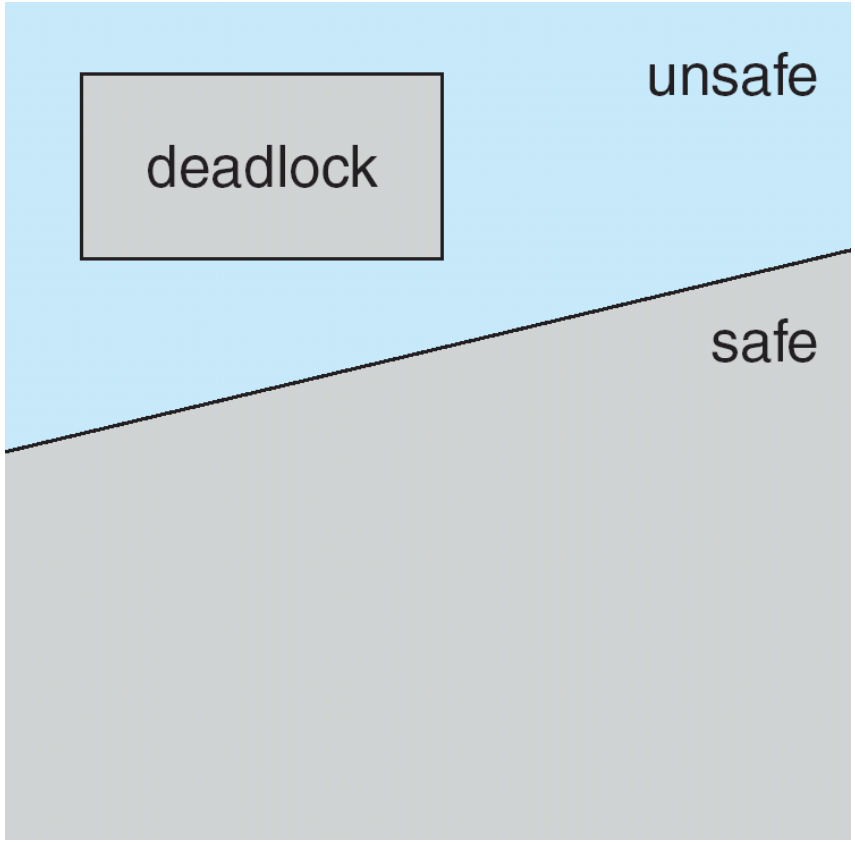
Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state
- System is in a **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ 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,
- 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, ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State

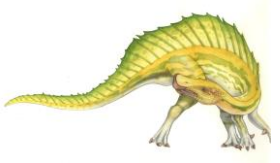


Avoidance Algorithms

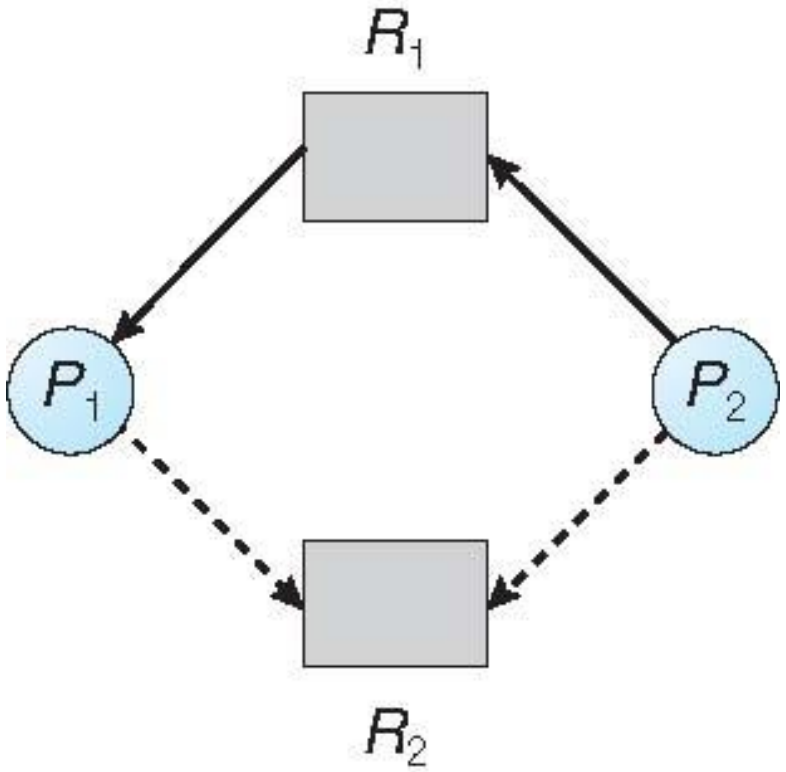
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

