Computer Organization and Software Systems

Session 12

Dr. Lucy J. Gudino
WILP & Department of CS & IS

BITS Pilani
Pilani Campus

Today's Session

List of Topic Title	Text/Ref Book/external resource
• Deadlock	T2

Deadlock Characterization

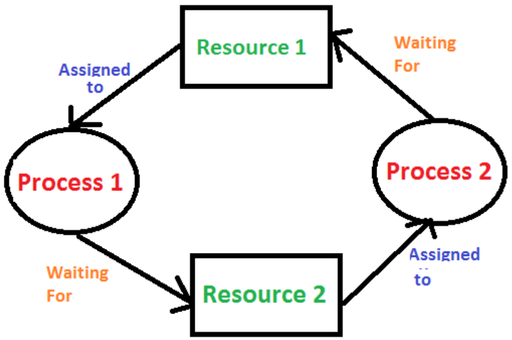
Mutual exclusion: only one process at a time can use a resource.

Hold and wait: a process holding at least one resource and 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 $\{P_0, P_1, \ldots, P_0\}$ 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

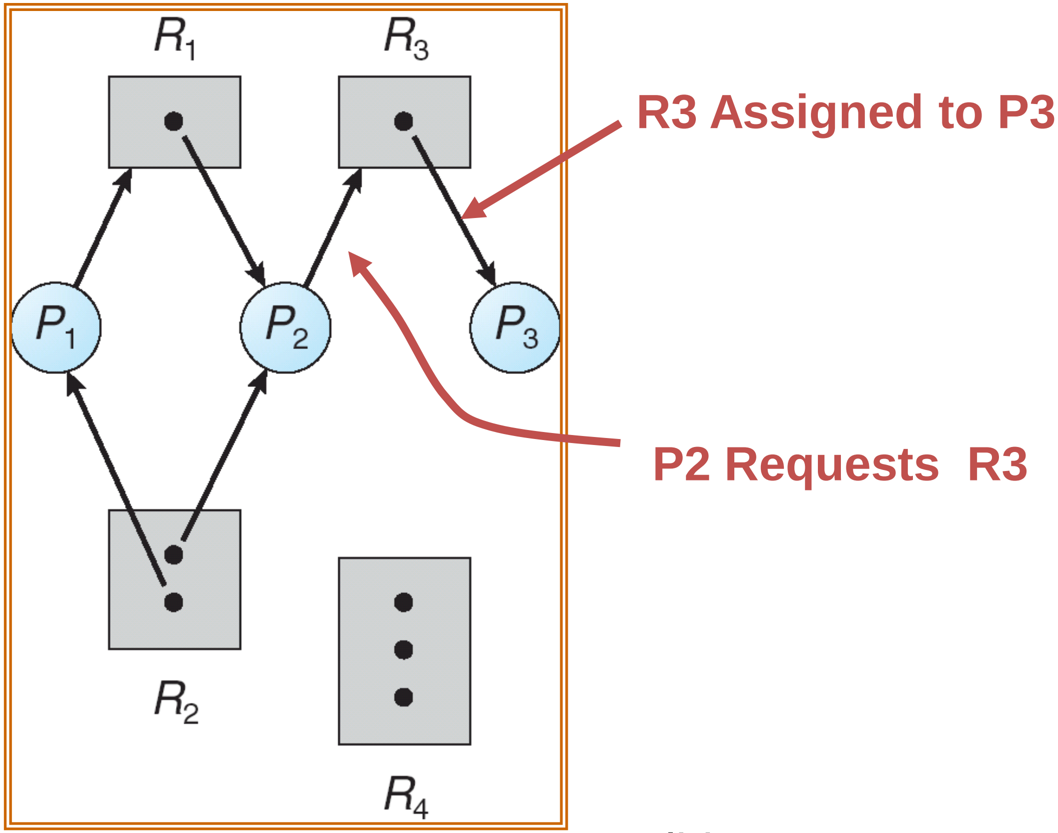
$P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



