Chapter 7: Deadlocks

Chapter 7: Deadlocks

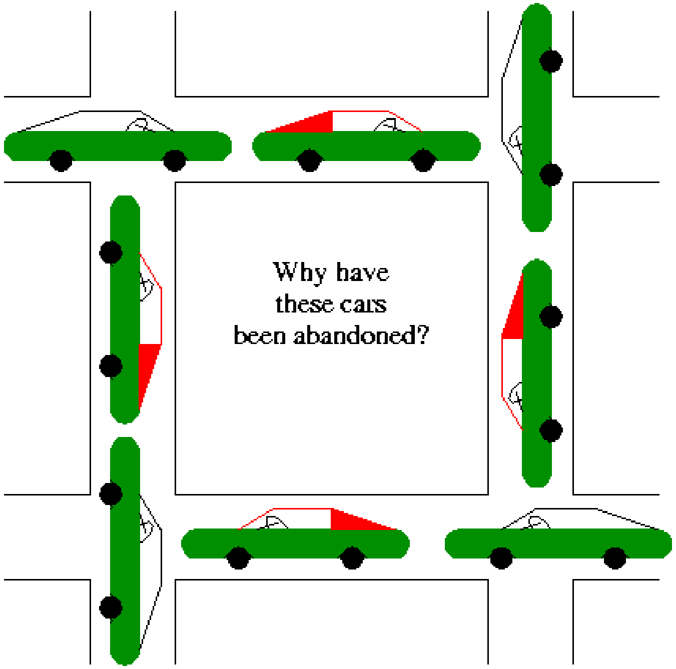
- **System Model**
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

- In mutliprogramming environment
  - Several processes compete for finite number of resources
  - Each process can request a resource and wait until it is available
- 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

What is a Deadlock?

- A **deadlock** occurs when a every member of a set of processes is waiting for an event that can only be caused by a member of the set.
- Consider the below figure:
- The processes are the cars.
- The resources are the spaces occupied by the cars



The Deadlock Problem (2)

- **Formal Definition:**
  - A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set of processes
  - Deadlock prevents sets of concurrent processes from completing their tasks

- **Example 1**
  - System has 2 disk drives
  - Process P_1 and process P_2 each hold one disk drive and each needs another one to complete their respective tasks.

- **Example 2**
  - In a database system, a program may have to lock several records it is using, to avoid race conditions. If process ***A*** locks **record *R1*** and process ***B*** locks **record *R2***, and then each process tries to lock the other one's record, we also have a deadlock.

System Model

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

  I/O devices (disk drives, printers etc.), files, database records, data, semaphore etc.
- Each resource type R_i has W_i instances/units.
  - If a system has two drives then resource type *Drives* has *two* instances/units
- Each process utilizes a resource as follows:
  - **request** ; a process must request first then wait until it can acquire resource
  - **use** ; the process operates on the resource
  - **Release** ; the process releases the resource
- Number of resources requested cannot exceed number of resources in system
- A set of processes is in a deadlocked state if each process in the set is waiting for an event that can be caused only by another process in the set
  - Processes never finish executing; preventing other processes from starting

Deadlock Characterization: Four Necessary Conditions

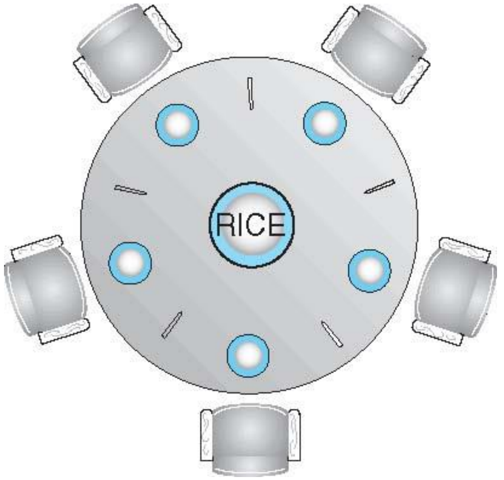
- Deadlock can arise if four conditions hold simultaneously.

    1. Mutual exclusion
    2. Hold and wait
    3. No preemption
    4. Circular wait

- All four conditions must hold for a deadlock to occur

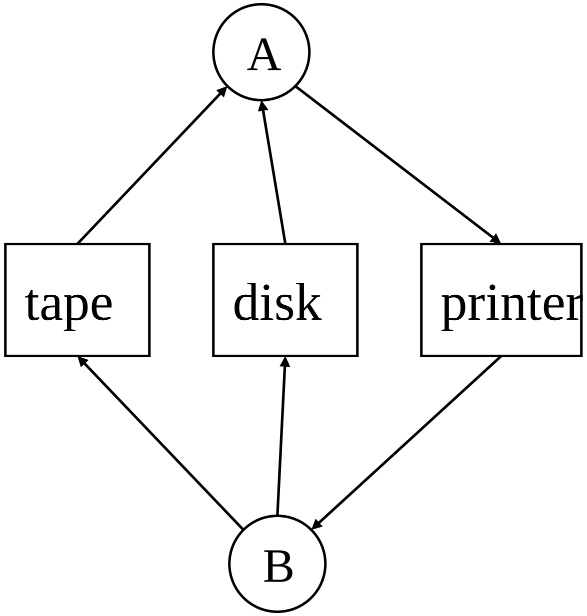
1. Mutual Exclusion

- **Mutual exclusion**: only one process at a time can use a resource ; that is, at least one resource must be held in non-sharable mode.
  - Requesting process must wait until resource is released
  - For example, in dining philosopher problem, a chopstick is a resource which can be used only by a single philosopher.



