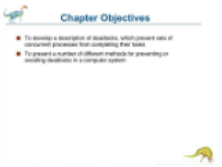
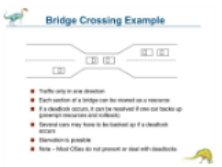
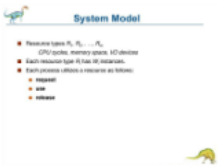
Deadlocks


Chapter 7: Deadlocks
The Deadlock Problem
Characterization
Handling Deadlocks
Prevention
Avoidance
Recovery from Deadlock

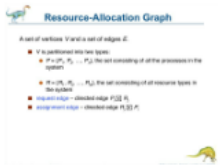

Chapter Objectives
To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
To present a number of different methods for preventing or avoiding deadlocks in a computer system

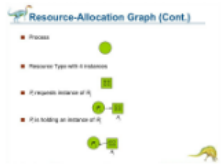

The Deadlock Problem
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
Example
System has 2 disk drives
P1 and P2 each hold one disk drive and each needs another one
Example
semaphores A and B, initialized to 1
wait (A); wait(B)
wait (B); wait(A)

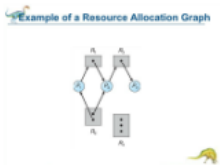

Bridge Crossing Example
Traffic only in one direction
Each section of a bridge can be viewed as a resource
If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
Several cars may have to be backed up if a deadlock occurs
Starvation is possible
Note – Most Chips do not prevent or deal with deadlocks

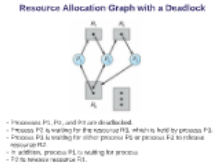

System Model
Resource types R1, R2, . . ., Rm
CPU cycles, memory space, I/O devices
Each resource type Ri has Wi instances.
Each process utilizes a resource as follows:
request
use
release

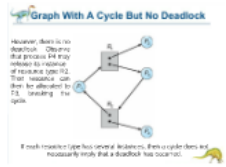

Deadlock Characterization
Deadlock can arise if four conditions hold simultaneously.
Mutual exclusion: only one process at a time can use a resource
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
Circular wait: there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.


Resource-Allocation Graph
A set of vertices V and a set of edges E.
V is partitioned into two types:
P = {P1, P2, …, Pn}, the set consisting of all the processes in the system
R = {R1, R2, …, Rm}, the set consisting of all resource types in the system
request edge – directed edge Pi → Rj
assignment edge – directed edge Rj → Pi

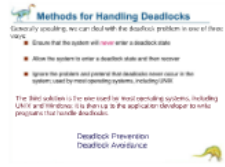

Resource-Allocation Graph (Cont.)
Process
Resource Type with 4 instances
Pi requests instance of Rj
Pi is holding an instance of Rj


Example of a Resource Allocation Graph

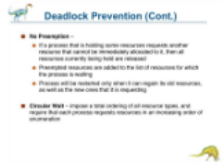

Resource Allocation Graph with a Deadlock
Processes P1, P2, and P3 are deadlocked.
Process P2 is waiting for the resource R3, which is held by process P3.
Process P3 is waiting for either process P1 or process P2 to release resource R2.
In addition, process P1 is waiting for process
P2 to release resource R1.

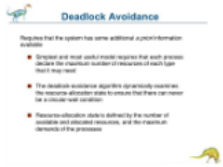

Graph With A Cycle But No Deadlock
If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.


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

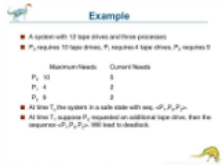

Methods for Handling Deadlocks
Generally speaking, we can deal with the deadlock problem in one of three ways:
Ensure that the system will never enter a deadlock state
Allow the system to enter a deadlock state and then recover
Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
Deadlock Prevention
Deadlock Avoidance

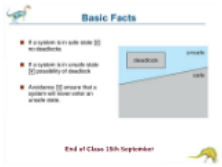

Deadlock Prevention
Restrain the ways request can be made
Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources
Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
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

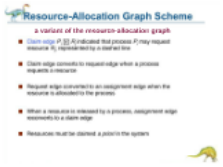

Deadlock Avoidance
Requires that the system has some additional a priori information available
Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
The deadlock-avoidance algorithm dynamically examines the resource-allocation 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.

