Chapter 7: Deadlocks

Chapter 7: Deadlocks

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

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.
- Resource types

  CPU cycles, memory space, I/O devices

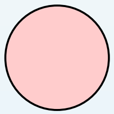
- Each process utilizes a resource as follows:
  - Request
    - open(), malloc(), wait()
  - Use
  - Release
    - close(), free(), signal()

Deadlock Characterization

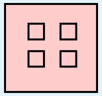
- Deadlock can arise if four conditions hold simultaneously.
  - *Mutual exclusion*: only one process at a time can use a resource.
  - *Hold and wait*: a process holding at least one resource is waiting to acquire additional resources held by other processes.
  - *No preemption*: a resource can be released only voluntarily by the process holding it.
  - *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, …, 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

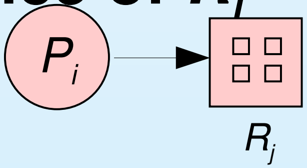
- Process



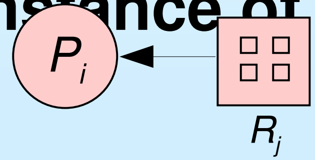
- Resource Type with 4 instances



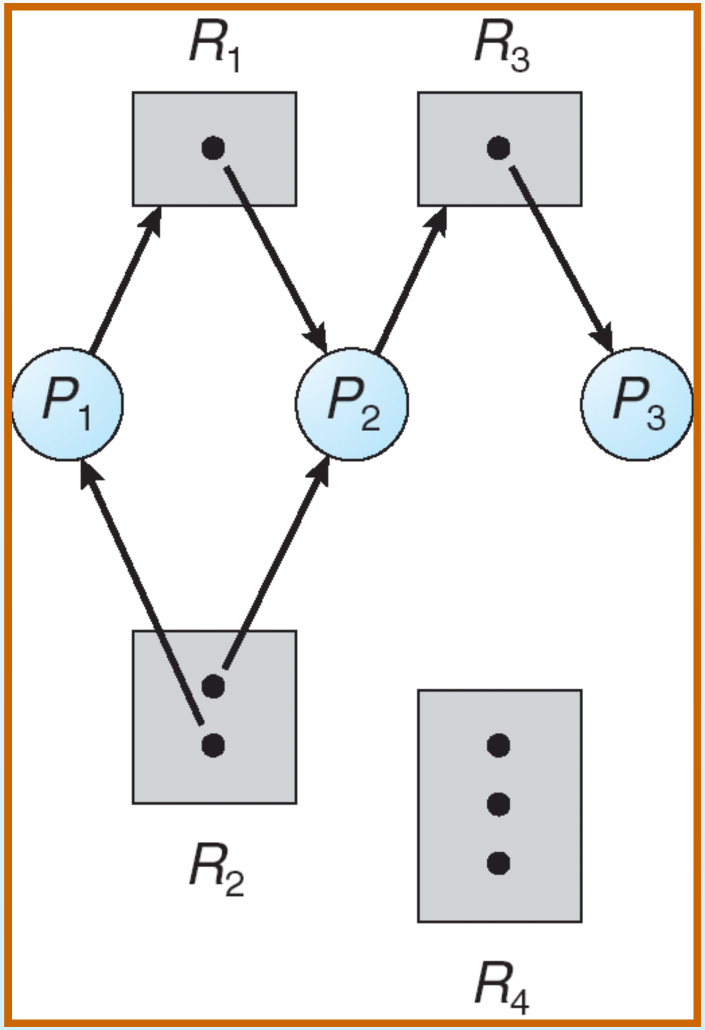
- P_i requests instance of R_j



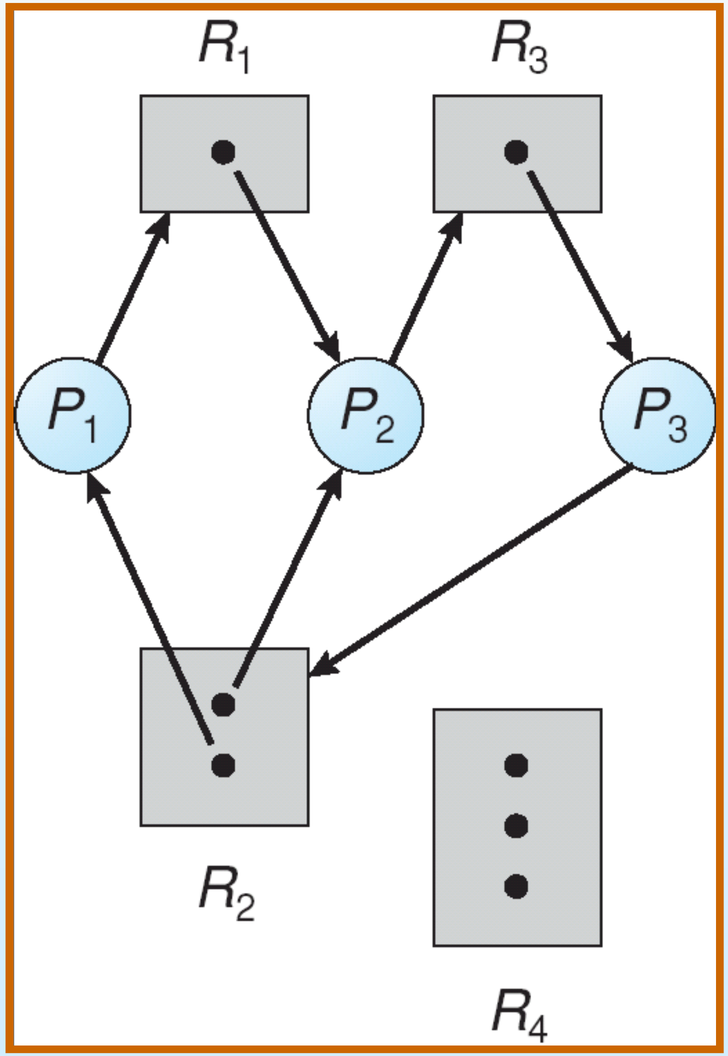
- P_i is holding an instance of R_j



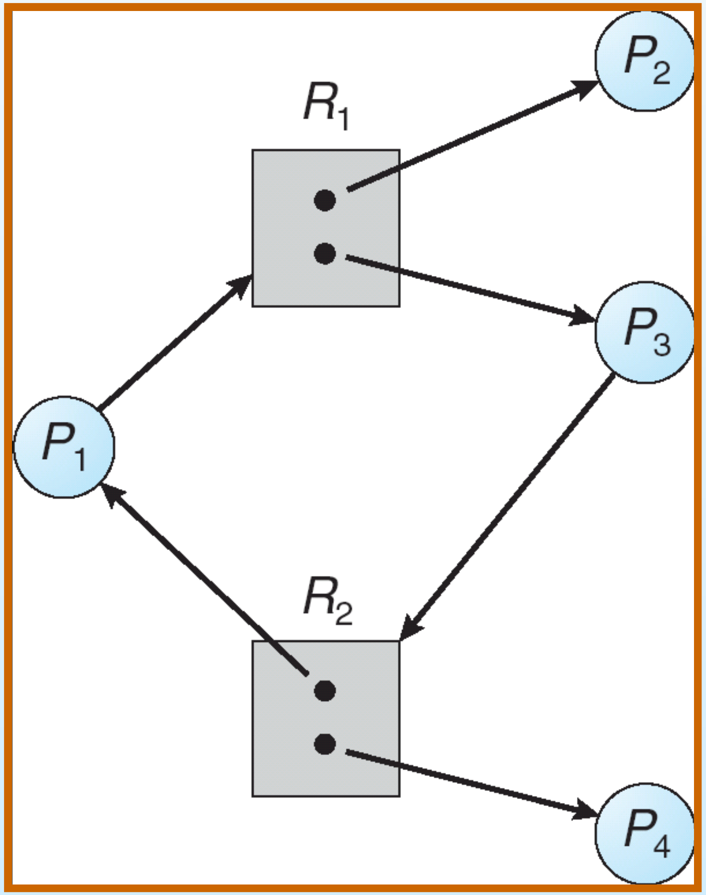
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- 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 and Windows.
  - It is up to the application developer to handle deadlock

Deadlock Prevention

- **Attack Mutual Exclusion**
  - not required for sharable resources
  - must hold for nonsharable resources.