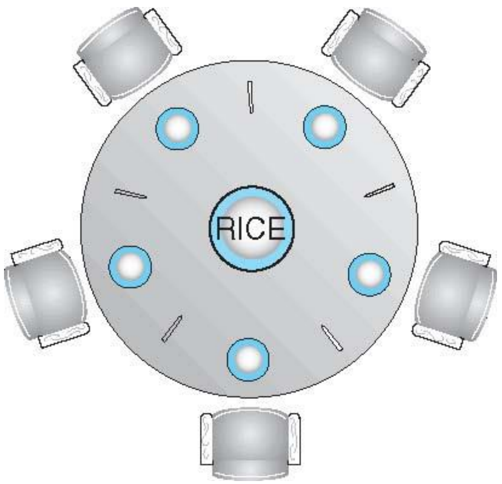
2. Hold and Wait

- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
  - For example, process ***A*** copies data from a tape drive to a file on disk, sorts the file and then prints. However, it has acquired only tape drive and disk drive and is waiting for printer which is held by Process B



3. No Preemption

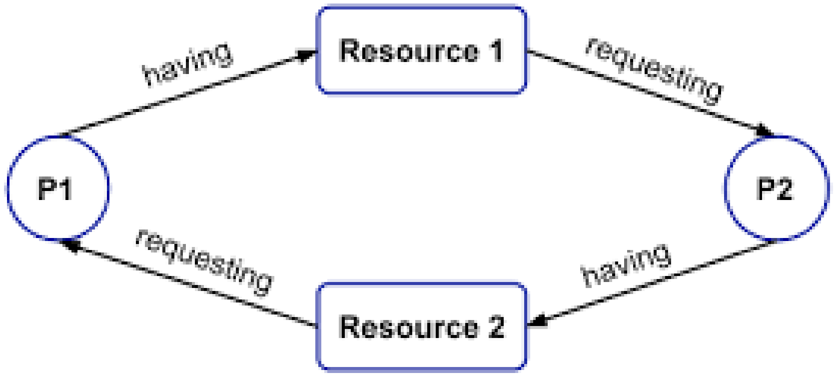
- **No preemption**: a resource can be released only voluntarily by the process holding it; that is, resources cannot be preempted. In other words, no resource can be forcibly removed from a process holding it



4. Circular Wait

•

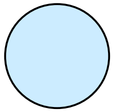
- **Circular wait**: there exists a set of waiting processes $\{P_0, P_1, \ldots, P_n\}$ such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



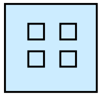
Resource-Allocation Graph

- ❑ Deadlock Description: Resource-Allocation Graph represents the state of a resource allocation system at a given moment in time.
- ❑ It is a directed ***graph*** consisting of vertices (Points) V and a set of edges (arcs) 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
- Process ; circles

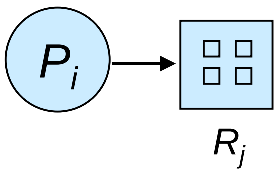


- Resource Type with 4 instances ; rectangles with a dot for each instance

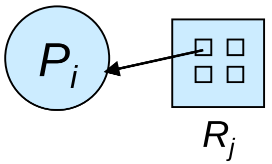


Resource-Allocation Graph (Cont.)

- **request edge** – directed edge $P_i \rightarrow R_j$
  - Process P_i has requested an instance of resource type R_j
  - And P_i is currently waiting for that resource
- Pi requests an of instance of R_j



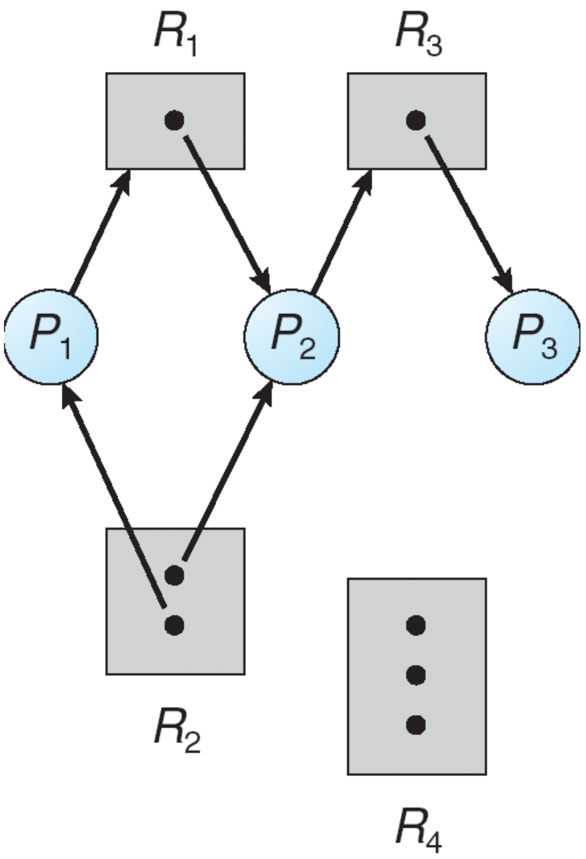
- **assignment edge** – directed edge $R_j \rightarrow P_i$
  - Instance of R_j has been allocated to process P_i
-
- P_i is **holding** an instance of R_j