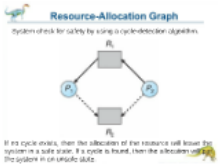

Safe State
When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
System is in safe state if there exists a sequence <P1, P2, …, Pn> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j<i
That is:
If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.
When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
When Pi terminates, Pi+1 can obtain its needed resources, and so on
Not all unsafe states are deadlocks; however an unsafe state may lead to a deadlock

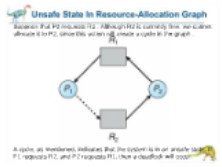

Example
A system with 12 tape drives and three processes
P0 requires 10 tape drives, P1 requires 4 tape drives, P2 requires 9
Maximum Needs Current Needs
P0 10 5
P1 4 2
P2 9 2
At time T0, the system in a safe state with seq. <P1, P0, P2>.
At time T1, suppose P2 requested an additional tape drive, then the sequence <P1, P0, P2> will lead to deadlock.


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.
deadlock
unsafe
safe
End of Class 15th September


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

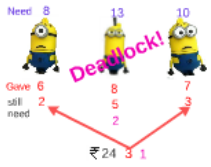

Resource-Allocation Graph Scheme
a variant of the resource-allocation graph
Claim edge Pi → Rj indicated that process Pi may request resource Rj; represented by a dashed line
Claim edge converts to request edge when a process requests a resource
Request edge converted to an assignment edge when the resource 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
If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

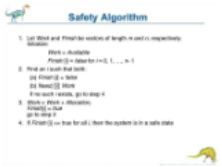

Unsafe State In Resource-Allocation Graph

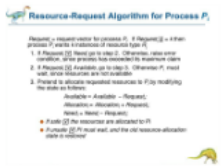

Resource-Allocation Graph Algorithm
Suppose that process Pi requests a resource Rj
The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

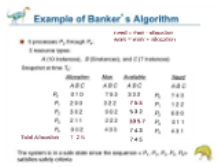

Banker's Algorithm
Multiple instances
Each process must a priori claim maximum use
When a process requests a resource it may have to wait
When a process gets all its resources it must return them in a finite amount of time

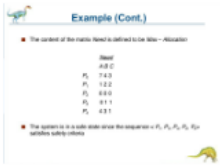

Need 8
13
10
Deadlock!
Gave 6
8
7
still need 2
5
3
2
₹24 3 1

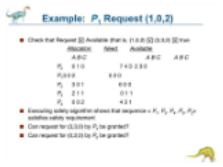

Data Structures for the Banker's Algorithm
Available: Vector of length m. If Available [j] = k, there are k instances of resource type Rj available
Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj
Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task
Need[i,j] = Max[i,j] – Allocation [i,j]


Safety Algorithm
Let Work and Finish be vectors of length m and n, respectively.
Initialize:
Work = Available
Finish [i] = false for i=0, 1, …, n-1
Find an i such that both:
(a) Finish [i] = false
(b) Need[i] ≤ Work
If no such i exists, go to step 4
Work = Work + Allocation
Finish[i] = true
go to step 2
If Finish [i] == true for all i, then the system is in a safe state


Resource-Request Algorithm for Process Pi
Request i = request vector for process Pi. If Request i [j] = k then process Pi wants k instances of resource type Rj
If Request i ≤ Need i go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
If Request i ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available
Pretend to allocate requested resources to Pi by modifying the state as follows:
Available = Available – Request i;
Allocation i = Allocation i + Request i;
Need i = Need i – Request i;
If safe ⇒ the resources are allocated to Pi
If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored


Example of Banker's Algorithm
5 processes P0 through P4;
3 resource types:
A (10 instances), B (5instances), and C (7 instances)
Snapshot at time T0:
Allocation Max Available Need
A B C A B C A B C A B C
P0 0 1 0 7 5 3 3 3 2 7 4 3
P1 2 0 0 3 2 2 7 4 5 1 2 2
P2 3 0 2 9 0 2 5 9 3 6 0 0
P3 2 1 1 2 2 2 10 5 7 P3 0 1 1
P4 0 0 2 4 3 3 7 4 3 P4 4 3 1
Total Allocation 7 2 5
7 4 5


Example (Cont.)
The content of the matrix Need is defined to be Max – Allocation
Need
A B C
P0 7 4 3
P1 1 2 2
P2 6 0 0
P3 0 1 1
P4 4 3 1
The system is in a safe state since the sequence < P1, P3, P4, P2, P0 > satisfies safety criteria

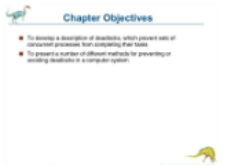

Example: P1 Request (1,0,2)
Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true
Allocation Need Available
A B C A B C A B C
P0 0 1 0 7 4 3 2 3 0
P1 3 0 1 0 2 0
P2 3 0 2 6 0 0
P3 2 1 1 0 1 1
P4 0 0 2 4 3 1
Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2 > satisfies safety requirement
Can request for (3,3,0) by P4 be granted?
Can request for (0,2,0) by P0 be granted?


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

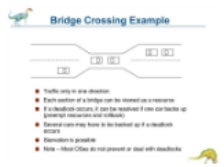

Recovery from Deadlock: Resource Preemption
Selecting a victim – minimize cost
Rollback – return to some safe state, restart process for that state
Starvation – same process may always be picked as victim, include number of rollback in cost factor

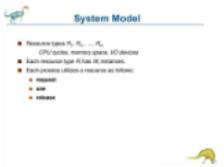
Deadlocks

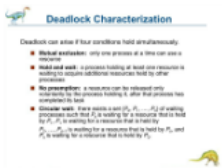

Chapter 7: Deadlocks
The Deadlock Problem
System Model
Deadlock Characterization
Deadlock Prevention
Deadlock Avoidance
Recovery from Deadlock

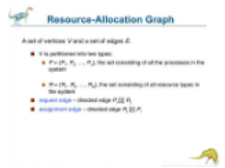

Chapter Objectives

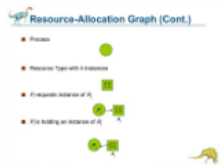

The Deadlock Problem

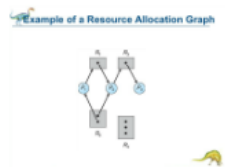

Bridge Crossing Example

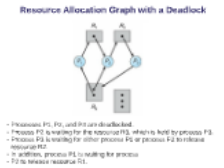

System Model

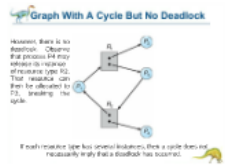

Deadlock Characterization


Resource-Allocation Graph


Resource-Allocation Graph (Cont.)


Example of a Resource Allocation Graph


Resource Allocation Graph with a Deadlock

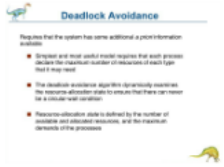

Graph With A Cycle But No Deadlock


Basic Facts

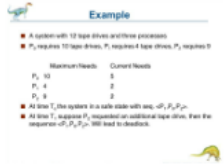

Methods for Handling Deadlocks

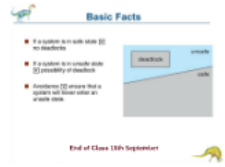

Deadlock Prevention


Deadlock Prevention (Cont.)