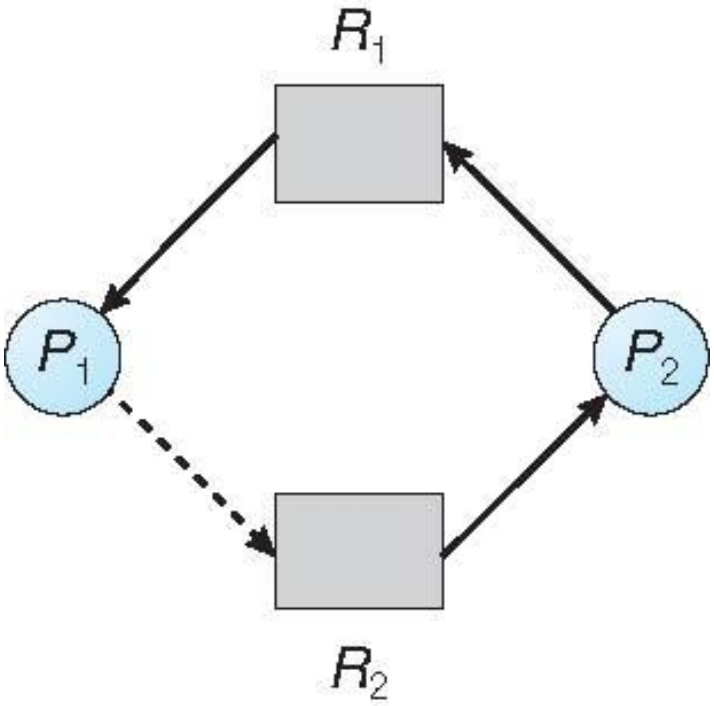
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource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

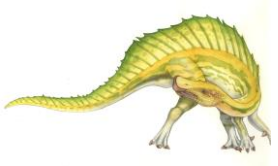


Resource-Allocation Graph



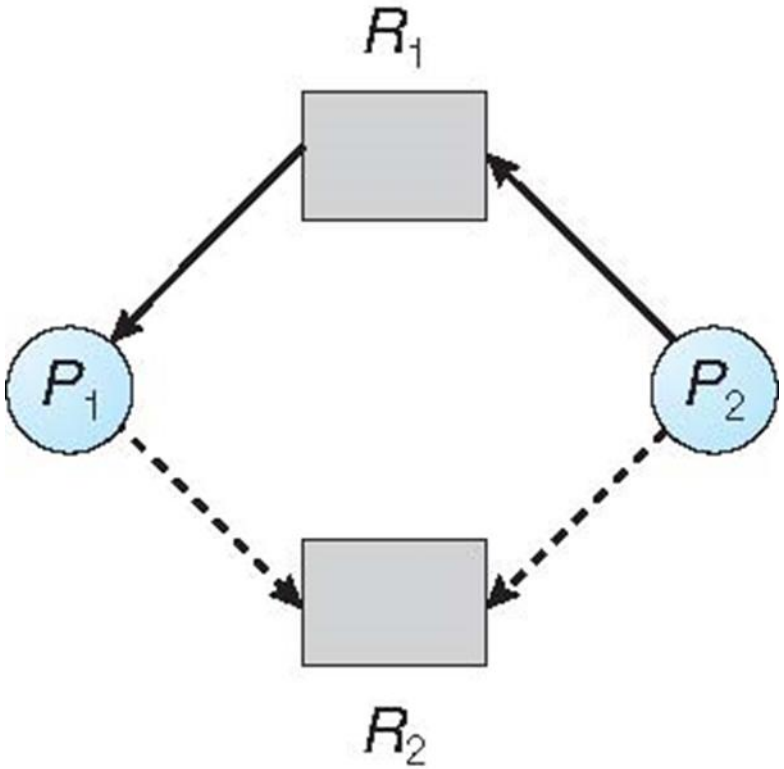
Unsafe State In Resource-Allocation Graph

