

Deadlocks

Chapter 7: Deadlocks

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

Chapter Objectives

- To develop a description of deadlocks, with correct use of relevant process, and compute the state
- To present a number of different methods for preventing, avoiding, and recovering from a deadlock in a computer system

The Deadlock Problem

- A set of related processes each holding resources and waiting to acquire resources held by other processes in the set
- Examples:
 - System has 2 disk drives
 - P_1 and P_2 each hold one disk drive and each needs another one
- Example:
 - completes A and B, holds to 1
 - P_1 wants C
 - wants B

Bridge Crossing Example

- Trucks only in one direction
- Truck on a bridge can be moved on a resource
- A deadlock exists if you're not careful if one car backs up behind resources (not allowed)
- Resource cars may have to be backed up if a deadlock exists
- Deadlocks possible
- How - must either be not present or deal with deadlocks

System Model

- Process types P_1, P_2, \dots, P_n
 - 0/1/many, never zero, 0/many
- Each resource type has M_i instances
- Each process holds a resource or resources:
 - 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 instance of a resource will not release it until it has released all other instances
- No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed execution
- Circular wait:** there exists a set $\{P_1, P_2, \dots, P_n\}$ of waiting processes such that P_i is waiting for a resource held by P_{i+1} (or P_1 waiting for a resource held by P_n), and P_{i+1} is waiting for a resource held by P_i , and P_n is waiting for a resource held by P_{n-1}

Resource-Allocation Graph

Kind of resource: R is a set of edges: E

- R is partitioned into two types:
 - P_i is a process
 - R_i is a resource type
 - $R_i = P_i$ is a resource instance
- P_i requests a resource: R_i is a resource instance
- P_i releases a resource: R_i is a resource instance
- P_i holds a resource: R_i is a resource instance
- P_i is a resource instance

Resource-Allocation Graph (Cont.)

- Process
- Resource type with instances
- Request edge R_i
- Assignment edge A_j
- Hold edge H_k

Example of a Resource Allocation Graph

Resource Allocation Graph with a Deadlock

- Process P_1, P_2 hold no resources
- Process P_3 is waiting for the resource R_1 , which is held by process P_1
- Process P_4 is waiting for either process P_2 or process P_3 to release resource R_2
- P_3 is waiting for resource R_3
- P_4 is waiting for resource R_4
- P_2 is waiting for resource R_2

Graph With A Cycle But No Deadlock

However, there is no deadlock. Why not? Consider the resource-allocation graph. This graph has a cycle, but it is not a deadlock because the cycle consists of processes that are not waiting for resources held by other processes in the cycle.

Basic Facts

- If graph contains no cycle, no deadlock
- If graph contains a cycle, then deadlock
 - if any resource type has a request edge
 - if any resource type has an assignment edge
 - if any resource type has a hold edge
 - if any resource type has a request edge

Methods for Handling Deadlocks

Can't prevent, we can deal with the deadlock problem in one of three ways

- Prevention: ensure that the system will never enter a deadlock state
- Detection: allow the system to enter a deadlock state and then recover
- Recovery: ignore the problem and pretend that deadlocks never occur in the system (used by operating systems, including UNIX)

The best solution is to avoid deadlocks in most operating systems, including UNIX and Windows, via a technique called the banker's algorithm

Deadlock Prevention

Resource request can't be made

- Request for a resource:** not guaranteed for all resources, must first get the needed resources
- Hold and Wait:** not guaranteed that whenever a process requests a resource, it can hold out other resources
- No Preemption:** a process must hold all the resources it requests until the process has finished its execution
- Resource Allocation:** not guaranteed that a resource will be allocated to a process if it is requested, only if the process has not requested more resources

Deadlock Prevention (Cont.)