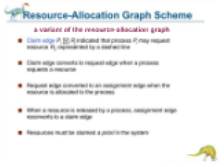

Deadlock Avoidance

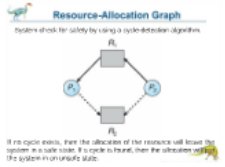

Safe State

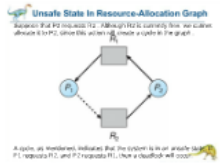

Example


Basic Facts

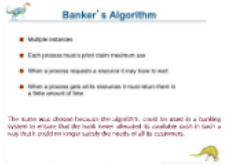

Avoidance algorithms

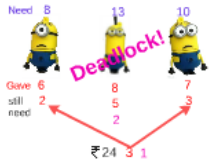

Resource-Allocation Graph Scheme


Resource-Allocation Graph

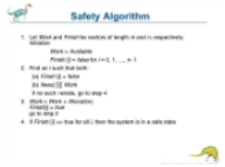

Unsafe State In Resource-Allocation Graph

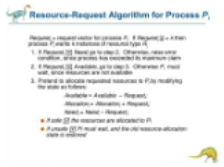

Resource-Allocation Graph Algorithm

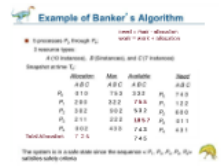

Banker's Algorithm

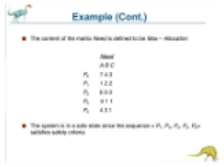

Need
Deadlock!
Give
still need

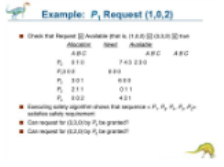

Data Structures for the Banker's Algorithm


Safety Algorithm


Resource-Request Algorithm for Process Pi


Example of Banker's Algorithm


Example (Cont.)


Example: P1 Request (1,0,2)


Recovery from Deadlock: Process Termination


Recovery from Deadlock: Resource Preemption

Chapter 7: Deadlocks

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

Chapter Objectives

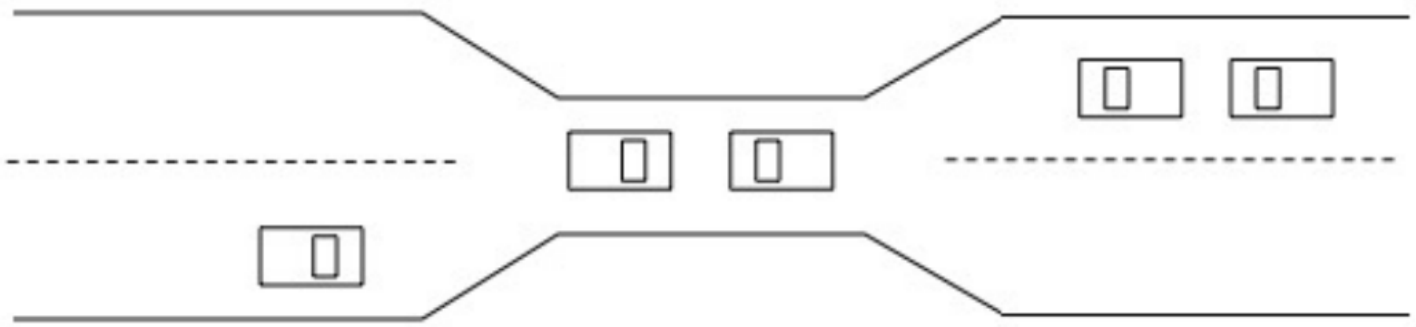
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

The Deadlock Problem

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

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- 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**

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 $\{P_0, P_1, \ldots, P_n\}$ 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

  $P_2, \ldots, P_{n-1}$ 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.

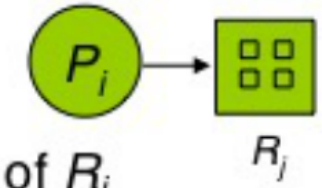
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge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

Resource-Allocation Graph (Cont.)