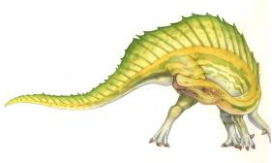




Resource-Allocation Graph Algorithm

- 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

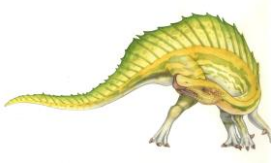
- Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.
- Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Every time a loan has to be granted by the bank; it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.
- Banker's algorithm works in a similar way in computers. It is used as deadlock avoidance algorithm with several instances of resources.
- Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.
- When a process requests a resource, and allocation of resource may lead to a deadlock state then it may have to wait otherwise the resource is allocated to the process.
- 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 assume n = number of processes, and m = number of resources types. Following data structures are used to implement the bankers algorithm:

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x 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, …, *n*- 1**

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

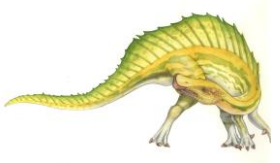
   (a) ***Finish* [*i*] = *false***

   (b) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$
   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

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

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

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

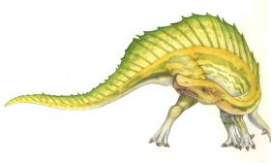
- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

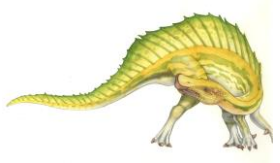
	Allocation	*Max*	*Available*	*Need (Max-Allocation)*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1



Example (Cont.)

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

	Need
	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



Example (Cont.)

Need

	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

- **Is the system in a safe state? If Yes, then what is the safe sequence?** Applying the Safety algorithm on the given system,

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ (7,4,3) > Work (3,3,2)

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $Need_1$ (1,2,2) < Work (3,3,2)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0)

Work =	A	B	C
	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ (6, 0, 0) > Work (5,3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ (0, 1, 1) < Work (5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)

Work =	A	B	C
	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ (4, 3, 1) < Work (7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)

Work =	A	B	C
	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need (7, 4, 5) < Work (7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1 , 0)

Work =	A	B	C
	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ (6, 0, 0) < Work (7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

Work =	A	B	C
	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Finish [i] = true for 0 ≤ i ≤ n — Step 4

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Example: P_1 Request (1,0,2)

- Suppose that process P_1 request an additional instance of resource of type A and two instances of resource type C.

A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

$Request_1 \leq Need_1$

1, 0, 2 ≤ 1, 2, 2 ✓

Step 2

$Request_1 \leq Available$

1, 0, 2 ≤ 3, 3, 2 ✓

	Allocation	*Max*	*Available*	*Need (Max-Allocation)*
	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

Step 3

$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

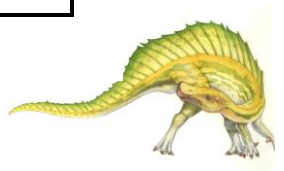
$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	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	

Example: P_1 Request (1,0,2)

- We pretend that this request has been fulfilled and we arrive at the following new state:

Process	Allocation	Need	Available
	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	



Example: P_1 Request (1,0,2) (Cont.)

- **Is the system in a safe state? If Yes, then what is the safe sequence?**

Applying the Safety algorithm on the given system.