- No Preemption - if a process that is holding resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Request for a resource - if a process requests a resource, it must wait until all the resources it requests are available, as well as the resources that it is requesting
- Preemption - a process must hold all the resources it requests until the process has finished its execution, as well as the resources that it is requesting
- Resource Allocation - a process must hold all the resources it requests until the process has finished its execution, as well as the resources that it is requesting

Deadlock Avoidance

Requires that the system has some additional information

- System and user each must know the state of each process in the system (the maximum number of resources each can hold)
- The available resource information must be updated whenever a process requests a resource
- The available resource information must be updated whenever a process releases a resource
- The available resource information must be updated whenever a process holds a resource
- The available resource information must be updated whenever a process requests a resource
- The available resource information must be updated whenever a process releases a resource

Safe State

- System is in a safe state if there exists a sequence P_1, P_2, \dots, P_n of all the processes in the system such that for each P_i , the resources that P_i can still request can be satisfied by resources currently allocated to P_i and P_j , for $j < i$
- If a process P_i requests a resource that it cannot be allocated, then P_i can wait until it is finished
- If a process P_i requests a resource that it cannot be allocated, then P_i can wait until it is finished
- If a process P_i requests a resource that it cannot be allocated, then P_i can wait until it is finished
- If a process P_i requests a resource that it cannot be allocated, then P_i can wait until it is finished

Example

System with 12 tape drives and five processes

P_1 requires 10 tape drives, P_2 requires 4 tape drives, P_3 requires 2 tape drives

Maximum/Needs	Current Needs
$P_1 = 10$	$P_1 = 4$
$P_2 = 4$	$P_2 = 2$
$P_3 = 2$	$P_3 = 2$

All are T_i , the system is in a safe state with seq. P_3, P_2, P_1

Basic Facts

- A system is in a safe state if there exists a sequence P_1, P_2, \dots, P_n of all the processes in the system such that for each P_i , the resources that P_i can still request can be satisfied by resources currently allocated to P_i and P_j , for $j < i$
- If a process P_i requests a resource that it cannot be allocated, then P_i can wait until it is finished
- If a process P_i requests a resource that it cannot be allocated, then P_i can wait until it is finished

Avoidance algorithms

- Single instance of a resource type
- Two or more resource allocation graphs
- Multiple instances of a resource type
- Use the banker's algorithm

Resource-Allocation Graph Scheme

- Each edge R_i is a resource instance that process P_j may request
- Each edge A_j is a resource instance that process P_j may request
- Request edge connects to an assignment edge when the resource is allocated to the process
- Hold edge connects to a process, assignment edge connects to a process
- Release edge connects to a process, assignment edge connects to a process

Resource-Allocation Graph

System is safe for safety by using a cycle-detection algorithm

Unsafe State in Resource-Allocation Graph

System is in an unsafe state if there exists a sequence P_1, P_2, \dots, P_n of all the processes in the system such that for each P_i , the resources that P_i can still request can be satisfied by resources currently allocated to P_i and P_j , for $j < i$

Resource-Allocation Graph Algorithm

- Request for a resource
- Request for a resource
- Request for a resource

Banker's Algorithm

- Multiple instances
- Each process must not hold more than one instance
- When process requests a resource it may be held
- When process gets all its resources it must have a finite amount of time



Data Structures for the Banker's Algorithm

- Need: $Need_i = Max_i - Allocation_i$
- Available: $Available = Total - Allocation - Request$
- Request: $Request_i = Allocation_i - Request_i$
- Allocation: $Allocation_i = Request_i - Allocation_i$

Safety Algorithm

- Find a process P_i such that its need can be satisfied by the resources currently allocated to P_i and the resources held by other processes in the system
- Mark P_i as finished
- Repeat until all processes are finished or no process can be finished
- If all processes are finished, the system is in a safe state

Resource-Request Algorithm for Process P_i

- Request for a resource
- Request for a resource
- Request for a resource

Example of Banker's Algorithm

System P_1 through P_5

Process	Max	Allocation	Need
P_1	100	10	90
P_2	50	5	45
P_3	30	3	27
P_4	20	2	18
P_5	10	1	9

Example (Cont.)