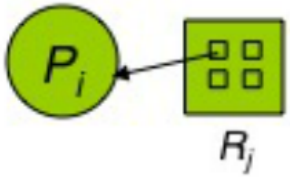
- Process

- Resource Type with 4 instances

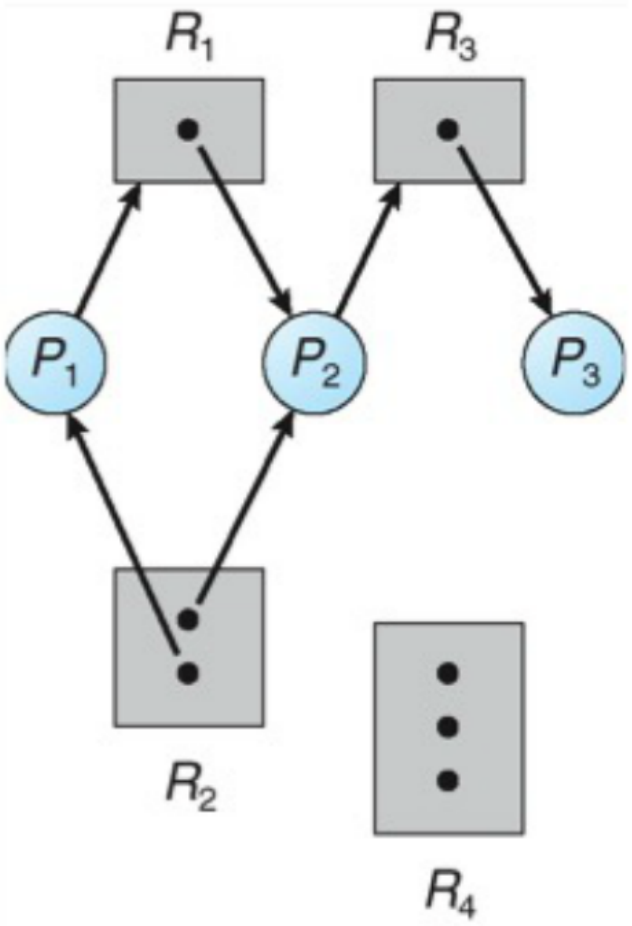
- P_i requests instance of R_j



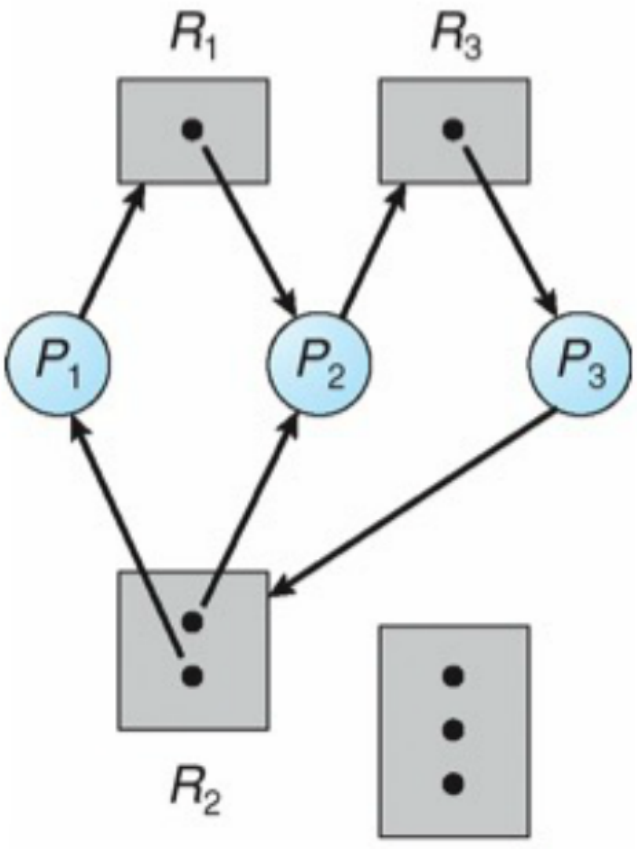
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



