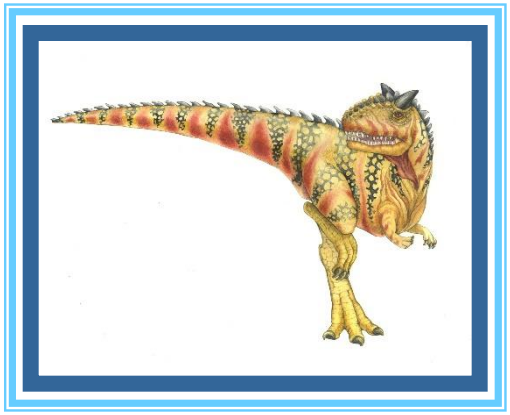
Operating Systems



Lecture 6:
Deadlocks

Chapter 6:
Deadlocks & Starvation
William Stallings
Chapter 7
Schilberchatz, Galvin

Contact Information

Instructor: Engr. Dr. Shamila Nasreen

Assistant Professor

Department of Software Engineering

MUST

Email: Shamila.se@must.edu.pk

Office hours: Wednesday: 8:30-10:00 AM

Thursday: 8:30-10:00 AM

Books

- **Text Book:**

  Silberschatz, Galvin, “Operating Sytems Concepts” 8th Edition, John Wiley, 2007

Reference Book:

1. William Stallings, “Operating Systems”
2. Harvey M. Deitel, “Operating Systems Concepts”

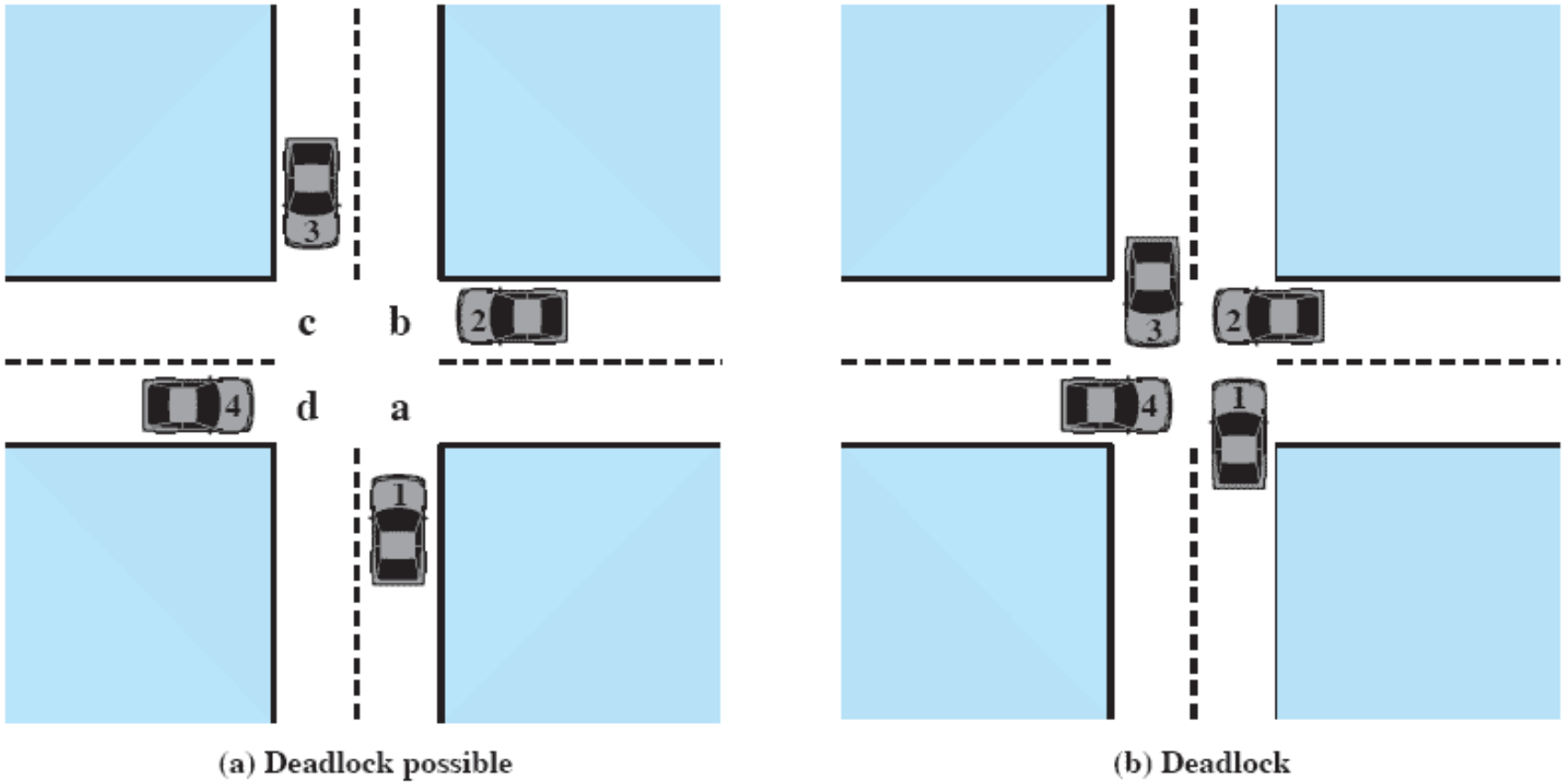
Today's Lecture Agenda

- Principles of Deadlocks
  - Reusable Resources
  - Consumable Resources
  - Resource Allocation Graphs
- The Conditions for Deadlock
- Deadlock Prevention
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait
- Deadlock Avoidance
  - Process Initiation Denial
  - Resource Allocation Denial
- Deadlock Detection
- Deadlock Detection Algorithm & Recovery

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event
  - (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set.
- Deadlock is permanent because none of the events is ever triggered.
- No efficient solution
- Involve conflicting needs for resources by two or more processes.

Deadlock



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

Deadlocks

- Each process needs exclusive use of both resources for a certain period of time.
- Two processes P and Q have the following general form:

