ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

Operating Systems

Lecture 7: Deadlocks

© Fall 2024 – **Dr. Alyaa Hamza – Dr. Ahmed Moawad**
Alyaa.hamza@sut.edu.eg ahmed.moawad@sut.edu.eg

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.

System Model

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

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

V is partitioned into two types:

$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

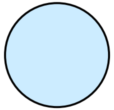
$R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

request edge – directed edge $P_i \rightarrow R_j$

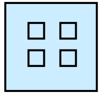
assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

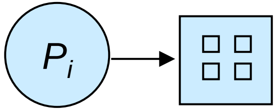
Process



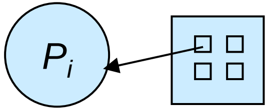
Resource Type with 4 instances



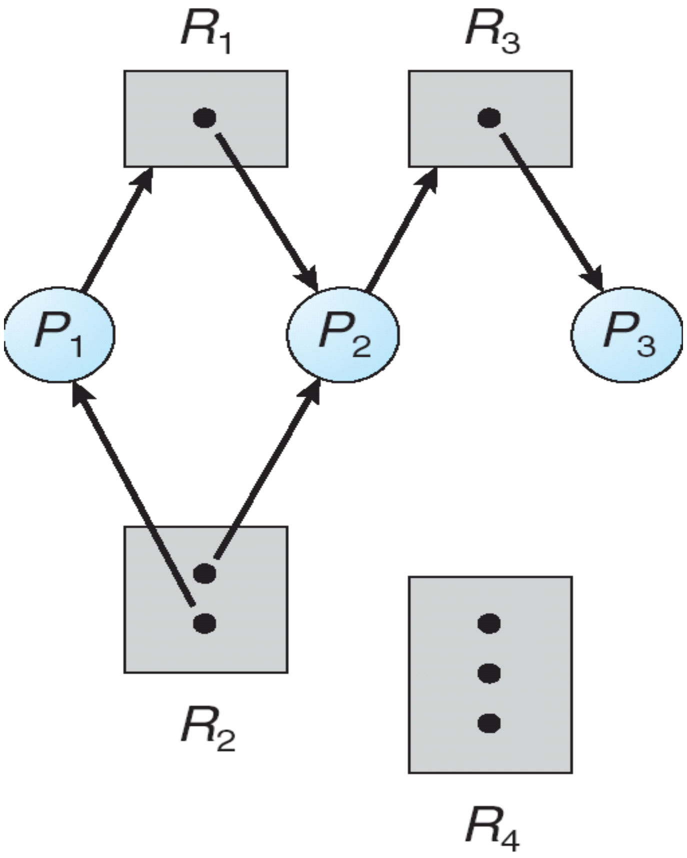
P_i requests instance of R_j



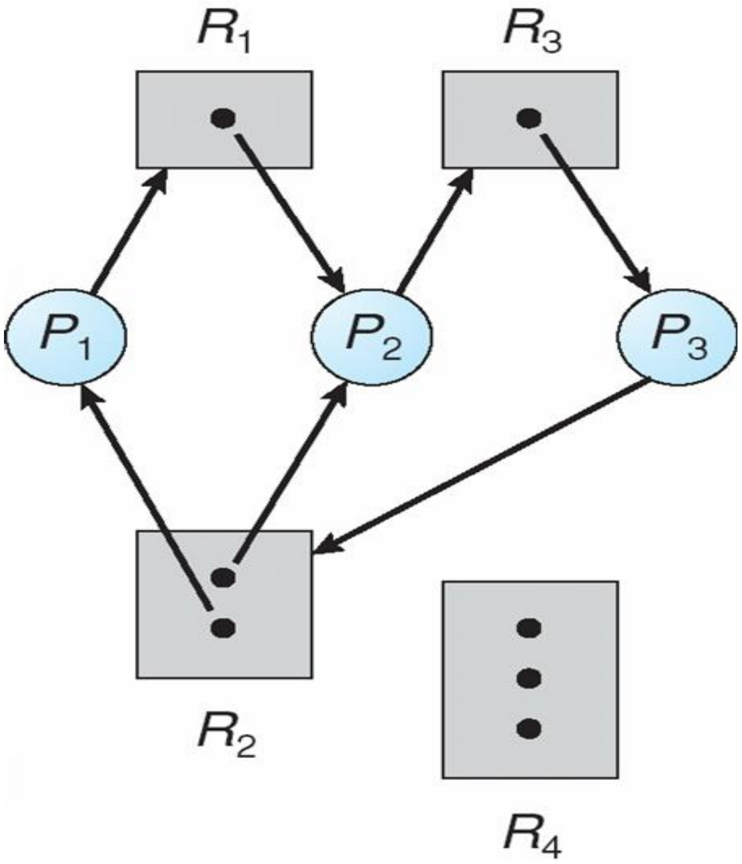
P_i is holding an instance of R_j



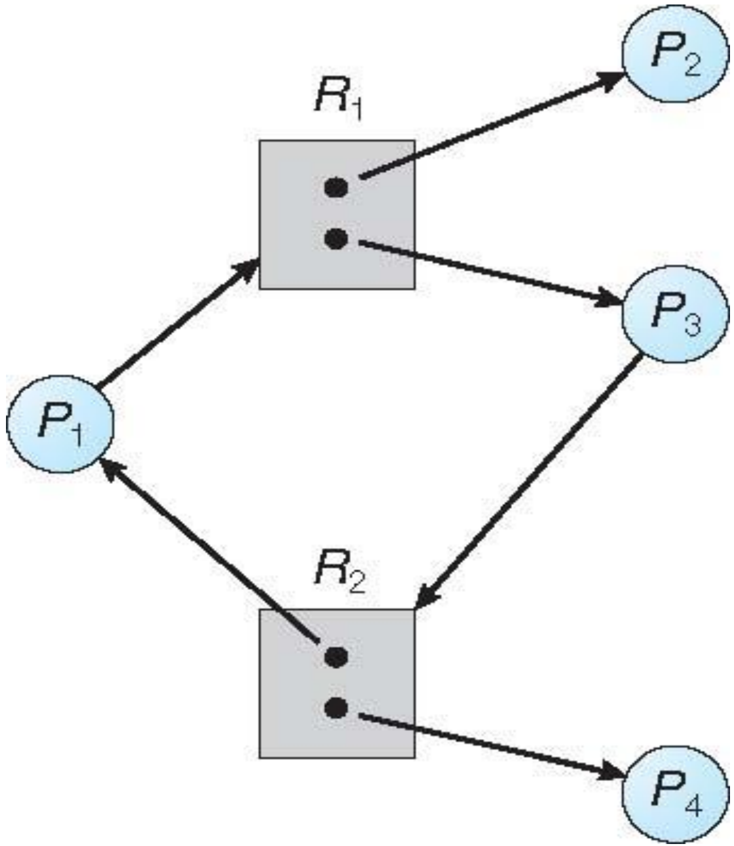
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



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

Deadlock Avoidance

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

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

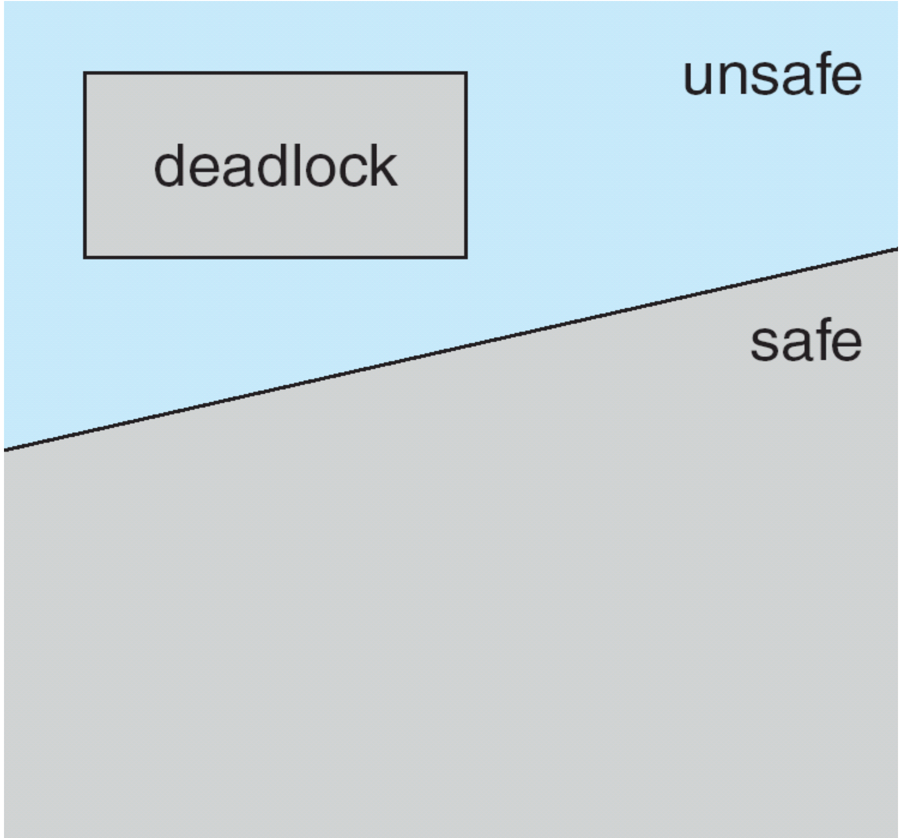
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, 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