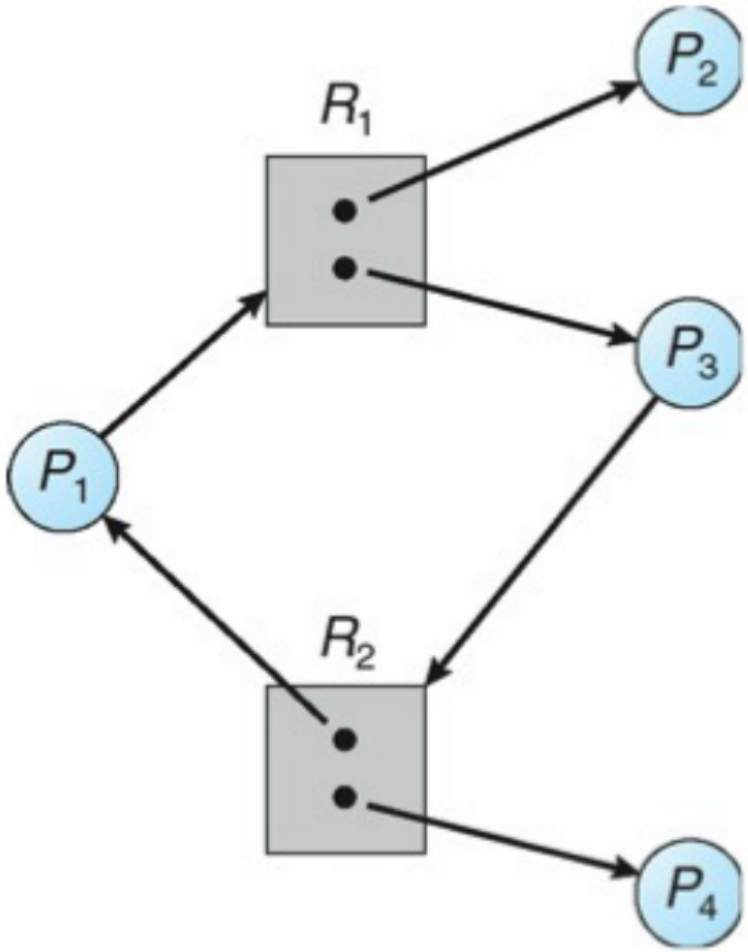
Resource Allocation Graph with a Deadlock



- Processes P1, P2, and P3 are deadlocked.
- Process P2 is waiting for the resource R3, which is held by process P3.
- Process P3 is waiting for either process P1 or process P2 to release resource R2.
- In addition, process P1 is waiting for process
- P2 to release resource R1.

Graph With A Cycle But No Deadlock

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.



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

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

Generally speaking, we can deal with the deadlock problem in one of three ways:

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

The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

Deadlock Prevention
Deadlock Avoidance

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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
  - 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 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

- 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 $<P_1, P_2, \ldots, P_n>$ of ALL the processes is 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

Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock.

Example

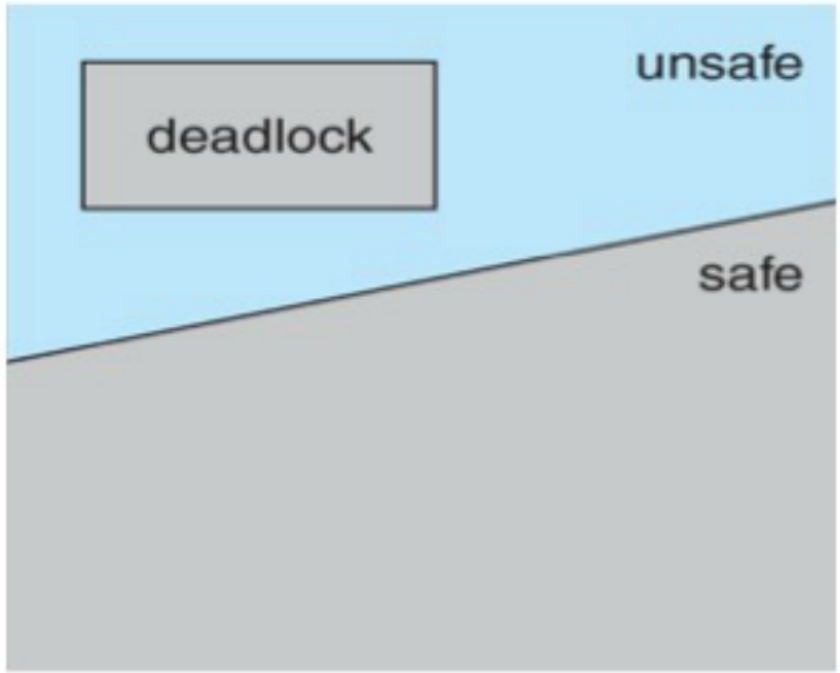
- A system with 12 tape drives and three processes
- P_0 requires 10 tape drives, P_1 requires 4 tape drives, P_2 requires 9