- The system is in a safe state because the resources are P_1, P_2, P_3, P_4, P_5 for safety policy

Example: P_1 Request (1,0,2)

- Request for a resource
- Request for a resource
- Request for a resource

Recovery from Deadlock: Process Termination

- Identify the deadlocked processes
- Identify the processes that are not deadlocked
- Identify the processes that are not deadlocked

Recovery from Deadlock: Resource Preemption

- Select a victim - lowest cost
- Release - return its resources to the system
- Reclaim - return its resources to the system

Deadlocks

Chapter 7: Deadlocks

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

Chapter Objectives

- To identify 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 thread 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 holds 1 just hold one disk drive and each needs another one
 - Example:
 - Initial state: A and B released to P
 - P_1 needs A, wants B
 - P_2 wants A, needs B

Bridge Crossing Example

- Only one in one direction
- Each section of a bridge can be treated as a resource
- If a deadlock occurs, it can be resolved if one can backtrack to former resources and reassign
- Deadlocks may have to be treated as if deadlocks occur
- Deadlocks are possible
- How - backtracking is in progress or deal with deadlocks

System Model

- Resources R_1, R_2, \dots, R_n
- CPU units, memory spaces, IO channels
- Each resource type R_i has M_i instances
- Each process obtains a resource as follows:
 - request
 - use
 - release

Resource Allocation Graph with a Deadlock

- Processes P_1, P_2, P_3 and 1 unit each of resource R_1
- Process P_3 is waiting for the resource R_2 which is held by process P_1
- P_1 holds R_1 and holding R_2 other process P_2 or process P_2 hold resource R_2
- In addition, process P_1 is holding the process
- P_2 to release resource R_1

Deadlock Characterization

Deadlocks can arise if four conditions hold simultaneously:

- Mutual exclusion:** only one process at a time can own a resource
- Hold and wait:** a process holding at least one resource is waiting to obtain additional resources held by other processes
- No preemption:** a resource can be released only to the process holding it, after that process has completed its task
- Circular wait:** a set of processes $\{P_1, \dots, P_n\}$ of holding resources such that P_i is waiting for a resource that P_{i+1} is holding, for holding that resource P_{i+1} is waiting for a resource that is held by P_1 , and P_n is waiting for a resource that is held by P_n .

Resource-Allocation Graph

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

- Types of vertices:
 - P_1 processes in the system
 - R_1, R_2, \dots, R_n the set consisting of all the processes in the system
 - R_1, R_2, \dots, R_n the set consisting of all resource types in the system
 - request edge - directed edge $P_i \rightarrow R_j$
 - assignment edge - undirected edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

- Process
- Resource types with a process
- Request resource of R_j
- File holding an instance of R_j

Methods for Handling Deadlocks

Concepts involving: we can deal with the deadlock problem in one of three ways:

- Ignore the problem altogether
- Avoid the problem
- Prevent the problem
- Recovery from deadlocks

The Fair job line is not used by most operating systems, including Linux and Windows. It is there for the speculative developer to write programs that handle deadlocks.

Graph With A Cycle But No Deadlock

However, every cycle in a Resource-Allocation Graph (RAG) is not a cycle in the resource-allocation graph. For each resource type, the cycle then is feasible in the graph if a deadlock has occurred.

Basic Facts

- A graph contains cycles
 - Each resource type has M_i instances
 - Only one instance per resource type, then deadlocks are possible
 - Every resource type has M_i instances

Safe State

- When a system requests an available resource, system must decide if it can be allocated based on the system in a safe state
- System is in a safe state if there exists a sequence P_1, P_2, \dots, P_n of all the processes in the system such that for each P_i , the resources held by P_i can be released and can be obtained by sequentially executing P_i in order (in the order of P_i with $i < n$)
- This is:
 - IO resource banks can be continuously available, then P_i can wait and all P_i can finish
 - When P_i finished, P_j can obtain needed resources, obtain needed allocated resources, and terminate
 - When P_i terminates, P_j can obtain to needed resources, and so on
- Not all cyclic states are deadlocks. However, an arbitrary cyclic state is a deadlocks

Safe State

A system with 10 tape drives and five processes:

- P_1 requires 10 tape drives, P_2 requires 6 tape drives, P_3 requires 5 tape drives, P_4 requires 4 tape drives, P_5 requires 3 tape drives.

Current state:

Maximum Needs	Current Needs
$P_1: 10$	8
$P_2: 6$	2
$P_3: 5$	4
$P_4: 4$	2
$P_5: 3$	2

At this time, the system is in a safe state with seq. P_4, P_5, P_2, P_3 .