Basic Facts

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

Safe, Unsafe, Deadlock State



Avoidance Algorithms

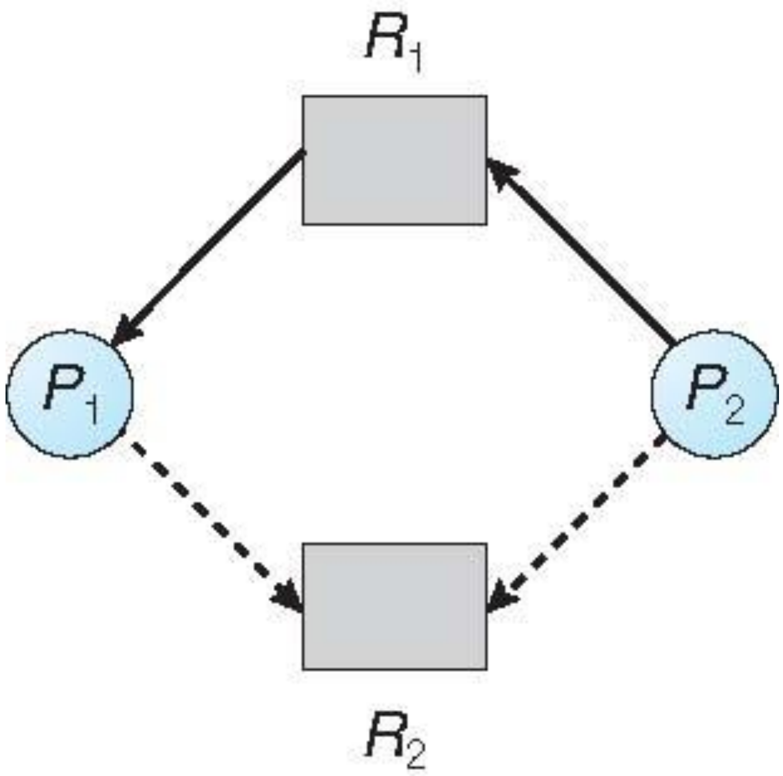
- Single instance of a resource type
  - Use a resource-allocation graph

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

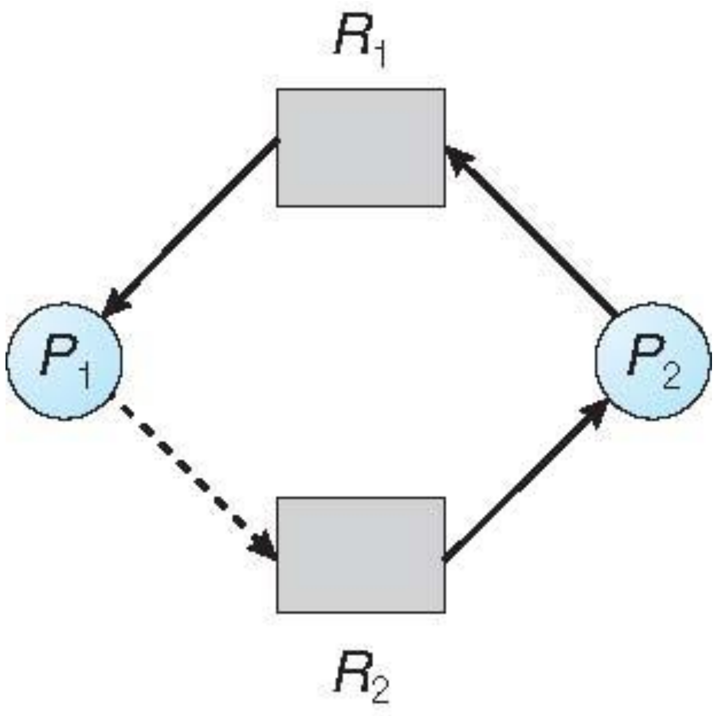
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



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*]**

New available = available + Allocation

Safety Algorithm

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

   Work* = *Available

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

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

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

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

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

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

   Finish*[*i*] = *true

   go to step 2

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

Example 1 of Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types:

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

Processes	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			

Example 1 (Cont.)

Solution

First, we calculate the Need column: Need = Max - Allocation

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

Example 1 (Cont.)