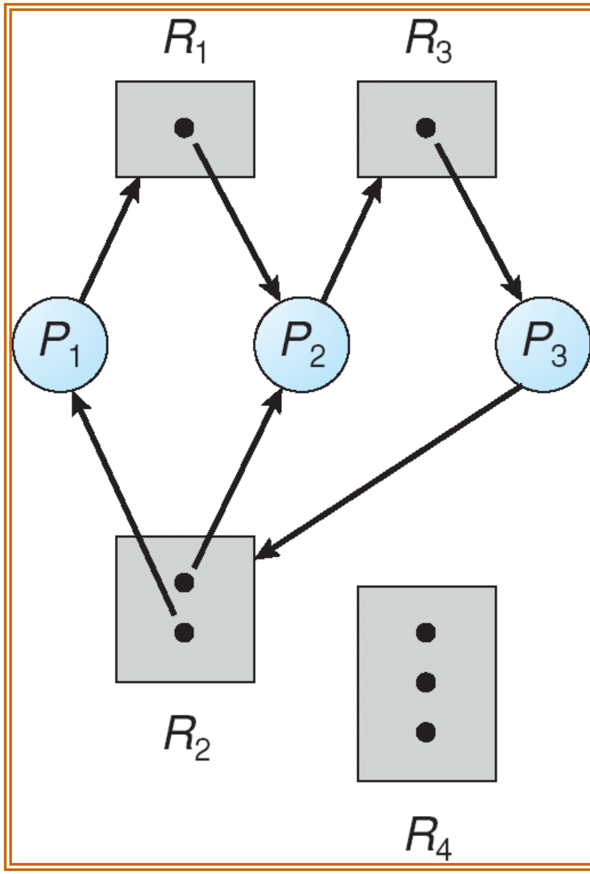
NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur

Example 1



Example 2



Cycle 1:
P1→R1→P2→R3→P3→R2→P1
Cycle 2:
P2→R3→P3→R2→P2

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then there is a deadlock.

    a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock
  - if several instances per resource type, there **may be** a deadlock.

    a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock

Graph With A Cycle But No Deadlock

Cycle 1:

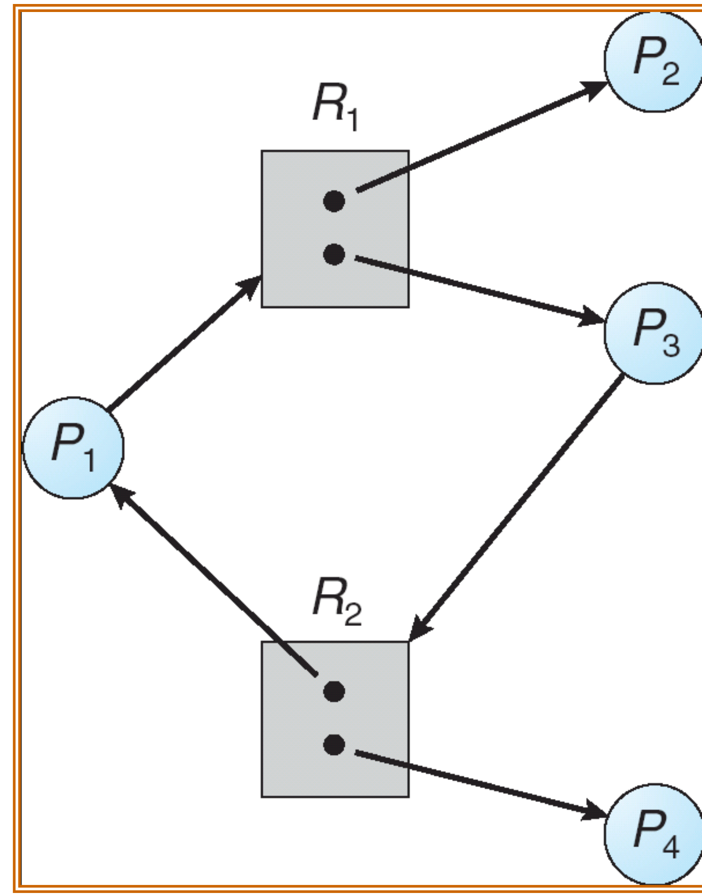
P1→R1→P3→R2→P1

But there is no deadlock.