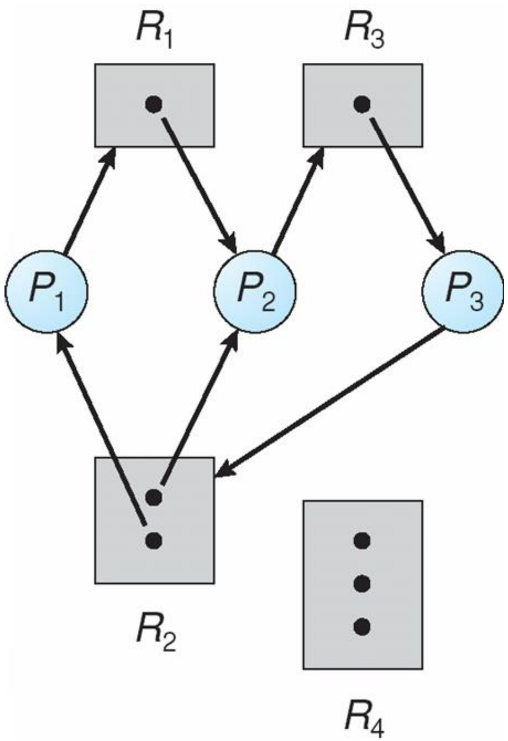
Example of a Resource Allocation Graph

- P_1 is **holding** an instance of R_2
- P_2 is **holding** an instance of R_1 and R_2 respectively.
- P_3 is **holding** an instance of R_3

- P_1 **requests** an of instance of R_1

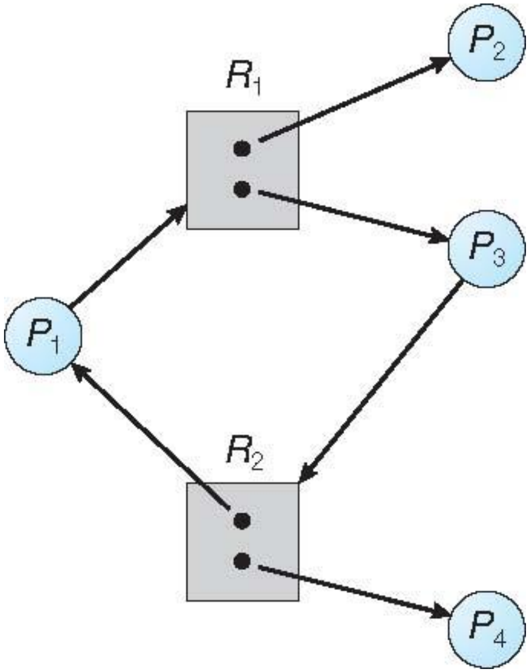


Resource Allocation Graph With A Deadlock



- **Deadlock:** P_2 is waiting for resource R_3 held by P_3,
  - P_3 is waiting for resource R_2 held by P_2
  - P_1 is waiting for resource R_1 held by P_2

Graph With A Cycle But No Deadlock



- **No deadlock:** since P_4 may release its instance of resource type R_2
  - which can be allocated to P_3; thus breaking the cycle

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$ deadlock may exist
  - if only one instance per resource type, then deadlock
    - Cycle is necessary and sufficient condition for deadlock
  - if several instances per resource type, possibility of deadlock
    - Cycle is necessary but not sufficient condition for deadlock

Chapter 7: Deadlocks

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

Methods for Handling Deadlocks

- ***Method 1:*** *Devise a protocol* ensuring that the system will ***never*** enter a deadlock state:
  - **Deadlock prevention scheme**
    - Ensure that at least one of the four necessary conditions cannot hold
    - Prevent deadlocks by constraining how requests can be made
  - **Deadlock avoidance**
    - OS uses additional knowledge:
      - Which resources a process will request in its lifetime
      - Currently available resources, and, currently allocated resources
      - Future release of each process
    - Then can decide If a process should wait, or, if a request can be satisfied
- ***Method 2:*** Allow the system to enter a deadlock state
  - System allows algorithm that examines the state of the system to determine if deadlock has occurred and an algorithm to recover from the deadlock
- ***Method 3:*** Ignore the problem and pretend that deadlocks never occur in the system;

Chapter 7: Deadlocks

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

Deadlock Prevention

- *Deadlock Prevention* **means making sure deadlocks never occur.**

- By ensuring that at least one of *four* conditions cannot hold, we can prevent the occurrence of a deadlock.

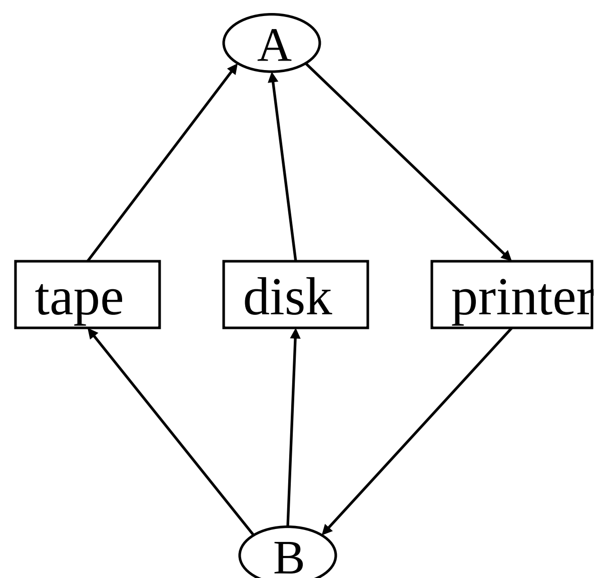
- **Eliminate Mutual Exclusion** – Make resources sharable (e.g., read-only files); But not all resources can be shared.
  - Sharable resources cannot be involved in a deadlock; wait never needed
  - But some resources are inherently non-sharable, e.g., mutex locks cannot be simultaneously shared by several process