- **Attack Hold and Wait**
  - Get all or none
    - Require process to request and get all its resources before it begins execution
    - Allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

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

- 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

- A process requests a resource, system must decide if the allocation leaves the system in a safe state.
- Safe state
  - A sequence $<P_1, P_2, \ldots, P_n>$ of processes
  - The resources that P_i can still request equal 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, release resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe or Unsafe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Safe or Unsafe?

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

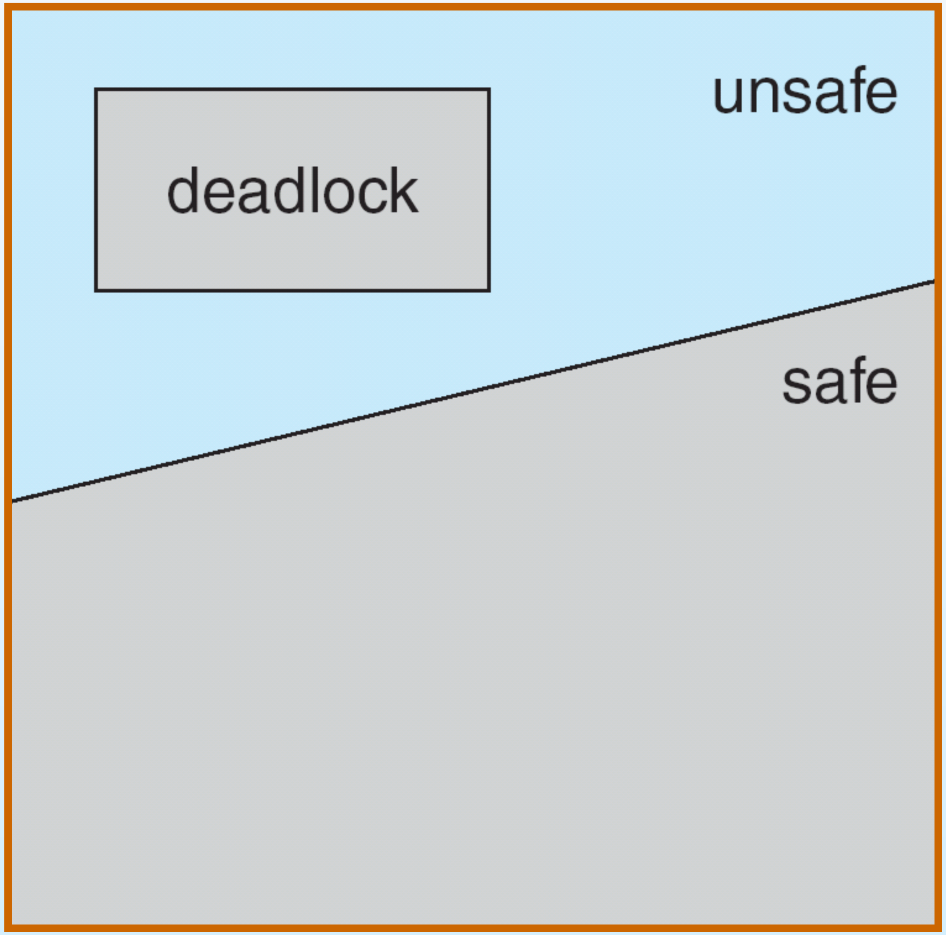
	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Basic Facts