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 2 | 3 | 0 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ (7, 4, 3) > Work (2, 3, 0)

So P_0 must wait

But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 0, 2, 0

Finish [1] is false and $Need_1$ (0, 2, 0) < Work (2, 3, 0)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (2, 3, 0 + 3, 0, 2)

Work =	A	B	C
	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6, 0, 0

Finish [2] is false and $Need_2$ (6, 0, 0) > Work (5, 3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ (0, 1, 1) < Work (5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)

Work =	A	B	C
	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ (4, 3, 1) < Work (7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)

Work =	A	B	C
	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need (7, 4, 5) < Work

So P_0 must be kept in safe sequence

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1, 0)

Work =	A	B	C
	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6, 0, 0

Finish [2] is false and $Need_2$ (6, 0, 0) < Work (7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

Work =	A	B	C
	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Finish [i] = true for 0 ≤ i ≤ n — Step 4

Hence the system is in Safe state

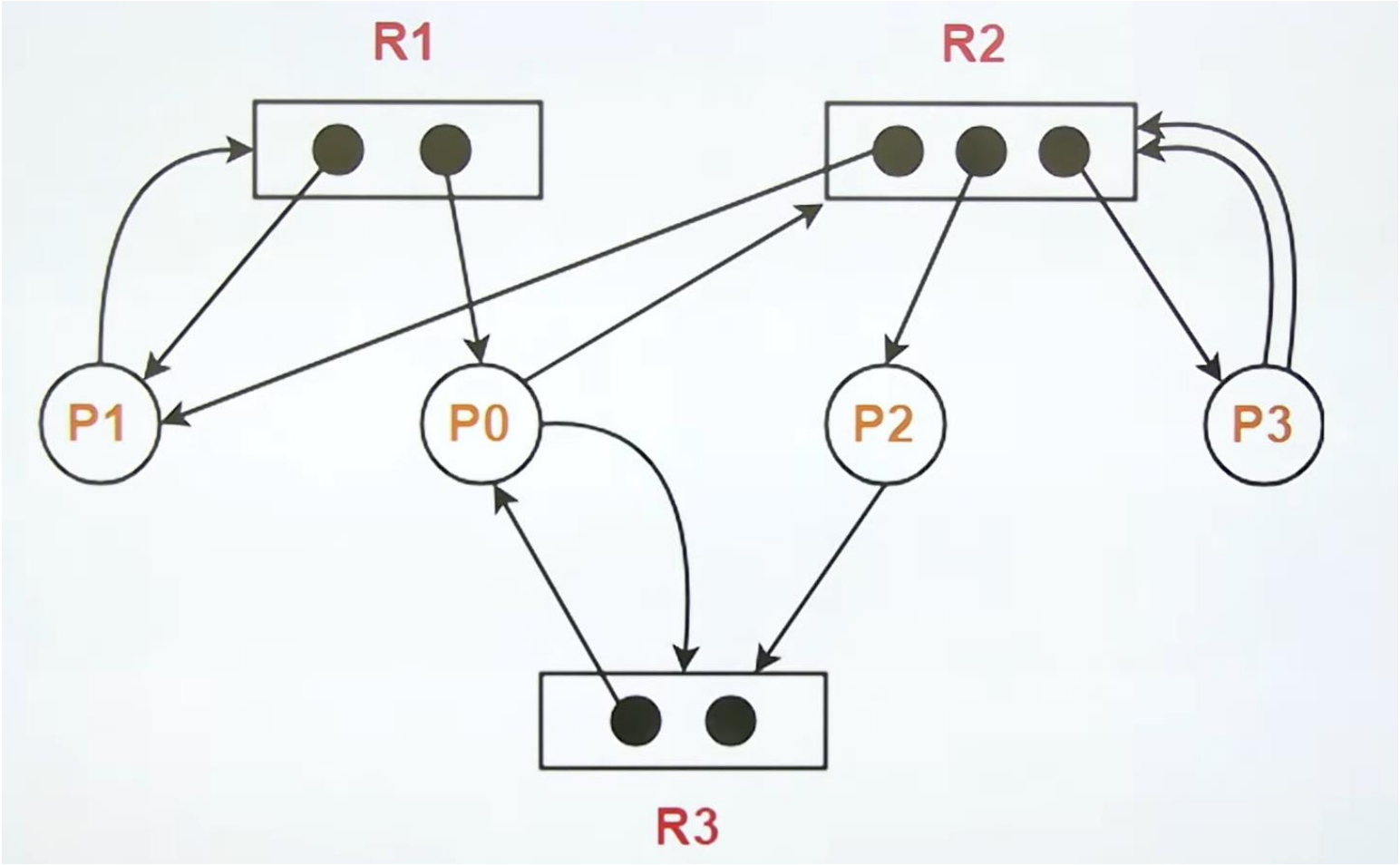
The safe sequence is P_1, P_3, P_4, P_0, P_2