Deadlock Prevention (**Eliminate Hold and Wait**)

- **Eliminate Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
- Removing Hold-and-Wait implies every process must acquire all resources it needs all at once.
  - **Protocol 1:** Require process to request and be allocated all its resources before it begins execution, *For example, a process copies data from a DVD drive to a file on disk, sorts the file and then prints.* According to protocol 1, process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution even though it needs printer only at the end
  - **Protocol 2:** Allow process to request resources only when the process has none allocated to it. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated. Using the same **example above**, this method requests only DVD and disk file initially, copies from DVD to disk and then release both. The process must then request the disk file and printer. And must release both after copying file from disk to printer and terminate.
  - The problem here is Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

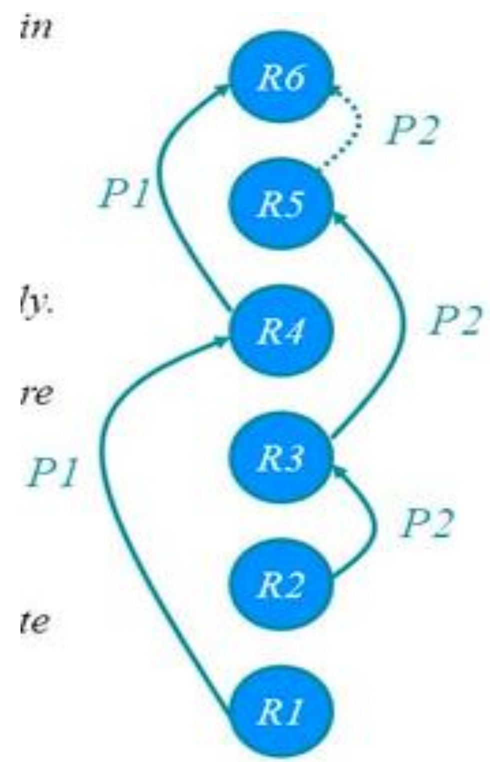
- **Eliminate *No Preemption*** –
- That means being able to forcibly take resources away from a process.
- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released ***(Preempted)*** and are allocated to another requesting process.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting



Deadlock Prevention (Cont.)

- **Eliminate Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - Request resources according to the decided order or request
- Define a linear ordering of resource types
- Let R ={R_1,R_2,R_3,..R_m} be the set of resource types. Assign each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our example: For example
  - Tape drive = 1
  - Disk drive = 4
  - Printer = 6

- If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
- Processes can request resources whenever they want to, but all requests must be made in numerical order.



Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Chapter 7: Deadlocks

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

Deadlock Avoidance

- ❑ Requires that the system has some additional information available in advance

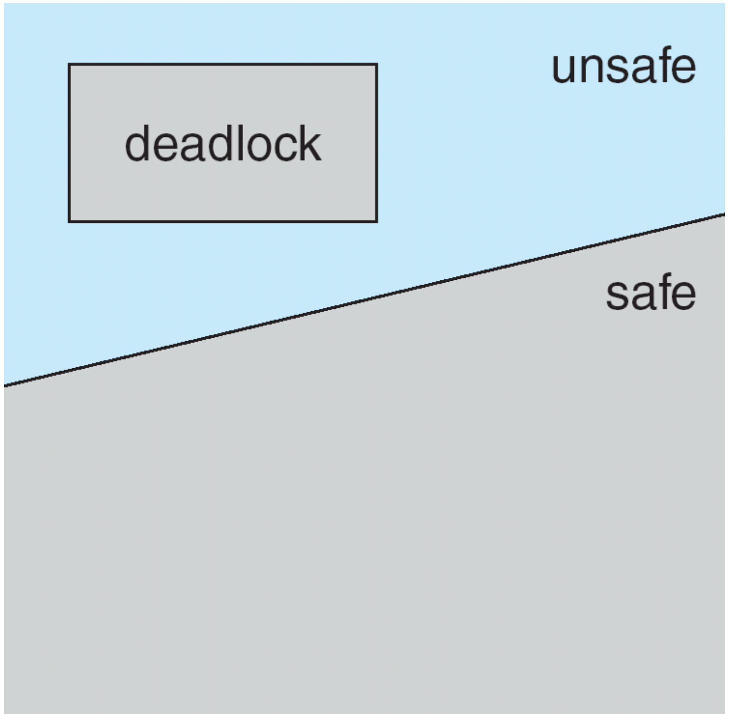
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The ***deadlock-avoidance algorithm*** dynamically examines the ***resource-allocation state*** to ensure that there can never be a ***circular-wait condition***
- ***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 in the systems such that for each process 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 process 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
- According to **Deitel:**
  - *"An unsafe state does not imply the existence of deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to deadlock."*
- **Avoidance** ⇒ ensure that a system will never enter an unsafe state.