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time T_o the system in a safe state with seq. $<P_1,P_0,P_2>$.
- At time T_1 suppose P_2 requested an additional tape drive, then the sequence $<P_1,P_0,P_2>$. Will lead to deadlock.

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.



End of Class 15th September

Avoidance algorithms

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

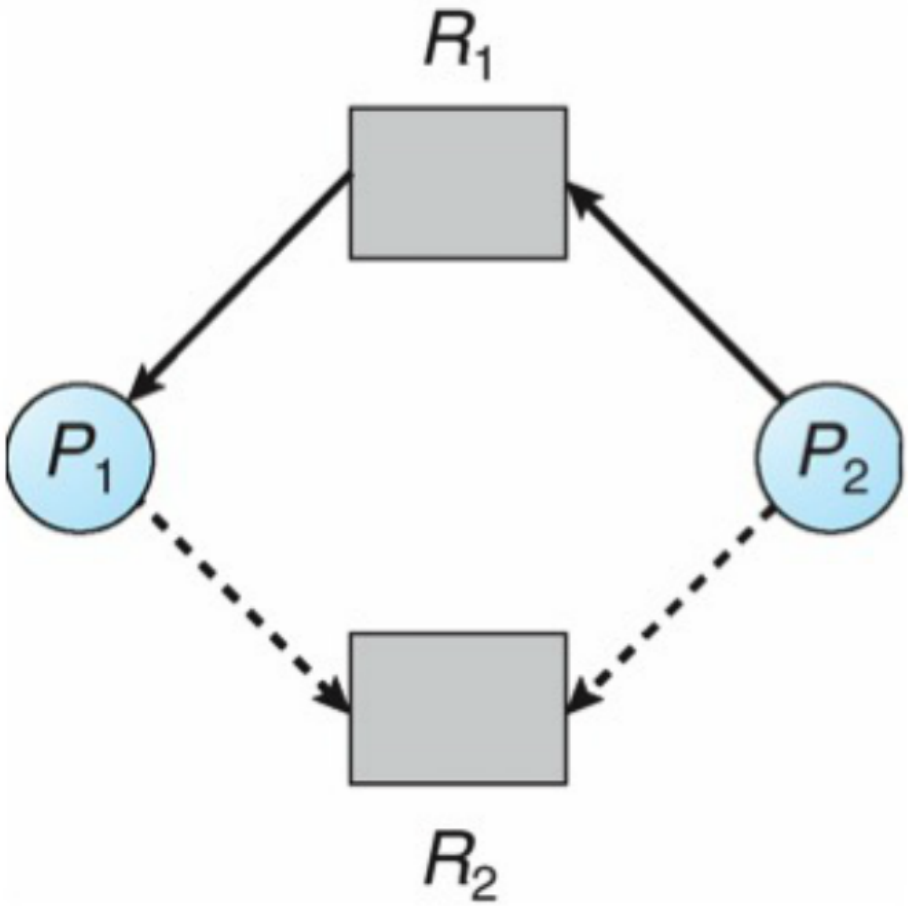
Resource-Allocation Graph Scheme

a variant of the resource-allocation graph

- Claim edge P_i ⚿ R_j indicated that process P_i may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge 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