If all five T_i request P_i , a non-satisfiable additional tape drive, then the sequence P_4, P_5, P_2, P_3, P_1 will lead to deadlocks.

Basic Facts

- A system is in a safe state if:
 - There is a sequence of processes P_1, P_2, \dots, P_n such that for each P_i , the resources held by P_i can be released and can be obtained by sequentially executing P_i in order (in the order of P_i with $i < n$)

Resource-Allocation Graph Scheme

- A variation of the resource-allocation graph
- Each request edge is labeled with the resource R_j requested by a process
- Each assignment edge is labeled with the resource R_j assigned to a process
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource released by a process, assignment edge converted to a request edge
- Resources that are not owned by a process in the system

Resource-Allocation Graph

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

Unsafe State in Resource-Allocation Graph

System not in a safe state if there exists a cycle in the graph.

Resource-Request Algorithm for Process P_i

- Request - request units for process P_i if Requested > Available
- Check - check if the system is in a safe state
- Assign - assign resources to process P_i
- Release - release resources to process P_i
- Wait - wait for resources to be released

Banker's Algorithm

- Multiple instances
- Each process has a prior 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 fixed amount of time

Example of Banker's Algorithm

Example: P_1 through P_5 and 10 instances of resources A, B, and C.

Instance name	A	B	Available	Max	Allocation	Need
P_1	4/10	4/10	4/10	4/10	0/0	0/0
P_2	5/10	7/10	2/2	5/10	5/10	5/10
P_3	2/4	3/8	3/4	3/4	0/0	0/0
P_4	1/2	2/2	2/2	1/2	1/2	1/2
P_5	1/2	2/2	2/2	1/2	1/2	1/2
P_6	1/1	2/2	4/4	1/1	0/0	0/0
P_7	1/1	2/2	4/4	1/1	0/0	0/0
Total Available	7/14	7/14	7/14			

Example (Cont.)

- The content of the matrix below is extracted from the above
- The system is in a safe state since the sequence $P_1, P_5, P_4, P_7, P_6, P_2, P_3$ satisfies safety criteria

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort the process in a time and the deadlock cycle is eliminated
- Do a cycle check about all the resources that are held by the processes
- How long process has completed, and how much longer to complete
- How many processes has been used
- Resources process needs to complete
- How many processes will never be terminated
- Process terminate in batch

Recovery from Deadlock: Resource Preemption

- Selecting a victim - minimize cost
- Rollback - return to some safe state, re-allocate resources for that state
- Denial - some process may always be pre-empted, re-allocate resources to victim process

Banker's Algorithm

Need: 8, 13, 10

Have: 6, 8, 7

Still needed: 2, 5, 3

Goal: 24, 3, 1

Deadlock!



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

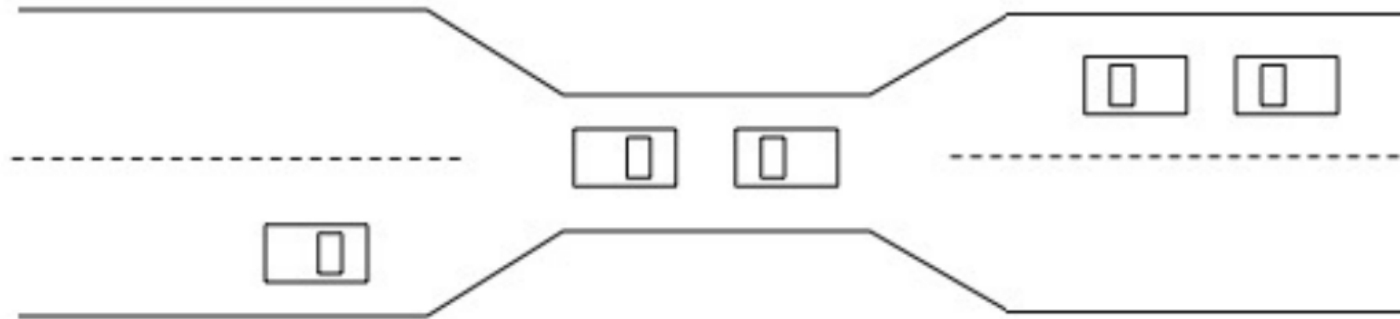
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
 - System has 2 disk drives
 - P_1 and P_2 each hold one disk drive and each needs another one
- Example
 - semaphores A and B , initialized to 1

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, \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 $P_i \xrightarrow{\square} R_j$
- **assignment edge** – directed edge $R_j \xrightarrow{\square} P_i$





Resource-Allocation Graph (Cont.)