First Example: Safe State

- System with ***24 tape drives*** and three processes
- P_0 may require 20 tape-drives during execution, P_1 may require 8, and P_2 may require up to 18.
- Assume, P_0 is holding 10 tape drives, P_1 holding 5 and P_2 holding 4 tape drives.
- The remaining ***available*** resources are: 24-19=5.
- P_1 needs 3 more tape drives to finish, so it can be allocated 3 drives.
- Once P_1 is done, it will return all 8 resources, making the total available resources = 10.
- P_0 may proceed further obtaining the available 10 tape drives, once it completes it will return all 20 tape drives → available = 20.
- P_2 may proceed finally, and obtain 14 tape drives out of 20 available.
- The system is said to be in safe state, since there is a safe sequence that avoids the deadlock. $\langle P_1, P_0, P_2 \rangle$

Process	**Max. Need**	**Allocated**	**Need**
P_0	20	10	10
P_1	8	5	3
P_2	18	4	14

Second Example: Safe State

- Again, consider system with ***12 tape drives*** and three processes
- The process, however, have different maximum needs and allocated resources.
- Following the same procedure as in the previous slide.
- The available resources are: 3
- It will be safe to first allocate resources to P_1
- After P_1 completes, 5 resources are allocated to P_0 and then finally P_2 may obtain the available resources.
- So safe sequence is: < P_1, P_0, P_2>

Process	**Max. Need**	**Allocated**
P_0	10	5
P_1	4	2
P_2	9	2

Second Example: UnSafe State

- Consider the system with ***12 tape drives*** and three processes from the previous example:
- **What if P_2 requests and is allotted one tape drive?**

Process	Max. Need	Allocated	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3 (instead of 2)	6

- The available resources with system = 2 tape drive.
- The system is no longer in safe state. Only P_1, can be allocated all its tape drives, when it is done it will return them and ***the available*** tapes will be four.
- Both P_0 ($need_0$ = 5) and P_2 ($need_2$ = 5), need more than the available four, if they request , they will have to wait which may result in deadlock.
- It was a ***mistake*** to grant one or more tape drives to P_2
- It would have been safe if we had made P_2 wait.

Deadlock Avoidance Algorithms

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

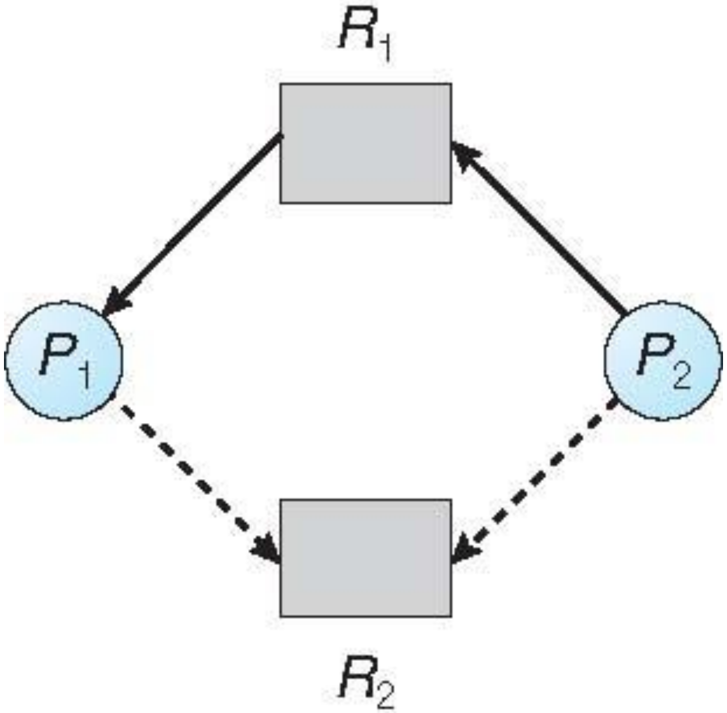
- **Multiple instances** of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- **Claim edge** P_i - - - > 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, $P_i \rightarrow R_j$
- **Request edge** converted to an **assignment edge** when the resource is allocated to the process $R_j \rightarrow P_i$
- When a resource is released by a process, **assignment edge** reconverts to a **claim edge** P_i - - - > R_j
- Resources must be claimed *a* ***priori*** in the system

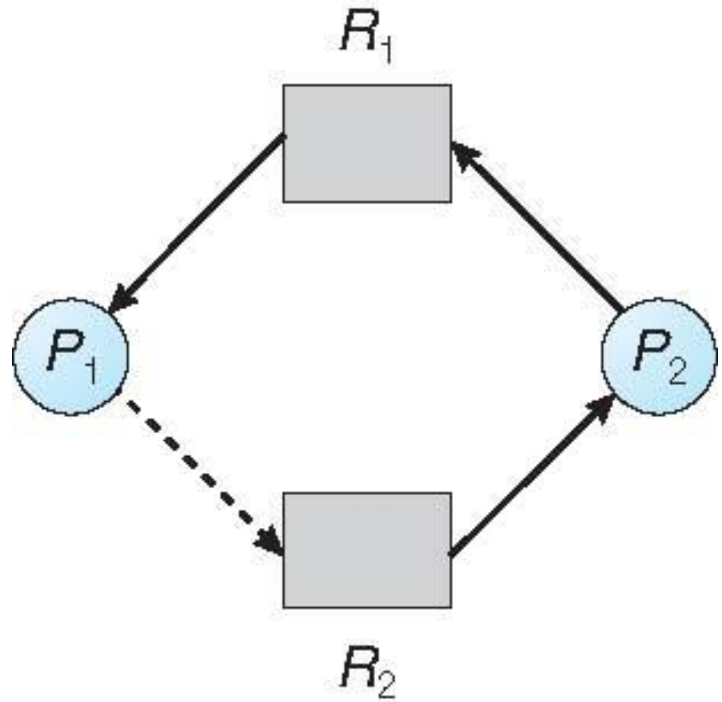
Resource-Allocation Graph with claim edges