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Example of Deadlocks

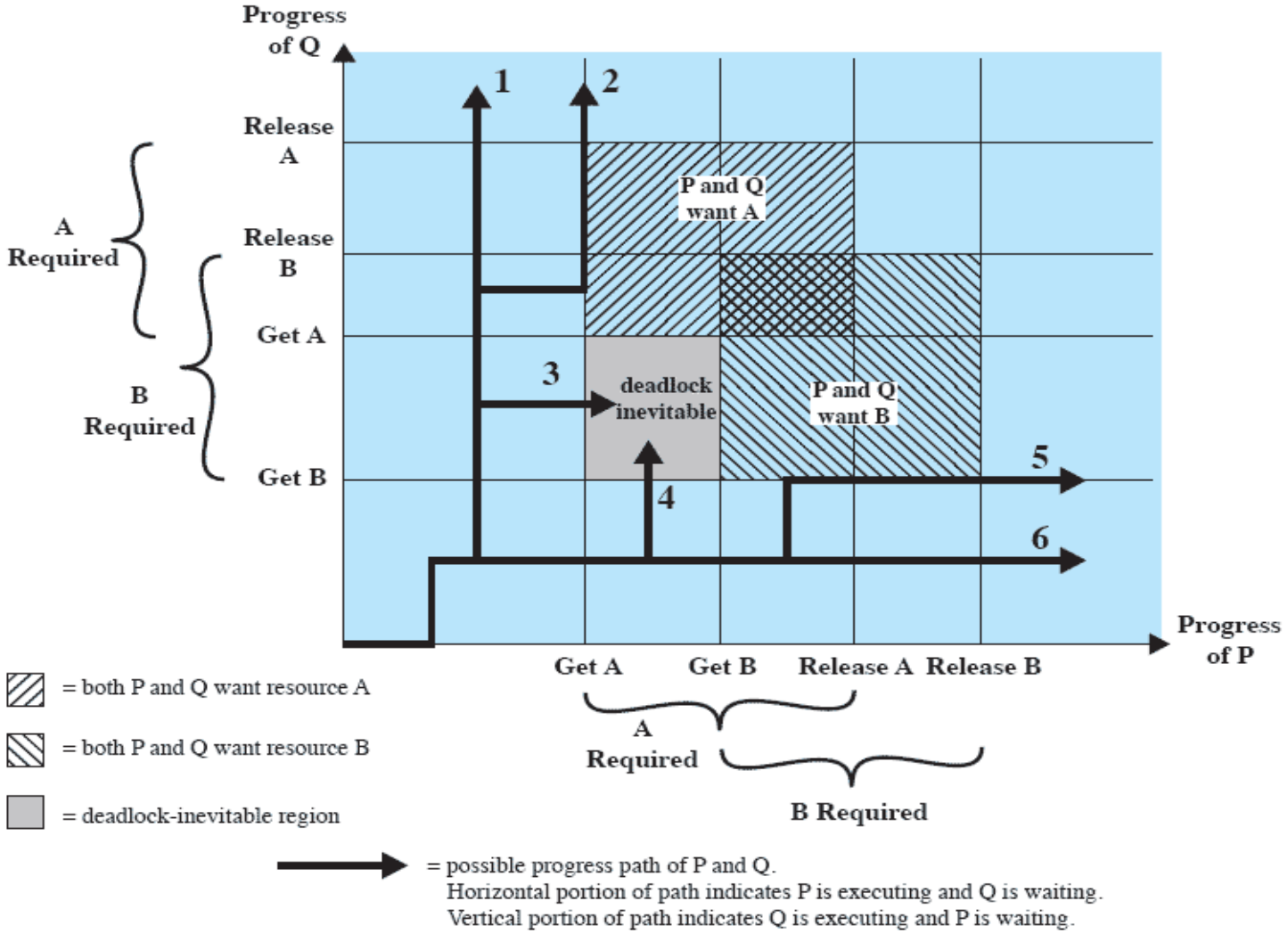


Figure 6.2 Example of Deadlock

Example of Deadlocks

The figure shows six different execution paths. These can be summarized as follows:

1. Q acquires B and then A and then releases B and A. When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and then A. P executes and blocks on a request for A. Q releases B and A. When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires A. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
4. P acquires A and then Q acquires B. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
5. P acquires A and then B. Q executes and blocks on a request for B. P releases A and B. When Q resumes execution, it will be able to acquire both resources.
6. P acquires A and then B and then releases A and B. When Q resumes execution, it will be able to acquire both resources.

Example of Deadlocks

- The gray-shaded area of Figure can be referred to as a fatal region.
  - applies to the commentary on paths 3 and 4.
  - If an execution path enters this fatal region,then deadlock is inevitable
  - The existence of a fatal region depends on the logic of the two processes.
  - However, deadlock is only inevitable if the joint progress of the two processes creates a path that enters the fatal region.

Example of Deadlocks

- Whether or not deadlock occurs depends on both the dynamics of the execution and on the details of the application.
- suppose that P does not need both resources at the same time so that the two processes have the following form:

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Example of Deadlocks

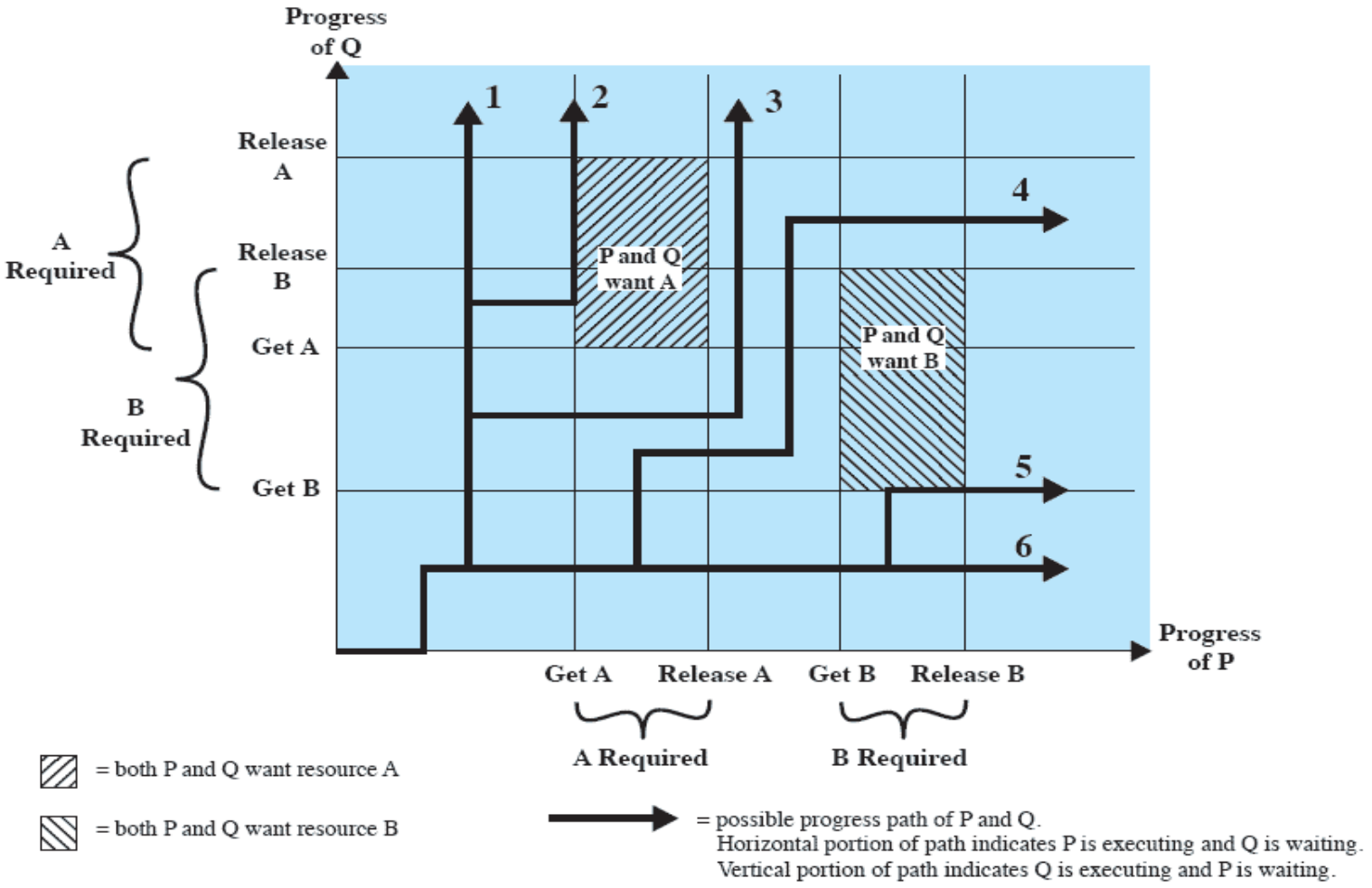


Figure 6.3 Example of No Deadlock [BACO03]

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other.

- For example, deadlock occurs if the multiprogramming system interleaves the execution of the two processes as follows: p0 p1 q0 q1 p2 q2

Reusable Resources

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

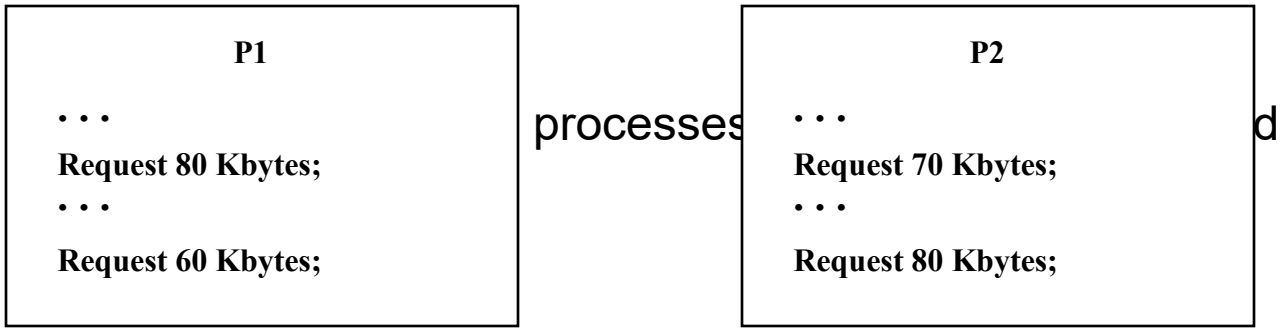
Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Reusable Resources