- Process



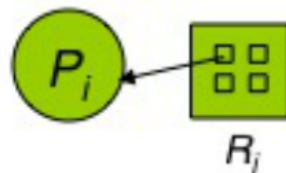
- Resource Type with 4 instances



- P_i requests instance of R_j

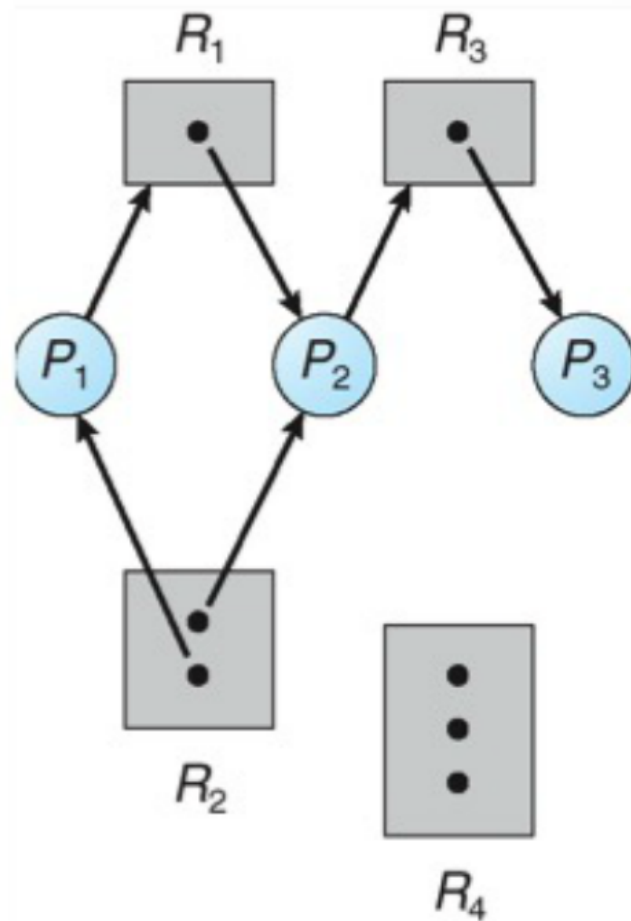


- P_i is holding an instance of R_j

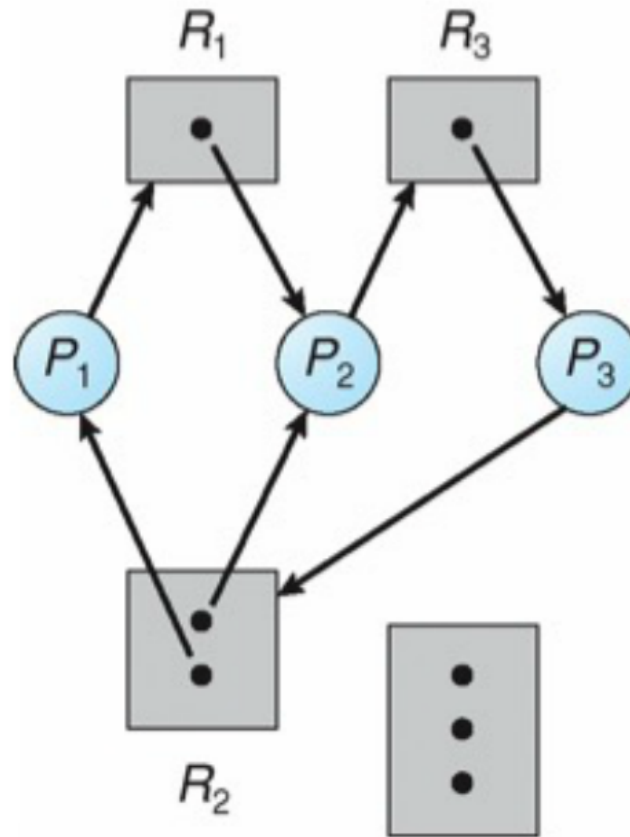




Example of a Resource Allocation Graph



Resource Allocation Graph with a Deadlock

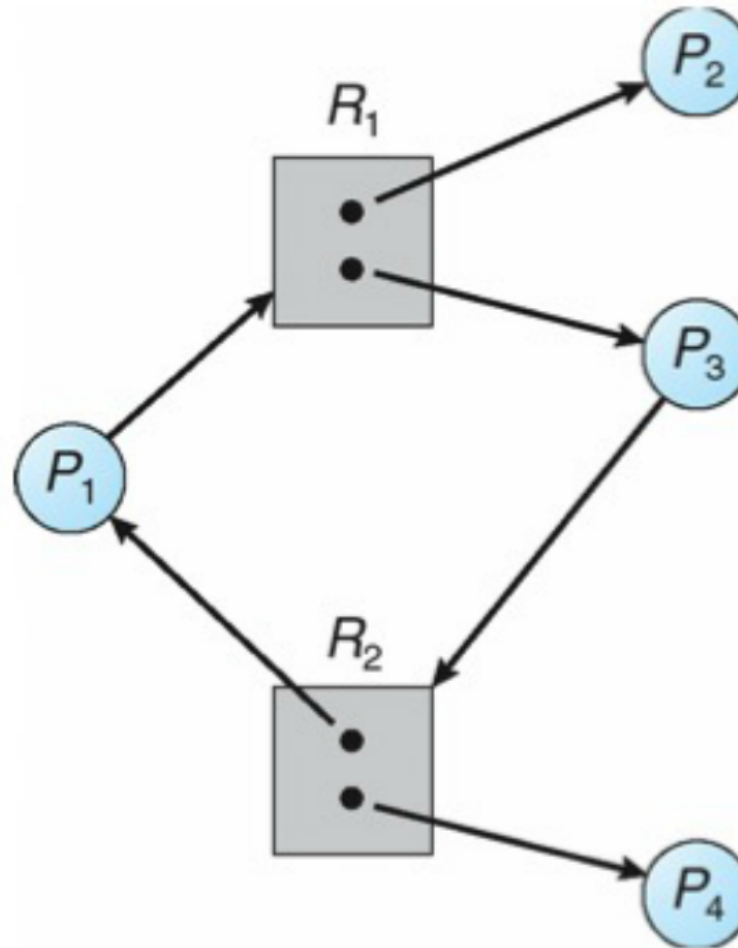


- Processes P_1 , P_2 , and P_3 are deadlocked.
- Process P_2 is waiting for the resource R_3 , which is held by process P_3 .
- Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2 .
- In addition, process P_1 is waiting for process P_2 to release resource R_1 .



Graph With A Cycle But No Deadlock

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2 . That resource can then be allocated to P_3 , 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 $\langle P_1, P_2, \dots, P_n \rangle$ 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