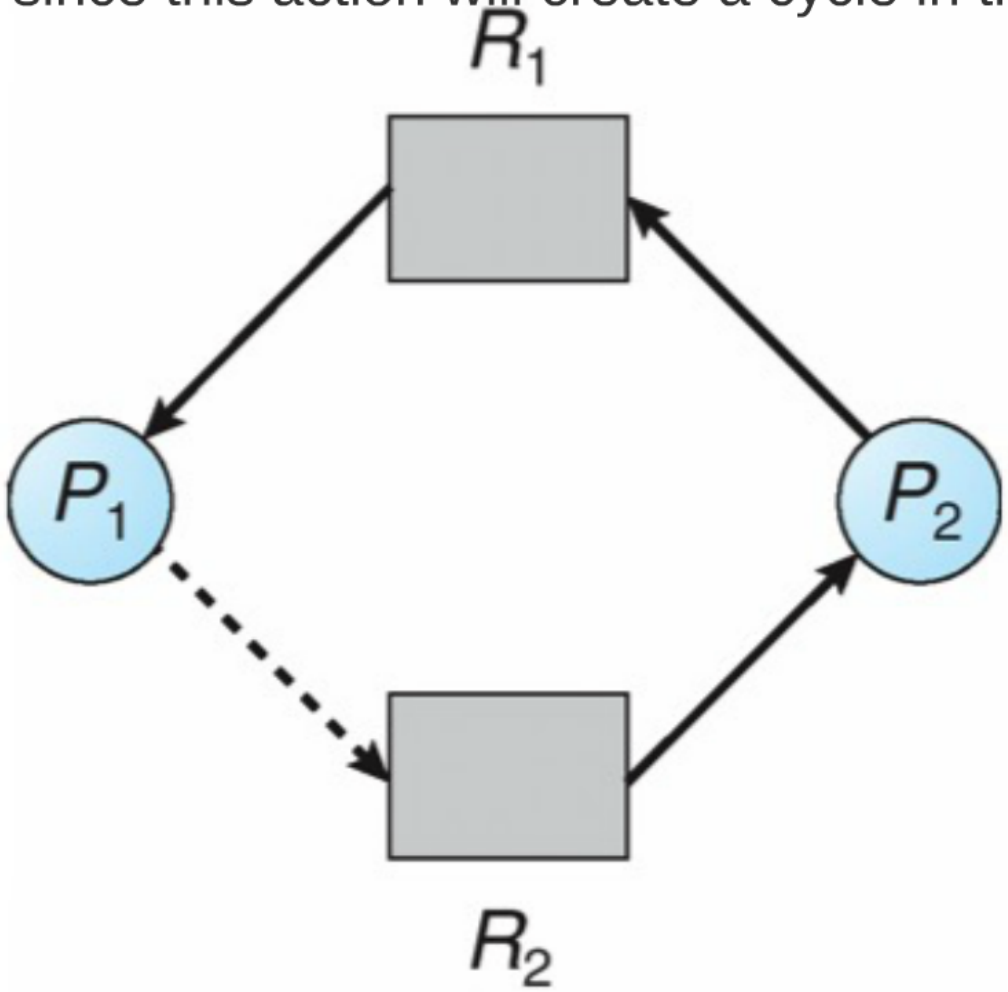
System check for safety by using a cycle-detection algorithm.



If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

Unsafe State In Resource-Allocation Graph

Suppose that P2 requests R2 . Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph .



A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

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

The name was chosen because the algorithm. could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Need 8 13 10

Deadlock!

Gave 6 8 7

still need 2 5 3

2

₹24 3 1

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, …, *n*- 1

2. Find an *i* such that both:

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

   (b) $Need_i$ ⊠ *Work*

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

3. *Work* = *Work* + $Allocation_i$
   Finish[*i*] = *true*
   go to step 2

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

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 ⇒ the resources are allocated to Pi*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

need = max - allocation
work = work + allocation

- 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*
	A B C	*A B C*	*A B C*		*A B C*
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2	**7 5 5**	P_1	1 2 2
P_2	3 0 2	9 0 2	**5 3 2**	P_2	6 0 0
P_3	2 1 1	2 2 2	**10 5 7**	P_3	0 1 1
P_4	0 0 2	4 3 3	**7 4 3**	P_4	4 3 1
Total Allocation	7 2 5		**7 4 5**		

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

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?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

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