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

- 

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of Deadlock

- Deadlock occurs if receives blocking

P1

```
. . .
Receive(P2);
. . .
Send(P2, M1);
```

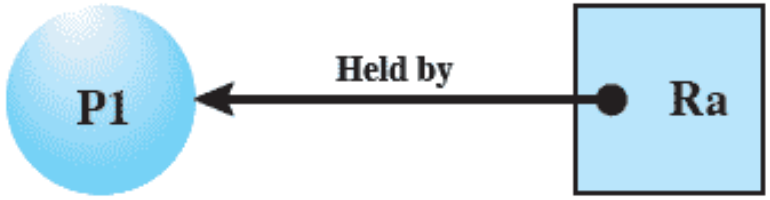
P2

```
. . .
Receive(P1);
. . .
Send(P1, M2);
```

Resource Allocation Graphs

- A useful tool in characterizing the allocation of resources to processes is the resource allocation graph, introduced by Holt .
- Directed graph that depicts a state of the system of resources and processes

(a) Resouce is requested



(b) Resource is held

Example of Resource Allocation Graph

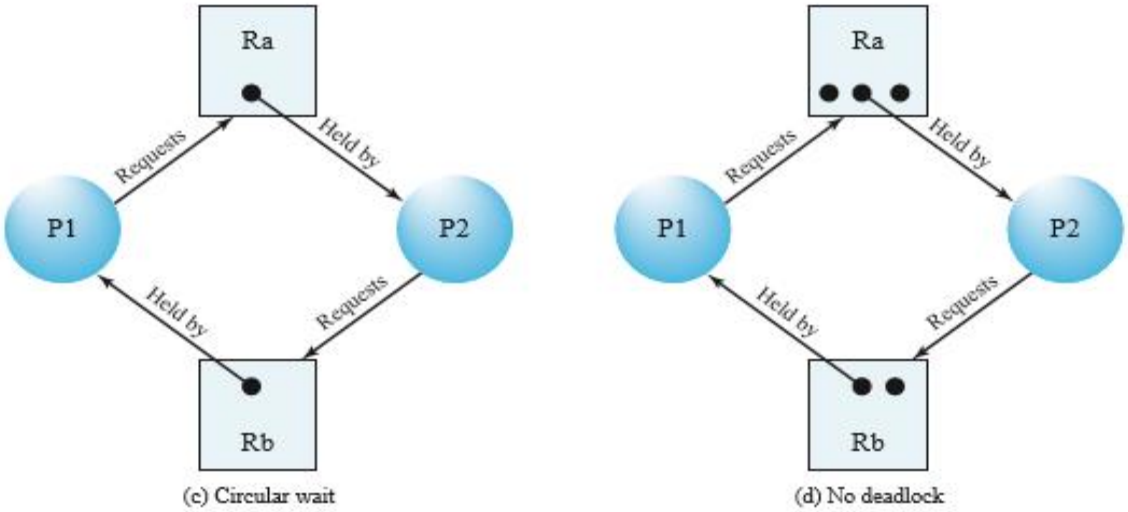


Figure 6.5 Examples of Resource Allocation Graphs

Example of Resource Allocation Graph

- The resource allocation graph corresponds to the deadlock situation.
- In this case, there is a circular chain of processes and resources that results in deadlock.