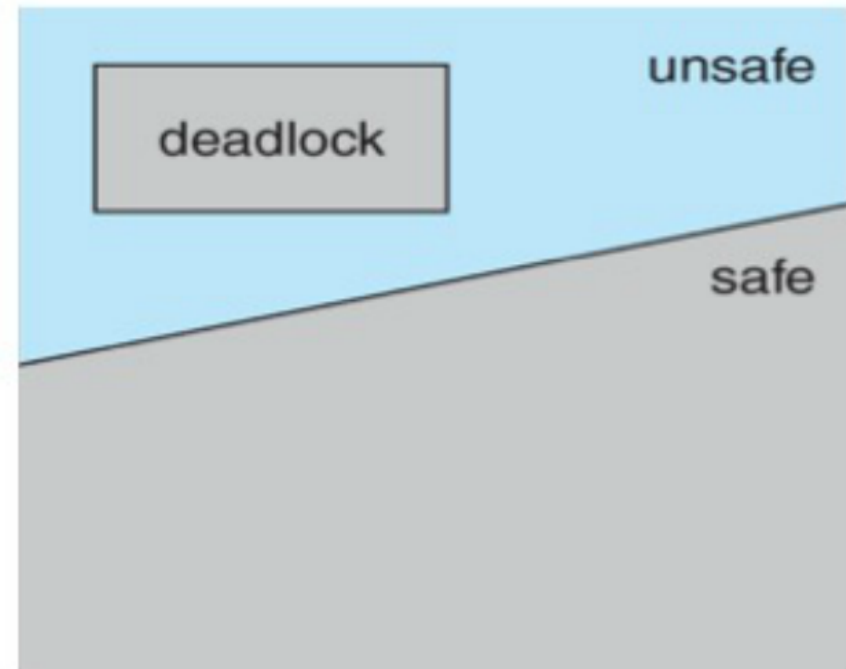
- At time T_0 the system in a safe state with seq. $\langle P_1, P_0, P_2 \rangle$.
- At time T_1 suppose P_2 requested an additional tape drive, then the sequence $\langle P_1, P_0, P_2 \rangle$. Will lead to deadlock.





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

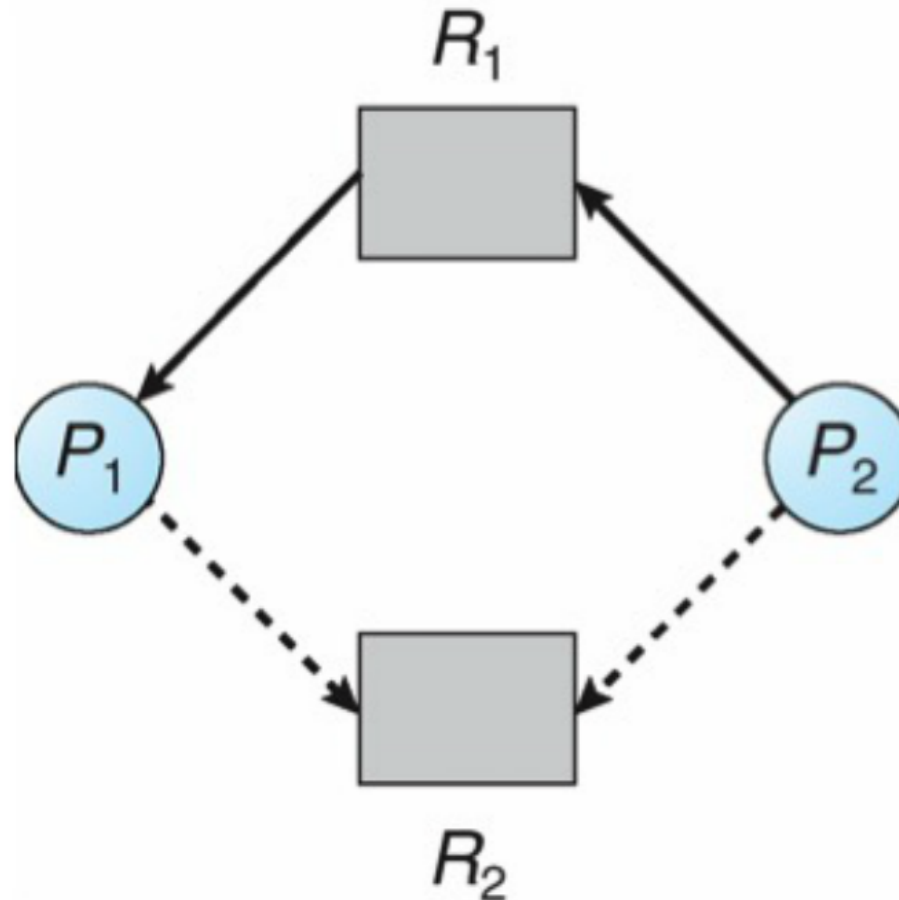
- **Claim edge** $P_i \text{ } \boxed{\times} \text{ } R_j$ indicated that process P_i may request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the 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.