- R_1 is held by P_1.
- P_2 requests R_1.
- P_1 and P_2 may need R_2 and therefore have claimed R_2. (Made a ***priori*** information available to the system)



Unsafe State In Resource-Allocation Graph

- Now suppose that process P_2 requests a resource R_2
- 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



P_2 must wait! A cycle found.

Banker’s Algorithm

- Banker’s algorithm is used to avoid deadlock in a system where the resources have multiple instances.
- **Key idea:** Ensure that the system of processes and resources is always in a safe state
- Multiple instances
- Each process must in advanced 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

Banker’s Algorithm

- The Banker's algorithm allows:
  - mutual exclusion
  - wait and hold
  - no preemption
- Prevents:
  - circular wait

- User process may only request one resource at a time.
- System grants request only if the request will result in a ***safe state***.

Banker’s Algorithm

- Banker’s algorithm comprises of ***Data Structures*** and two algorithms:
  - Safety algorithm
  - Resource request algorithm

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 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*]**

1. Safety Algorithm

1.Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. **Initialize:**

Work = Available *//The number of resources available with the system*

***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1** //unfinished processes

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

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

(b) $\boldsymbol{Need_i \le Work}$ *//resources needed by the process i are less than the available resources*

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

3. $\boldsymbol{Work = Work + Allocation_i}$ *//assume Pi is finished then return resources back to the system*

Finish*[*i*] = *true *//process is assumed to be finished.*

go to step 2

4.If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state. otherwise, the processes whose index is false may potentially be involved in a deadlock in the future.

Example of Banker's Algorithm (Safety Algorithm)

- 5 processes P_0 through P_4;

  3 resource types: *A* (10 instances), *B* (5 instances), and *C* (7 instances)

  :[10 5 7]

- Snapshot at time T_0:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Allocated = [7 2 5]
- **Find out the Need vector for each process for each of the resource?**

Example (Cont.)

- The content of the matrix ***Need[i,j]*** is defined to be ***Max[i,j] – Allocation[i,j]***

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Process	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

- Find out if the System is in safe state using the ***Safety Algorithm***.
  - The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria. [See the next slide for the step by step details of the algorithm]

Example (cont'd)

Step 1: *Work = Available* $\rightarrow$ *Work = [3 3 2]*

Step 2:

$Need_i \leq Work$*//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Let's start from P_0, $Need_0$ *<= Work?* ***No***

Process	**Need**			**Allocation**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0			
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

Safe Sequence: < >

Example (cont'd)

Work = [3 3 2]

How about P_1?

$Need_i \leq Work$//resources needed by the process i are less than the available resources

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

Safe Sequence: <P_1, >

Example (cont'd)

Work = [5 3 2]

For P_2? No

For P_3?

$Need_i \le Work$ //resources needed by the process i are less than the available resources

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

Safe Sequence: <P_1, P_3, >

Example (cont'd)

Work = [7 4 3]

For P_4?

$Need_i \leq Work$//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	**Need**			**Allocation**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2			

Safe Sequence: $<P_1, P_3, P_4, \quad >$

Example (cont'd)

Work = [7 4 5]

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	**Need**			**Allocation**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2	7	5	5

Safe Sequence: < P_1, P_3,P_4,P_0, >

Example (cont'd)

Work = [7 5 5]

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	**Need**			**Allocation**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2	7	5	5

Safe Sequence: $< P_1, P_3, P_4, P_0, P_2 >$

Example: Execution of Safety algorithm on the above data structures to find if system is safe.

Step 1 of Safety Algo

m=3, n=5

Work = Available

Work = | 3 | 3 | 2 |

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

Step 2

For i = 0

$Need_0$ = 7, 4, 3

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

So P_0 must wait

But Need ≤ Work

Step 2

For i = 1

$Need_1$ = 1, 2, 2

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

So P_1 must be kept in safe sequence

Step 3

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

	A	B	C
Work =	5	3	2

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

Step 2

For i = 2

$Need_2$ = 6 , 0, 0

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

So P_2 must wait

Step 2

For i=3

$Need_3$ = 0, 1, 1

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

So P_3 must be kept in safe sequence

Step 3

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

	A	B	C
Work =	7	4	3

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

Step 2

For i = 4

$Need_4$= 4, 3, 1

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

So P_4 must be kept in safe sequence

Step 3

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

	A	B	C
Work =	7	4	5

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

Step 2

For i = 0

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need < Work$ (7, 4, 3 < 7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

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

	A	B	C
Work =	7	5	5

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

Step 2

For i = 2

$Need_2$ = 6 , 0, 0

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

So P_2 must be kept in safe sequence

Step 3

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

	A	B	C
Work =	10	5	7

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

Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

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

2. 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 (*tentatively*) allocate requested resources to ***P$_i$*** by modifying the state as follows:

   Available = Available – Request$_i$; *//subtract from available*

   Allocation$_i$ = Allocation$_i$ + Request$_i$; *//Allocate to Process i*

   Need$_i$ = Need$_i$ – Request$_i$;

- If ***safe*** (execute safety algorithm) $\Rightarrow$ if a safe sequence is found then resources are allocated to ***P$_i$***
- If ***unsafe*** $\Rightarrow$ ***P$_i$*** must wait. and the old resource-allocation state is restored. The requested resources are not allocated to ***P$_i$***.
- .