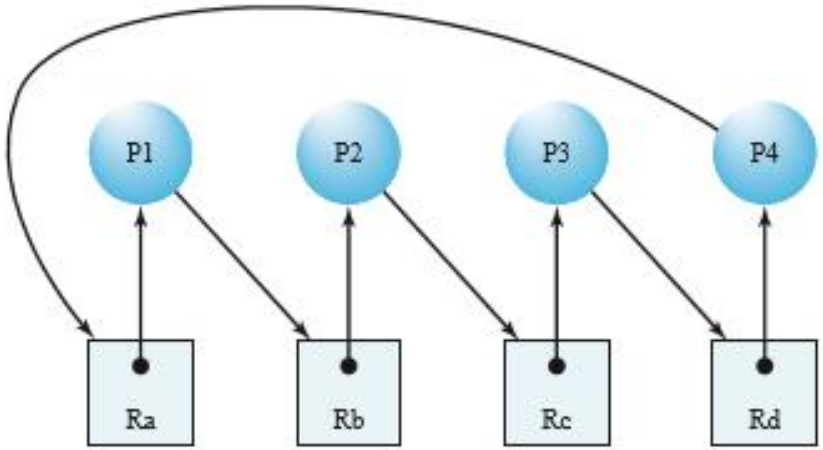


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others

Conditions for Deadlock

- No preemption
  - No resource can be forcibly removed form a process holding it
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

- The first three conditions are necessary but not sufficient for a deadlock to exist.For deadlock to actually take place,a fourth condition is required:

Existence of Deadlock

- The four conditions, taken together, constitute necessary and sufficient conditions for deadlock
  - Mutual Exclusion
  - No preemption
  - Hold and wait
  - Circular wait

- The fourth condition is, actually, a potential consequence of the first three.
- That is, given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait.
- The unresolvable circular wait is in fact the definition of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - **Deadlock prevention**
  - **Deadlock avoidance**
- Allow the system to enter a deadlock state and then recover(**Detection**)

Deadlock prevention

- The strategy of deadlock prevention is simply put, to design a system in such a way that the possibility of deadlock is excluded
- deadlock prevention methods as falling into two classes:
- An **indirect method** of deadlock prevention is to prevent the occurrence of one of the three necessary conditions
- A direct method of deadlock prevention is to prevent the occurrence of a circular wait

Deadlock Prevention

- Mutual Exclusion
  - If access to a resource requires mutual exclusion, then it must be supported by the OS.
  - not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tap drive are inherently non-shareable
- Hold and Wait
  - Require a process request all of its required resources at one time
  - and blocking the process until all requests can be granted simultaneously.
  - Two Problems
  - First, a process may be held up for a long time waiting for all of its resource requests to be filled.
  - resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

Deadlock Prevention

- No Preemption
  - if a process holding certain resources is denied a further request, Process must release resource and request them again together with the additional resource
  - Alternatively, if a process requests a resource that is currently held by another process
    - the OS may preempt the second process and require it to release its resources
    - prevent deadlock only if no two processes possessed the same priority.

Deadlock Prevention

Circular wait:

- The circular-wait condition can be prevented by defining a linear ordering of resource types.
- Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.
- Suppose resource Ri precedes Rj in the ordering if i < j.
- Now suppose that two processes A and B, are deadlocked because A has acquired Ri and requested Rj, and B has acquired Rj and requested Ri.
- This condition is impossible because it implies i < j and j < i.

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests
- Deadlock avoidance ,on the other hand allows the three necessary conditions
  - but makes judicious choices to assure that the deadlock point is never reached.
  - avoidance allows more concurrency than prevention.