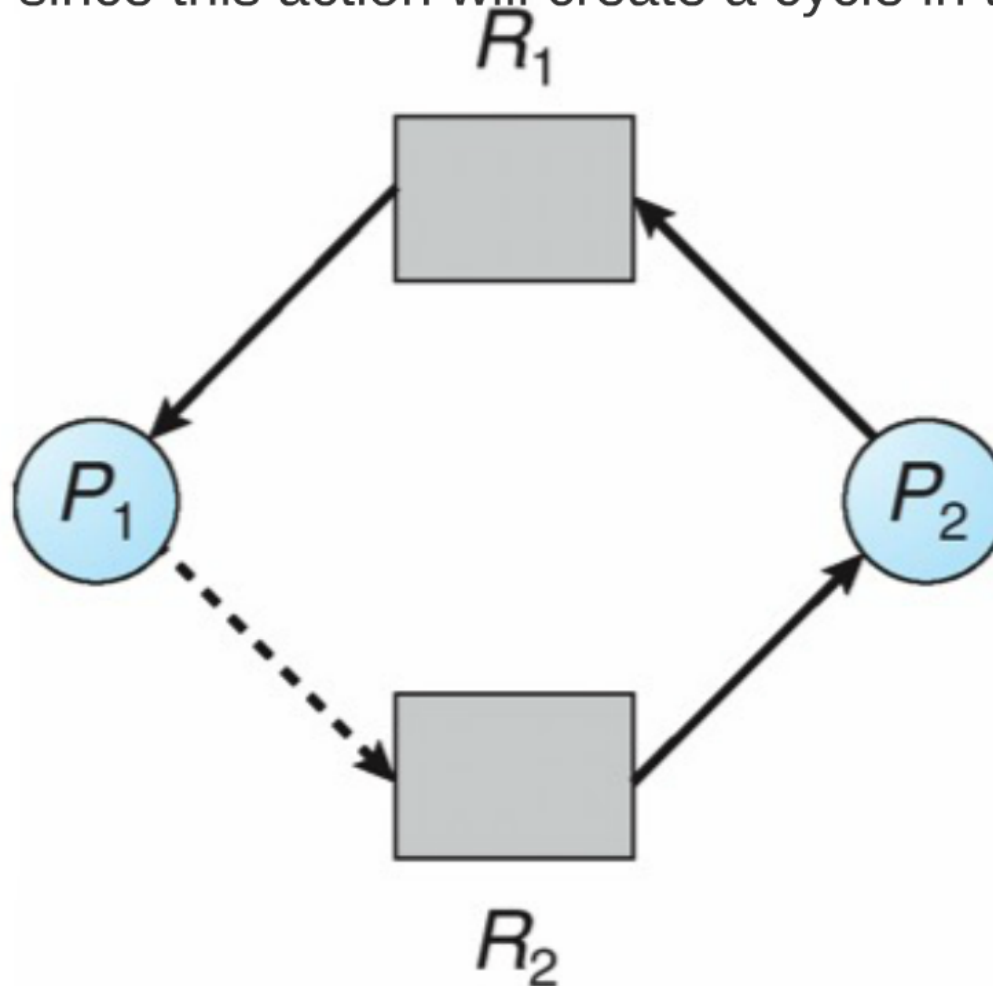
If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.





Unsafe State In Resource-Allocation Graph

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



A cycle, as mentioned, indicates that the system is in an unsafe state. If P_1 requests R_2 , and P_2 requests R_1 , 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

still 2

need

8

5

2

7

3

₹ 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) $Need_i \leq Work$If no such *i* exists, go to step 4
3. *Work* = *Work* + *Allocation*_{*i*}
Finish[*i*] = *true*
go to step 2
4. If *Finish* [*i*] == *true* for all *i*, then the system is in a safe state





Resource-Request Algorithm for Process P_i

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

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe \leq the resources are allocated to P_i
- If unsafe \leq P_i 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 (5 instances), and C (7 instances)

Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>			
	A	B	C	A	B	C	A	B	C	A	B	C	
P_0	0	1	0	7	5	3	3	3	2	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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 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