Conclusion:

No Cycle: No deadlock

Cycle : may or may not be deadlock



Methods for Handling Deadlocks

- Deadlock Prevention and Deadlock avoidance:
  - ensuring that the system will *never enter a deadlock state.*
- Deadlock detection and recovery:
  - Allow the system to enter a deadlock state, detect it, and recover
- Ignore the deadlock problem altogether and pretend that deadlocks never occur in the system.

Deadlock Prevention Vs. Deadlock avoidance

Deadlock Prevention:

- The goal is to ensure that at least one of the necessary conditions for deadlock can never hold.
- The system does not require additional apriori information regarding the overall potential use of each resource for each process.

Deadlock avoidance:

- The goal for deadlock avoidance is to the system must not enter an unsafe state.
- The system requires additional apriori information regarding the overall potential use of each resource for each process.

Deadlock Prevention

- **Mutual Exclusion** – Sharable vs nonsharable resources
  - must hold for nonsharable resources . Example : Printer
  - not required for sharable resources. Example: Read only files
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Protocol 1: requires each process to request and be allocated all its resources before it begins execution
    - No partial resource allocation
  - Protocol 2: Allow process to request resources only when the process has none.
    - may request resources and use them, needs some more then release first.
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption –**

  Protocol 1:

  – If a process is holding some resources and 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

  – Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

  Protocol 2:

  – If a process requests some resources, check whether they are available. If so, allocate them.

  – If not, check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resources from the waiting process and allocate them to the requesting process.

Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

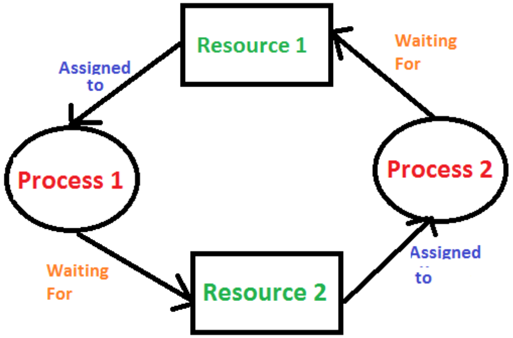
  $F: R \rightarrow N$

protocol to be followed is $F(R_i) < F(R_j)$

Resource allocated

Resource to be requested

Main Drawback: low device utilization and reduced system throughput



Deadlock Avoidance

- Each process provides OS with information about its requests and releases for resources R_i
- Advantage: Simplest and most useful model
- Disadvantage : knowing all future requests and releases is difficult
- A resource allocation state is defined by
  - # of available resources
  - # of allocated resources to each process
  - maximum demands by each process
- On process request for a resource, OS needs to check whether the system is in **safe state** or not