Example: P_1 Request (1,0,2)

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Process	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

A B C

$Request_1$ = 1, 0, 2

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

Step 1

1, 0, 2 1, 2, 2

$Request_1 < Need_1$

Step 2

1, 0, 2 3, 3, 2

$Request_1 < Available$

Step 3

$Available = Available - Request_1$ [3 3 2] – [1 0 2]= [2 3 0]

$Allocation_1 = Allocation_1 + Request_1$ [[2 0 0] + [1 0 2]= [3 0 2]

$Need_1 = Need_1 - Request_1$ [1 2 2] - [1 0 2]= [0 2 0]

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	

Execution of Safety algorithm after resource request algorithm to find if new system state is safe.

Using below steps of Safety algorithm it can be found if there exists a safe sequence.

Work = available → *Work = [2 3 0]*

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

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			

- Safe Sequence : < P1 ,P3 , P4 ,P0 ,P2 >
- Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement. So Yes, P1’s request may be granted immediately.

Execution of Safety algorithm after request is granted to find if new system state is safe.

Step 1 of Safety Algo

m=3, n=5

Work = Available

Work =

2	3	0

Finish =

0	1	2	3	4
false	false	false	false	false

Step 2

For i = 0

$Need_0$ = 7, 4, 3

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

So P_0 must wait

But Need ≤ Work

Step 2

For i = 1

$Need_1$ = 0, 2, 0

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

Finish =

0	1	2	3	4
false	true	false	false	false

Step 2

For i = 2

$Need_2$ = 6 , 0, 0

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

So P_2 must wait

Step 2

For i=3

$Need_3$ = 0, 1, 1

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

Finish =

0	1	2	3	4
false	true	false	true	false

Step 2

For i = 4

$Need_4$ = 4, 3, 1

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

Finish =

0	1	2	3	4
false	true	false	true	true

Step 2

For i = 0

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 < 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

Finish =

0	1	2	3	4
true	true	false	true	true

Step 2

For i = 2

$Need_2$ = 6 , 0, 0

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

Finish =

0	1	2	3	4
true	true	true	true	true

Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

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

More Examples

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Process	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

- Using the resource request algorithm, Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Class Exercise

- Five processes $\boldsymbol{P_0}$ through $\boldsymbol{P_4}$; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time $\boldsymbol{T_0}$:

	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	

- What is the safe sequence, such that all the processes execute without deadlock?

Class Exercise

- Five processes $\boldsymbol{P_0}$ through $\boldsymbol{P_4}$; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time $\boldsymbol{T_0}$:

	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	

- Sequence <$\boldsymbol{P_0, P_2, P_3, P_1, P_4}$> will result in ***Finish[i] = true*** for all ***i***

Chapter 7: Deadlocks

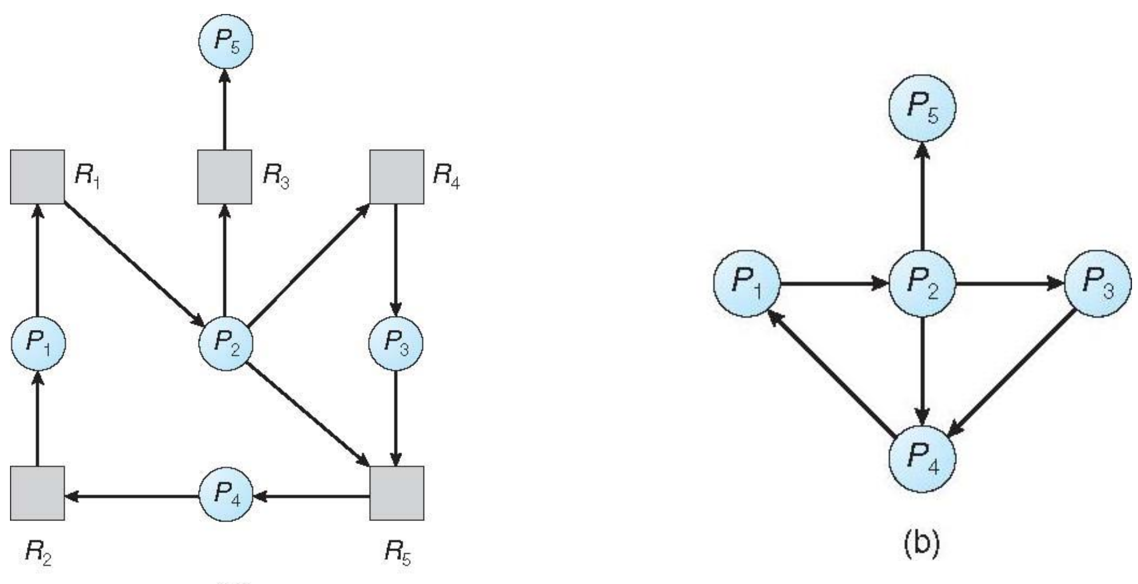
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- **Deadlock Detection**
- Recovery from Deadlock

Deadlock Detection

- Most systems do not do anything about deadlock. They just allow the system to enter a deadlock state if it happens.
- Such systems can use a **deadlock detection algorithm** to discover when deadlock has occurred.
- If deadlock is detected, the system can then use a **deadlock recovery algorithm** to undo the deadlock.