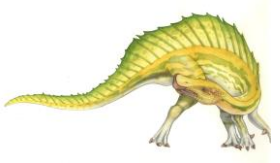
Example: P_1 Request (1,0,2) (Cont.)

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement. Hence we can immediately grant the request from Process P1.

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?
- Can request for (0,0,2) by P_3 be granted?

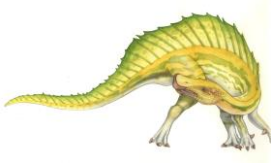
Find that System is in Deadlock or Not





Deadlock Detection

- If a system does not employ 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 has occurred
- An algorithm to recover from the deadlock
- It means allowing the system to enter a deadlock state and then
  - Apply the Detection algorithm
  - Apply Recovery scheme

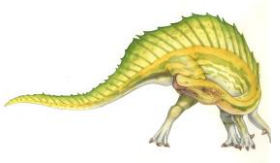


Deadlock Detection

- In the following discussion, we elaborate on these two requirements as they pertain to systems with only
- A single instance of each resource type
- A system with several instances of each resource type.

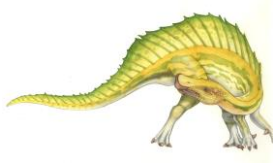
Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq

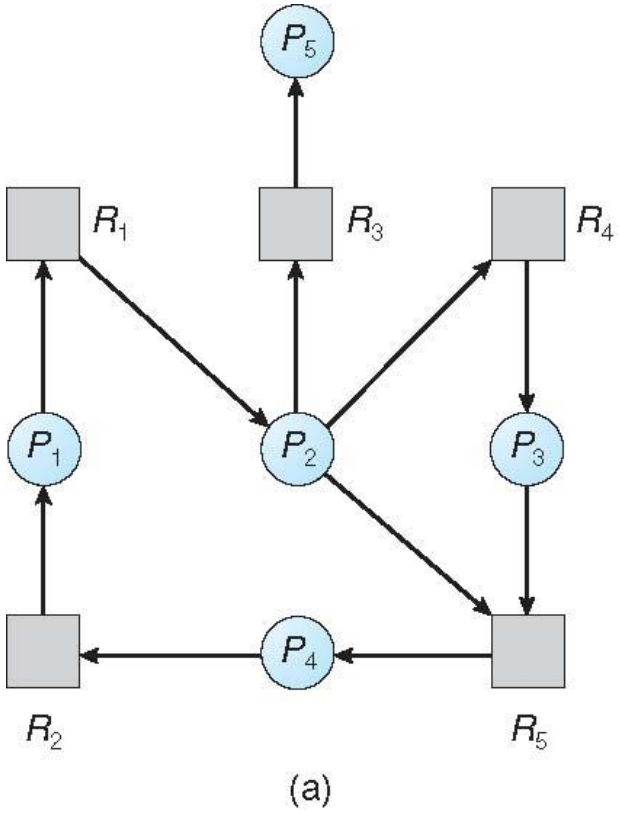


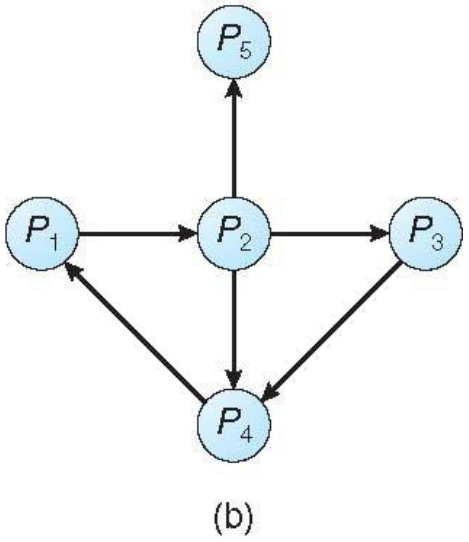
Single Instance of Each Resource Type

- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to 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





Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- The wait-for-graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock detection algorithm that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

Several Instances of a Resource Type

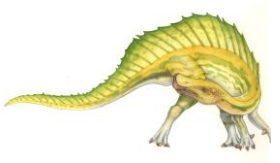
- **Available***:* A vector of length ***m*** indicates the number of available resources of each type
- **Allocation***:* An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process
- **Request***:* An ***n*** **x** ***m*** matrix indicates the current 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, …, *n***, if ***Allocation*$_i$ ≠ 0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***
2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request*$_i$ ≤ *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] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state.

   Moreover, if ***Finish*[*i*] == *false***, then ***P*$_i$** is deadlocked

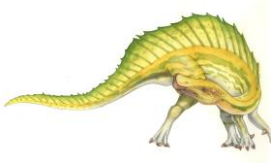


Example of Detection Algorithm

- 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:

	Allocation	*Request*	*Available*
	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	



Example of Detection Algorithm

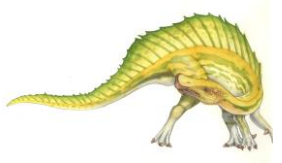
Solution:

- Snapshot at time T_0:

	Allocation	Request	Available
	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	

1. In this, Work = [0, 0, 0] &
   Finish = [false, false, false, false, false]
2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].
4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
   Finish = [true, false, true, false, false].
6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 2] =>[5, 1, 5] &
   Finish = [true, true, true, false, false].
8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 5].
9. Work =[5, 1, 5]+[2, 1, 1] =>[7, 2, 6] &
   Finish = [true, true, true, true, false].
10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 6].
11. Work =[7, 2, 6]+[0, 0, 2] =>[7, 2, 8] &
    Finish = [true, true, true, true, true].
12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish[i] = true* for all *i*



Example (Cont.)

- P_2 requests an additional instance of type **C**

	Request
	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

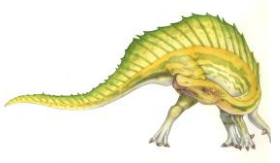
- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
- One is simply to abort one or more processes to break the circular wait.
- The other is to preempt some resources from one or more of the deadlocked processes.

Recovery from Deadlock: Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes

- **Abort all deadlocked processes**: This method clearly will break the deadlock cycle but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later

- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.



Recovery from Deadlock: Process Termination

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
- Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.
- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions.
- The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one.
- Many factors may affect which process is chosen, including:

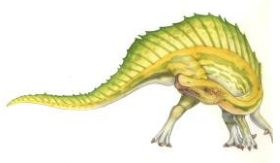
1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

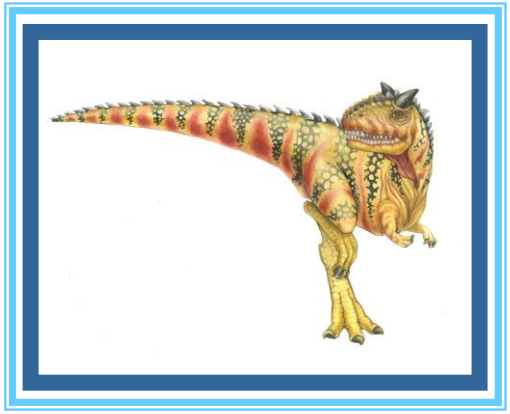
- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed
- **Rollback** – If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.
- **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.



End of Chapter 7



Silberschatz, Galvin and Gagne ©2013