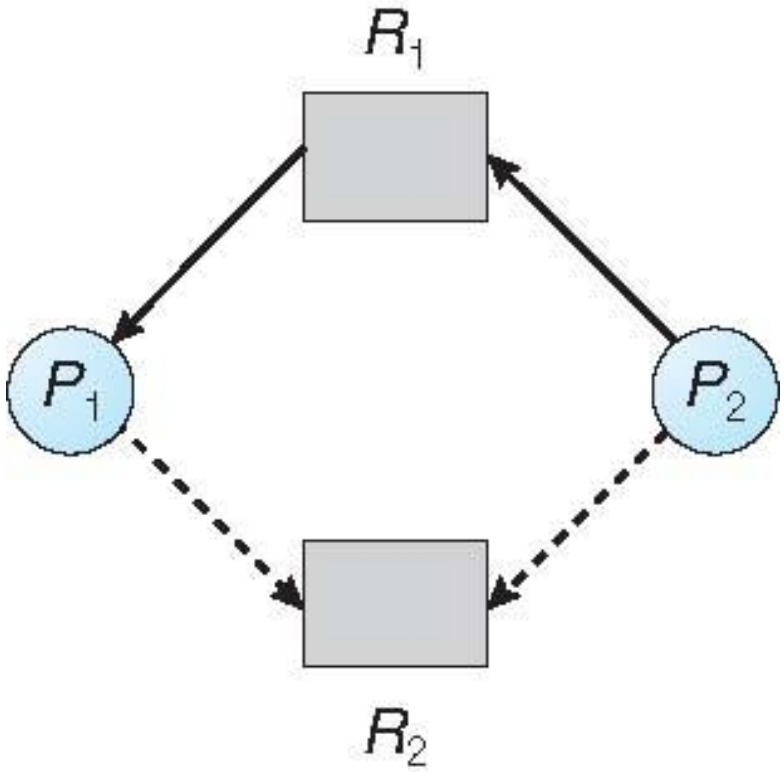
Deadlock Avoidance

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

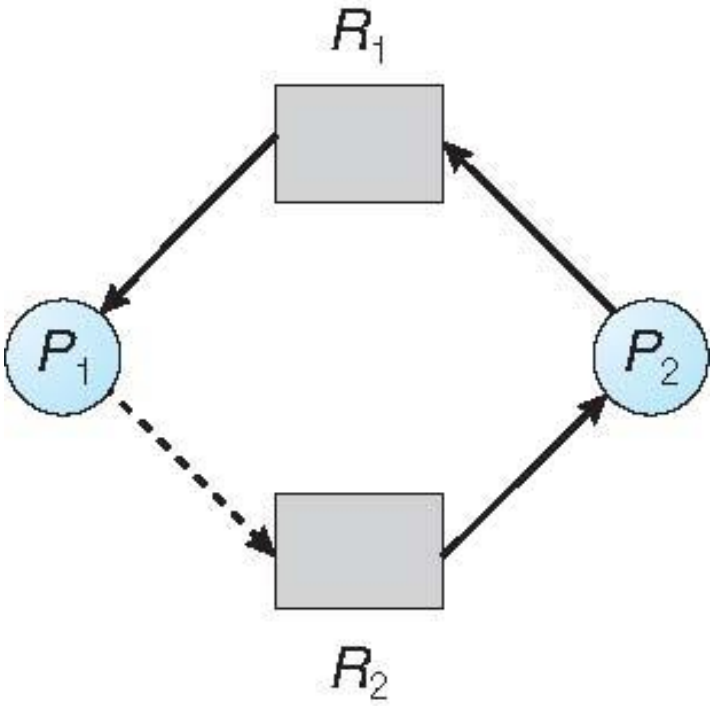
Resource-Allocation Graph Scheme

- **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



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

Deadlock Avoidance(Multiple Instances)

- Do not start a process if its demands might lead to deadlock
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Process Initiation Denial

- Consider a system of n processes and m different types of resources.

[Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

Process Initiation Denial

- The following relationships hold:

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j	All resources are either available or allocated.
2. $C_{ij} \leq R_j$, for all i, j	No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i, j	No process is allocated more resources of any type than the process originally claimed to need.

- we can define a deadlock avoidance policy that refuses to start a new process if its resource requirements might lead to deadlock.
- Start a new process Pn+1 only if $R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$ for all j

A process is only started if the maximum claim of all current processes plus those of the new process can be met.

- strategy is hardly optimal because all processes will make their maximum claims together.

Resource Allocation Denial

- Referred to as the banker's algorithm
- **State** of the system is the current allocation of resources to process
- The state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation
- **Safe state** is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe

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 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

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

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.

Determination of an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

	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 (Cont.)

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

	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

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

	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	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement

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

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

Deadlock Avoidance

- Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection,
- and is less restrictive than deadlock prevention.
- It does have a number of restrictions on its use:
  - Maximum resource requirement must be stated in advance
  - Processes under consideration must be independent; no synchronization requirements
  - There must be a fixed number of resources to allocate

Deadlock Detection

- With deadlock detection, requested resources are granted to processes whenever possible.
- Periodically, the OS performs an algorithm that allows it to detect the circular wait condition
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