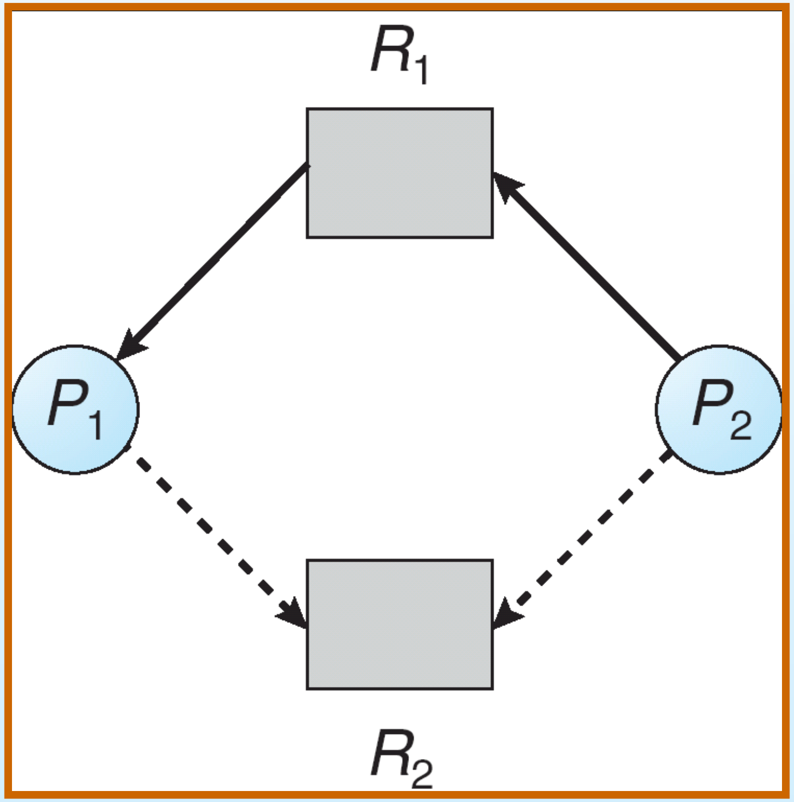
- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