Single Instance of Each Resource Type

- If all resources have only a ***single instance***, then for deadlock detection we use a **wait-for** graph.
  - **Wait-for** graph does not have resource nodes.
- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{P_i \rightarrow P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock



Resource-Allocation Graph Corresponding **wait-for graph**

Several Instances of a Resource Type

- The **wait-for** graph is not applicable to a resource-allocation system with **multiple instances** of each resource type.
- Deadlock detection algorithm used for system with multiple instances of resource types uses **data structures** similar to the banking algorithm.
- **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*** *//resources currently available with the system*

   (b) For ***i* = 1,2, …, *n***, if ***Allocation****$_i$* $\neq$ **0**, then

   Finish*[i] = *false; otherwise, ***Finish*[i] = *true****//Process 'i' is assumed finished, no resources have been allocated to it.*

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request****$_i$* $\leq$ ***Work***
   If no such ***i*** exists, go to step 4

3. ***Work* = *Work* + *Allocation****$_i$* *//System reclaims resources.*

   Finish*[*i*] = *true *//Process 'i' is finished.*

   go to step 2

4. If ***Finish[i] == false***, for some ***i***, 1 $\leq$ ***i*** $\leq$ ***n***, then the system is in deadlock state. Moreover, if

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

Example of Detection Algorithm

- Five processes ***P*$_0$** through ***P*$_4$**; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

[7 2 6]

- 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	

- Does deadlock exist?

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

  [7 2 6]

- 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	

- **Does deadlock exist?**
- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***
- Hence No deadlock exist.

Example of Detection Algorithm (Cont'd)

- ***P_2*** requests an additional instance of type ***C***
- 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 1**	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now considering a new request from P_2 , is the system still safe?

Example of Detection Algorithm (Cont'd)

- $\boldsymbol{P_2}$ requests an additional instance of type $\boldsymbol{C}$
- Snapshot at time $\boldsymbol{T_0}$:

	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 1**	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now considering a new request from P_2 , is the system still safe?
  - Can reclaim resources held by process $\boldsymbol{P_0}$, available resources becomes [0 1 0] but insufficient resources to fulfill other processes' requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

Deadlock Detection Exercise (1/5)

- Four processes P_1 through P_4; three resource types R_1 (5 instances), R_2 (4 instances), and R_3 (5 instances)

  [5 4 5]
- Snapshot at time T_0:
- Since $Allocation_i \neq 0$ for all i, therefore Finish [i]= FALSE, initially.

Marked = {F, F, F, F};

Available = (0, 0, 1); **W** is initialized to (0, 0, 1)

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation A

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3	0	0	1
P_4	1	1	1

Request Q

Q(P3) = (0, 0, 1) <= (0, 0, 1) = W

Deadlock Detection Exercise (2/5)

- P3 is included in safe sequence

  < P3, >

Marked = {F, F, T, F};

$W = (1, 1, 1) = (0, 0, 1) + (1, 1, 0)$

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation A

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3			
P_4	1	1	1

Request Q

Deadlock Detection Exercise (3/5)

- Now P_4 is included in safe sequence

  $< P_3, P_4, \ >$

Marked = {F, F, T, T};

W = (2, 2, 2) = (1, 1, 1) + (1, 1, 1);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation A

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3			
P_4			

Request Q

Deadlock Detection Exercise (4/5)

- Now P_2 is included in safe sequence

  $< P_3, P_4, P_2, \quad >$

Marked = {F, T, T, T};

W = (4, 3, 2) = (2, 2, 2) + (2, 1, 2);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation A

	R_1	R_2	R_3
P_1	3	2	1
P_2			
P_3			
P_4			

Request Q

Deadlock Detection Exercise (5/5)

- Finally, P_1 is included in safe sequence

  < P_3,P_4, P_2, P_1 >
- Sequence <***P_3, P_4, P_2, P_1***> will result in ***Finish[i] = true*** for all ***i***
- Hence No deadlock exist.

Marked = {T, T, T, T};

W = (5, 4, 3) = (4, 3, 2) + (1, 1, 1);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation A

	R_1	R_2	R_3
P_1			
P_2			
P_3			
P_4			

Request Q

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- **Option 1:** If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.
- **Option 2:** We can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
  - We cannot only identify the deadlocked set of processes but also the specific process that caused the deadlock.
  - **Downside:** However, invoking the deadlock detection algorithm for every resource request will incur overhead.

Chapter 7: Deadlocks

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

Recovery from Deadlock

- Two options for breaking the deadlock:
  - **Process Termination:** Abort one or more processes to break the circular wait.
    - Abort all deadlocked processes.
    - Or abort one process at a time until the deadlock cycle is eliminated
  - **Resource Preemption:** Preempt some resources from one or more of the deadlock processes. Using resource preemption, we successively preempt some resource from processes and give these resources to other processes until the deadlock is broken.
  - Some issues:
    - **Selecting a victim:** Which resources and which processes are to be preempted?
    - **Rollback:** Return to some safe state, restart process for that state
    - **Starvation:** same process may always be picked as victim, include number of rollback in cost factor

End of Chapter 7