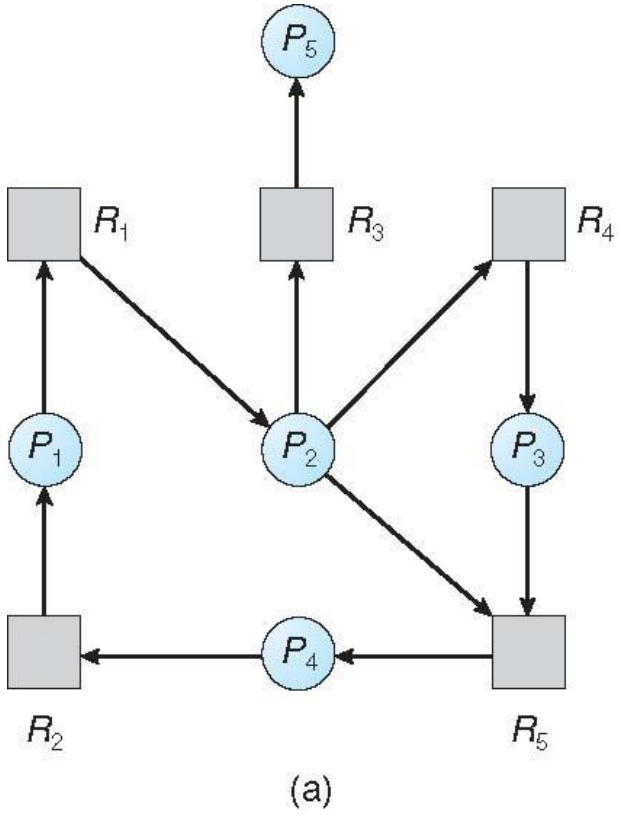
Single Instance of Each Resource Type

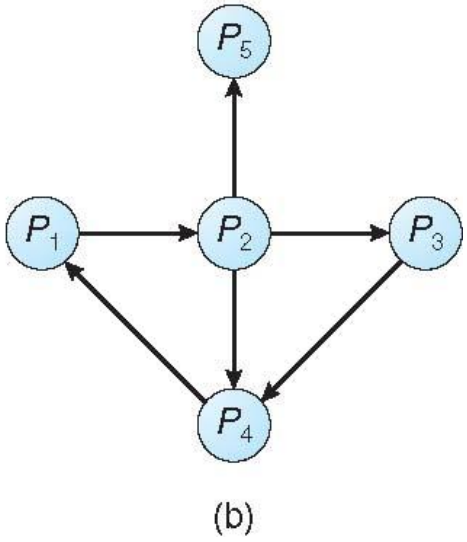
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- 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, 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, …, *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 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 <P_0, P_2, P_3, P_1, P_4> 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 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

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- 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.

Deadlock Detection

- A check for deadlock can be made as frequently as each resource request or, less frequently ,depending on how likely it is for a deadlock to occur.
- .Checking at each resource request has two advantages:
  - it leads to early detection
  - the algorithm is relatively simple because it is based on incremental changes to the state of the system.
  - such frequent checks consume considerable processor time.

Deadlock Recovery

- Once deadlock has been detected, some strategy is needed for recovery.
- The following are possible approaches, listed in order of increasing sophistication:

1. Abort all deadlocked processes. This is one of the most common, solution adopted in operating systems.
2. Back up each deadlocked process to some previously defined checkpoint, and restart all processes
   - This requires that rollback and restart mechanisms be built in to the system.
   - original deadlock may recur.

Deadlock Recovery

3. Successively abort deadlocked processes until deadlock no longer exists.
   - The order in which processes are selected for abortion should be on the basis of some criterion of minimum cost.
   - After each abortion, the detection algorithm must be re-invoked to see whether deadlock still exists.
4. Successively preempt resources until deadlock no longer exists.
   - A cost based selection should be used.
   - Re-invocation of the detection algorithm is required after each preemption.
   - A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

Deadlock Recovery

For (3) and (4),the selection criteria could be one of the following.

Choose the process with the :

- least amount of processor time consumed so far
- least amount of output produced so far
- most estimated time remaining
- least total resources allocated so far
- lowest priority

Deadlock Recovery

- **Starvation** – same process may always be picked as victim
  - include number of rollback in cost factor

End of Chapter 7