Second, we must check the availability:

- P_0 *needs (7, 4, 3) > available (3, 3, 2) , P_0 will not work*
- P_1 *needs (1, 2, 2) < available (3, 3, 2) , P_1 will work*
  - *After P_1 ended, available will equal = (3, 3, 2) + (2, 0, 0) = (5, 3, 2)*
- P_2 needs (6, 0, 0) > available (5, 3, 2), P_2 *will not work*
- P_3 *needs (0, 1, 1)* < available (5, 3, 2), P_3 *will work*
  - *After P_3 ended, available will equal = (5, 3, 2) + (2, 1, 1) = (7, 4, 3)*
- P_4 *needs (4, 3, 1)* < available (7, 4, 3), P_4 *will work*
  - *After P_4 ended, available will equal = (7, 4, 3) + (0, 0, 2) = (7, 4, 5)*
- Now again check P_0 (7, 4, 3) < available (7, 4, 5), P_0 *will work*
  - *After P_0 ended, available will equal = (7, 4, 5) + (0, 1, 0) = (7, 5, 5)*
- Now again check P_2 (6, 0, 0) < available (7, 5, 5), P_2 *will work*
  - *After P_2 ended, available will equal = (7, 5, 5) + (3, 0, 2) = (10, 5, 7)*
- **The system will be in the safe state for the sequences (*P_1, P_3, P_4, P_0, P_2*)**

Example 2 of Banker's Algorithm (Class Work)

- 5 processes P_0 through P_4;
- 3 resource types:

  A (10 instances), *B* (6 instances), and *C* (7 instances)

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

Example 2 (Cont.)

Solution

First, we calculate the Need column: Need = Max - Allocation

Processes	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

Example 2 (Cont.)

Second, we must check the availability:

- *P_0 needs (3, 2, 1) > available (2, 1, 0), P_0 will not work*
- *P_1 needs (1, 1, 0) < available (2, 1, 0), P_1 will work*
  - *After P_1 ended, available will equal = (2, 1, 0) + (2, 1, 2) = (4, 2, 2)*
- P_2 needs (5, 0, 1) > available (4, 2, 2), P_2 *will not work*
- P_3 *needs (7, 3, 3)* < available (4, 2, 2), P_3 *will not work*
- P_4 *needs (0, 0, 0)* < available (4, 2, 2), P_4 *will work*
  - *After P_4 ended, available will equal = (4, 2, 2) + (1, 1, 2) = (5, 3, 4)*
- Now again check P_0 *(3, 2, 1) > available (5, 3, 4), P_0 will work*
  - *After P_0 ended, available will equal = (5, 3, 4) + (1, 1, 2) = (6, 4, 6)*
- Now again check P_2 *(5, 0, 1) > available (6, 4, 6), P_2 will work*
  - *After P_2 ended, available will equal = (6, 4, 6) + (4, 0, 1) = (10, 4, 7)*
- Now again check P_3 *(7, 3, 3) > available (10, 4, 7), P_3 will work*
  - After P_3 ended, available will equal = (10, 4, 7) + (0, 2, 0) = (10, 6, 7)
- **The system will be in the safe state for the sequences (P_1, P_4, P_0, P_2, P_3)**

Detection Algorithm

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

   Initialize:

   (a) ***Work*** = ***Available***

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

Detection Algorithm (Cont.)

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 1 of Detection Algorithm

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

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

Example 1 of Detection Algorithm (Cont.)

Solution

- *P_0 requests (0, 0, 0) = available (0, 0, 0), P_0 will work*
  - *After P_0 ended, available will equal = (0, 0, 0) + (0, 1, 0) = (0, 1, 0)*
- *P_1 requests (2, 0, 2) > available (0, 1, 0), P_1 will not work*
- *P_2 requests (0, 0, 0) < available (0, 1, 0), P_2 will work*
  - *After P_2 ended, available will equal = (0, 1, 0) + (3, 0, 3) = (3, 1, 3)*
- *P_3 requests (1, 0, 0) < available (3, 1, 3), P_3 will work*
  - *After P_0 ended, available will equal = (3, 1, 3) + (2, 1, 1) = (5, 2, 4)*
- Now again check *P_1 (2, 0, 2) > available (5, 2, 4), P_1 will work*
  - *After P_1 ended, available will equal = (5, 2, 4) + (2, 0, 0) = (7, 2, 4)*
- *P_4 requests (0, 0, 2) < available (7, 2, 4), P_4 will work*
  - *After P_4 ended, available will equal = (7, 2, 4) + (0, 0, 2) = (7, 2, 6)*
- The system will be in the safe state for the sequences *(P_0, P_2, P_3, P_1, P_4)*

Example 2 of Detection Algorithm

- P_2 requests an additional instance of type C

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

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

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

Recovery from Deadlock: Resource Preemption

- **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

End of Chapter 7