Deadlock Avoidance…

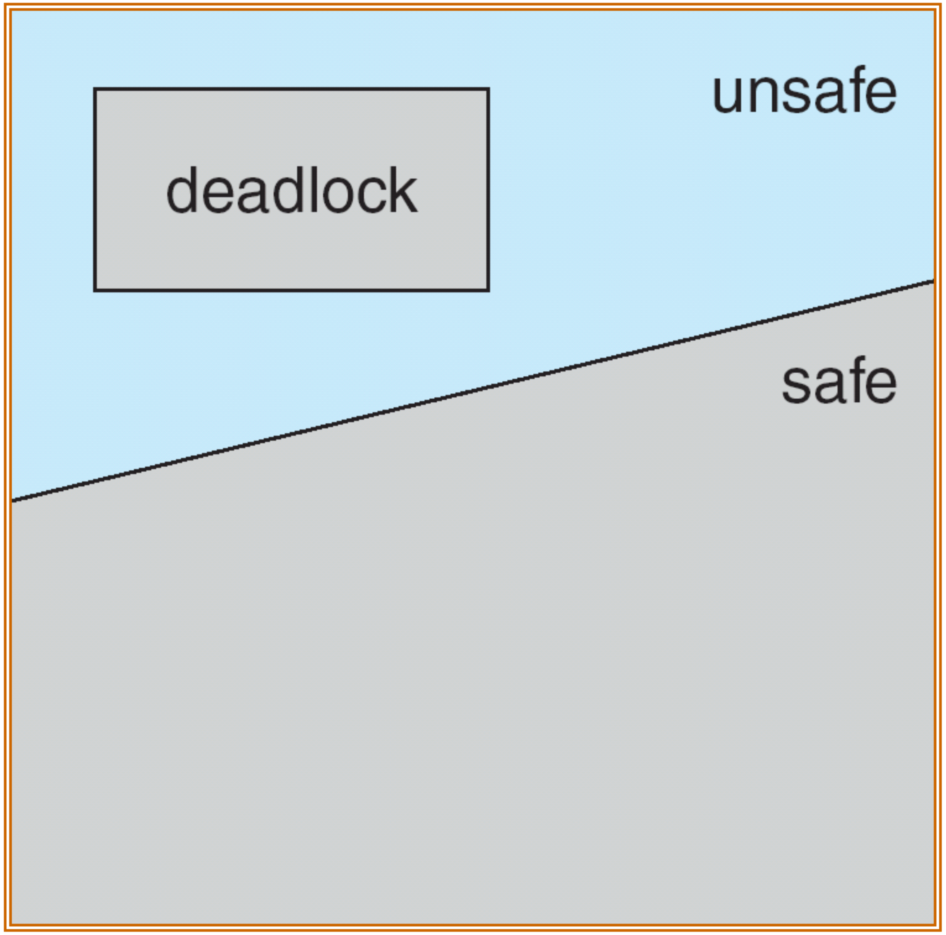
- A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources that are enough for the next one to finish
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.

Safe, Unsafe, Deadlock State



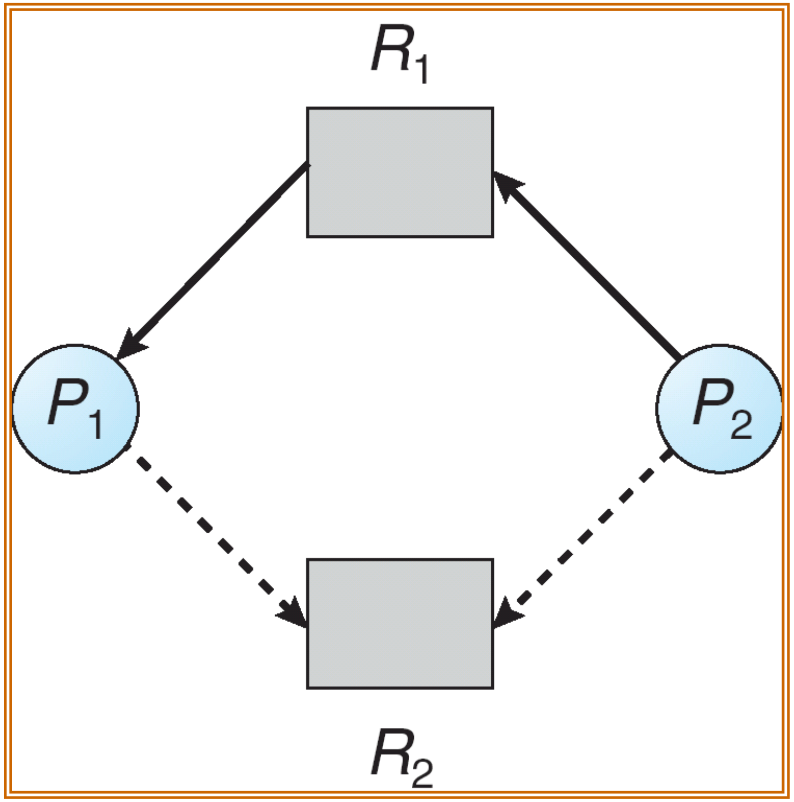
Avoidance algorithms

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

Resource-Allocation Graph Scheme

- Two edges : Request and assignment edge
- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; some time in future
  - represented by a dashed line.
- Claim edge converted 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



Banker's Algorithm

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

Data Structures for the Banker's Algorithm

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

- ***Available***: Vector of length m. If Available$[j] = k$, there are k instances of resource type R_j available.
- ***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.