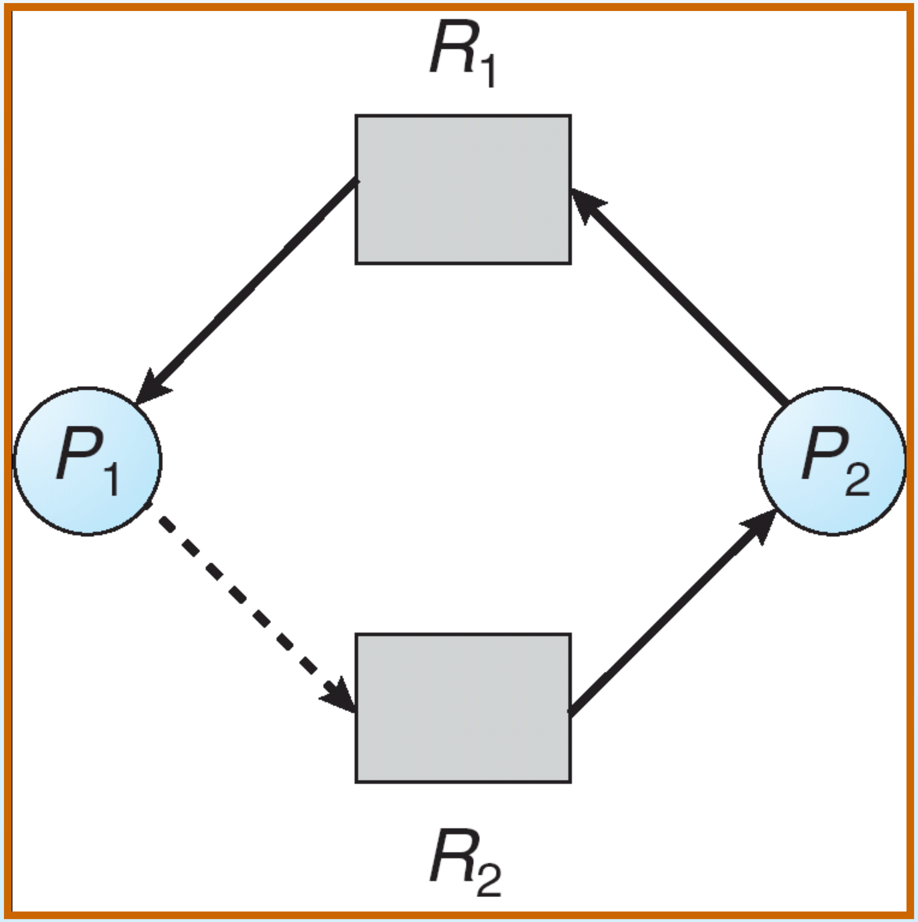
Avoidance algorithms

- 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



Unsafe State In Resource-Allocation Graph



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.
  - Is this possible?
- 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.

Example of Banker's Algorithm

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	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 $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available.

	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Sequence < P_1, P_3, P_4, P_0, P_2> is safety.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

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

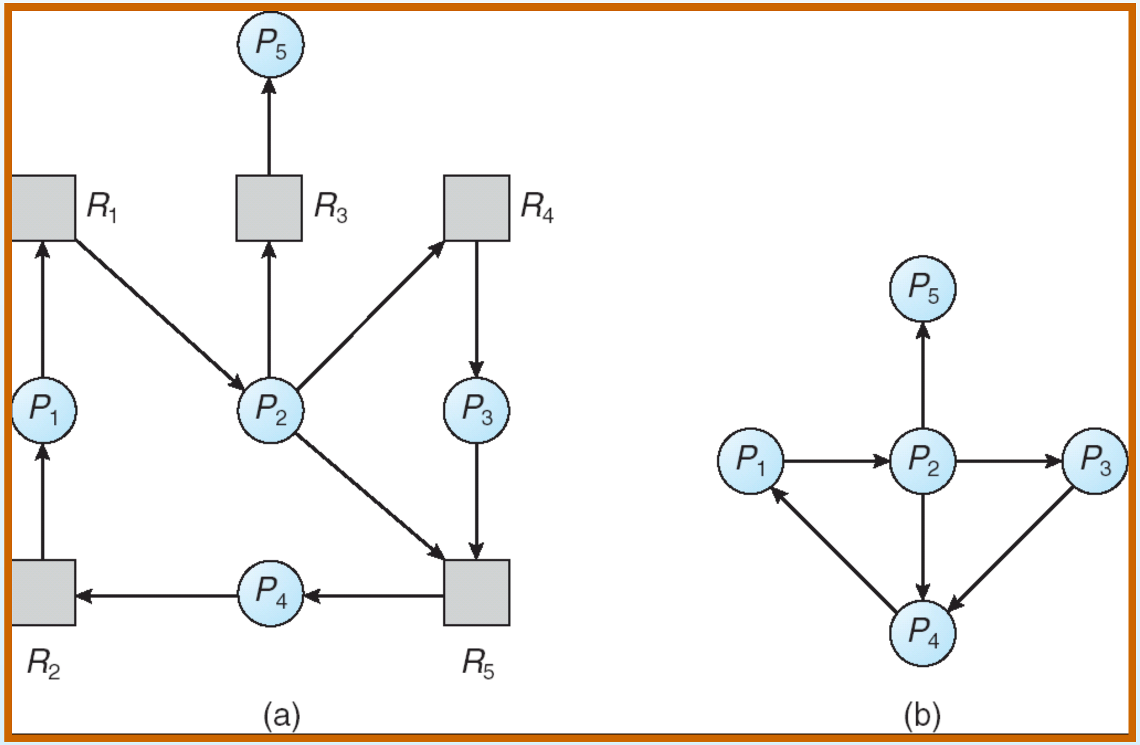
Single Instance of Each Resource Type

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

- 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.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Example of Detection Algorithm

- 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 $<P_0, P_2, P_3, P_1, P_4>$.

Example (Cont.)

- P_2 requests an additional instance of type C.

	Request		
	A	B	C
P_2	0	0	1

- 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?

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
  - So include number of rollback in cost factor.

End of Chapter 7