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

Example of Banker's Algorithm

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:

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

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.

Example (Cont. 1/3)

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

	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	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

The system is in a safe state since the sequence < > satisfies safety criteria.

Example (Cont. 2/3)

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

	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	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

The system is in a safe state since the sequence < > satisfies safety criteria.

Example (Cont. 3/3)

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

	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	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

The system is in a safe state since the sequence < > satisfies safety criteria.

Example (Solution)

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

	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	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

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

Work = 332

Finish = { F F F F F }

P1: Finish [1] = T;
Work = 5 3 2

P3: Finish [3] = T;
Work = 7 4 3

P4: Finish [4] = T;
Work = 7 4 5

P0: Finish [0] = T;
Work = 7 5 5

P2: Finish [2] = T;
Work = 10 5 7

Example: P_1 Request (1,0,2)

Two steps:

1. Run Resource – Request Algorithm
2. Check whether the system is safe using safety algorithm

Resource-Request Algorithm for Process P_i

BACK

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

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

$$Available = Available - Request_i\ ;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- *If safe $\Rightarrow$ the resources are allocated to Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example: P_1 Request (1,0,2)

Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i\ ;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

Check that Request <= need

Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	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	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Example: P_1 Request (1,0,2)

	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	

Check This…

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

	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	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Deadlock Detection

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

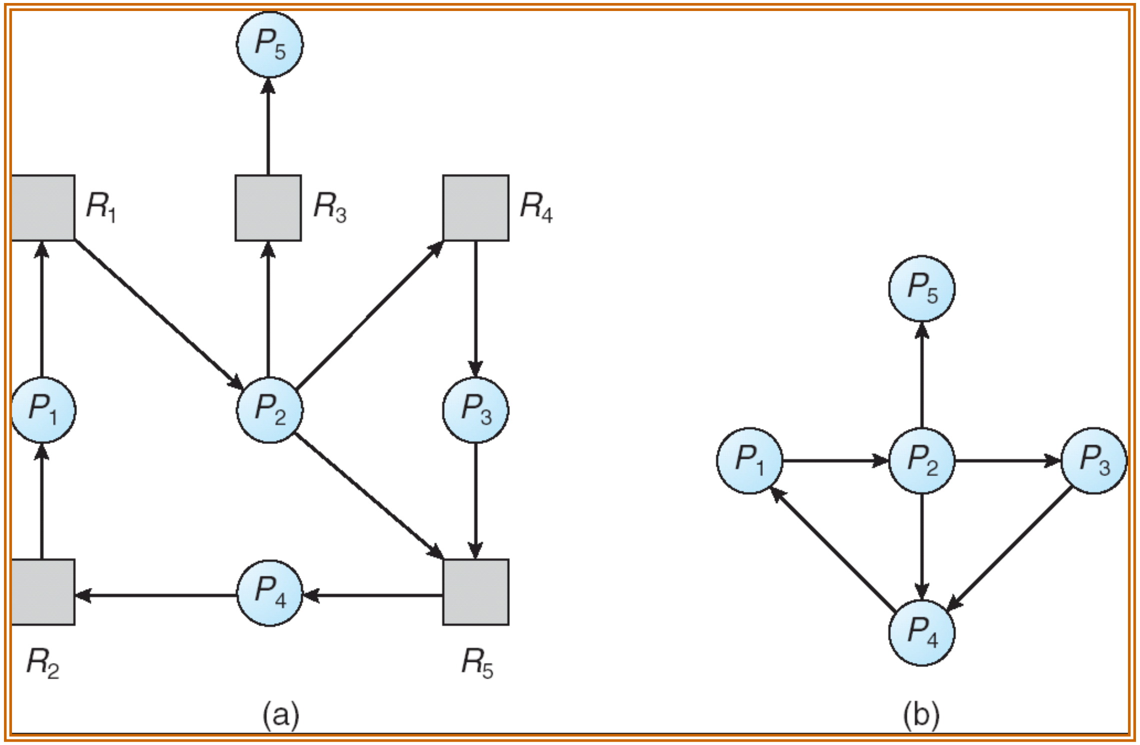
Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j *to release the resource*

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

Available: A vector of length *m* indicates the number of available resources of each type.

Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i,j*] = *k*, then process P_i is requesting *k* more instances of resource type R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

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

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

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

Example of Detection Algorithm

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

Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence $\langle P_0, P_2, P_3, P_4, P_1 \rangle$ will result in $Finish[i]$ = true for all i.

Example (Cont.)

P_2 requests an additional instance of type C.

	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

State of system?

- Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
- Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Recovery from Deadlock: 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?

Today's Session

List of Topic Title	Text/Ref Book/external resource
• Deadlock… • Memory Management	T2

Deadlock Detection

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

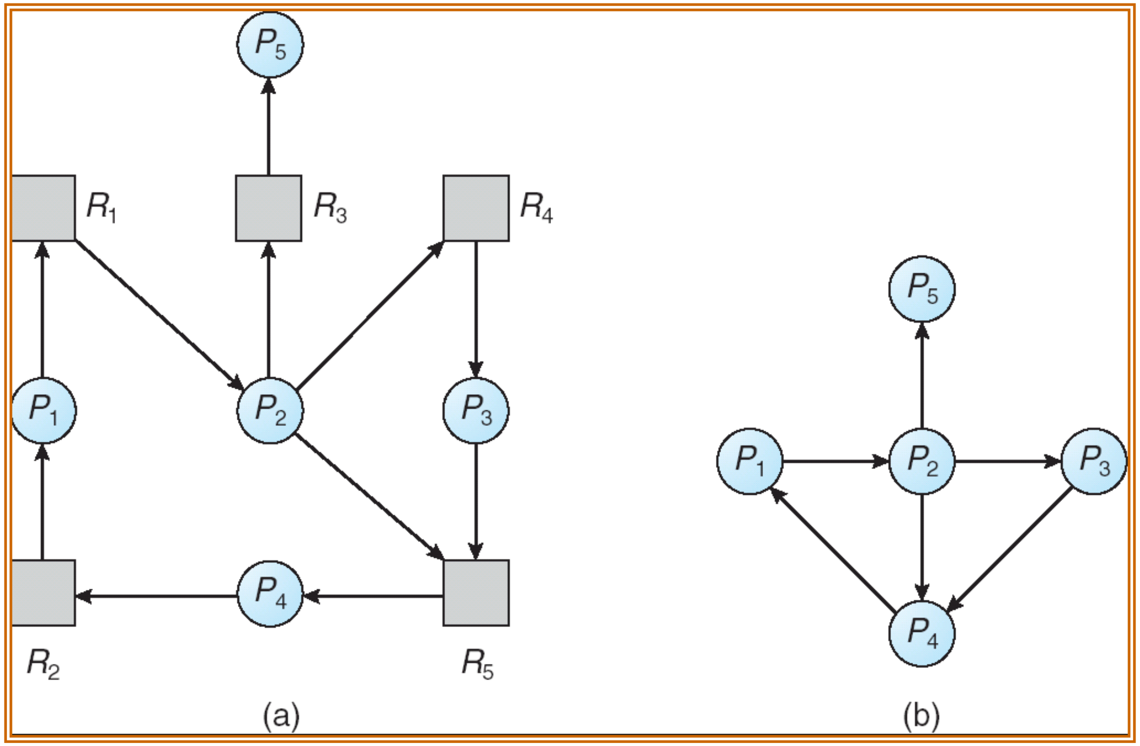
Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j *to release the resource*

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

Available: A vector of length *m* indicates the number of available resources of each type.

Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i,j*] = *k*, then process P_i is requesting *k* more instances of resource type R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

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

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

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

Example of Detection Algorithm

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

Snapshot at time T_0:

vecto

	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 $\langle P_0, P_2, P_3, P_4, P_1 \rangle$ will result in $Finish[i]$ = true for all i.

Example (Cont.)

P_2 requests an additional instance of type C.

	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

State of system?

- Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
- Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Contd…

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

Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence $\langle P_0, P_2, P_3, P_4, P_1\rangle$ will result in *Finish*[*i*] = true for all *i*.

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?