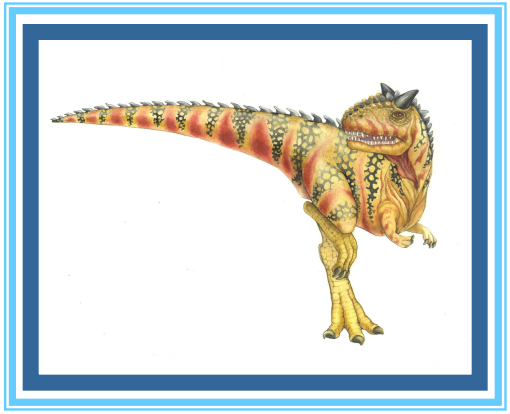
Chapter 7: Deadlocks
Lecture #11



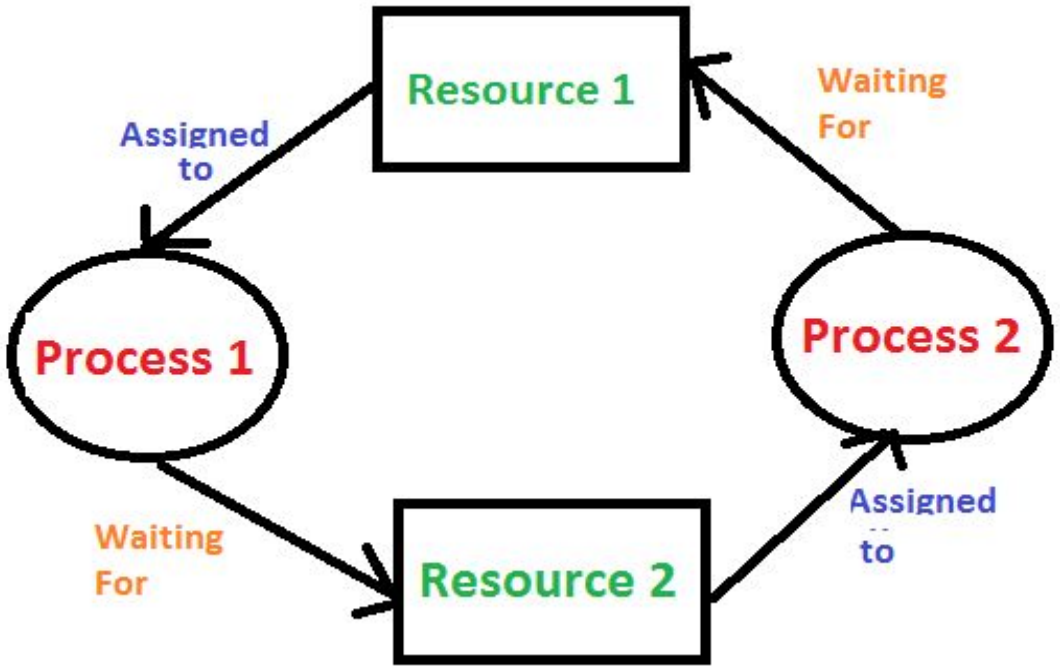
Chapter 7: Deadlocks

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

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**



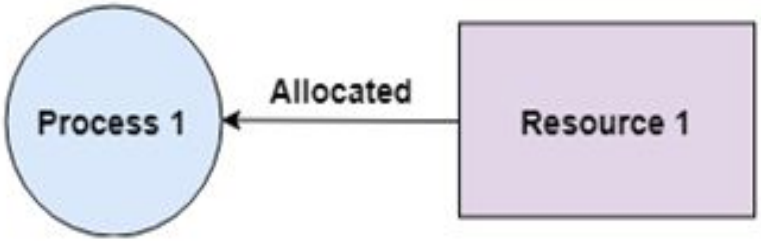
Example of Deadlock in OS

- A deadlock involving same resource type:
  - To illustrate a deadlocked state, consider a system with three CD RW drives.
  - Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event “CD RW is released,” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.
- A deadlock involving different resource types:
  - Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

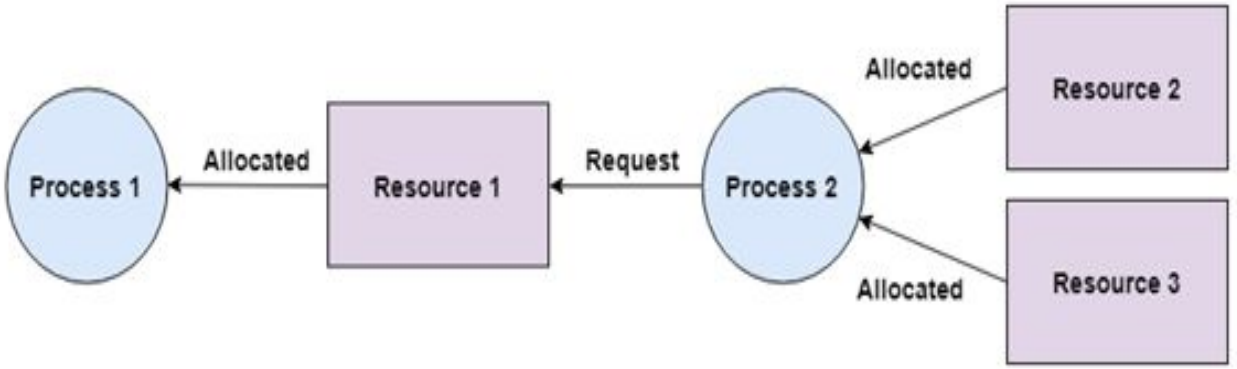
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

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

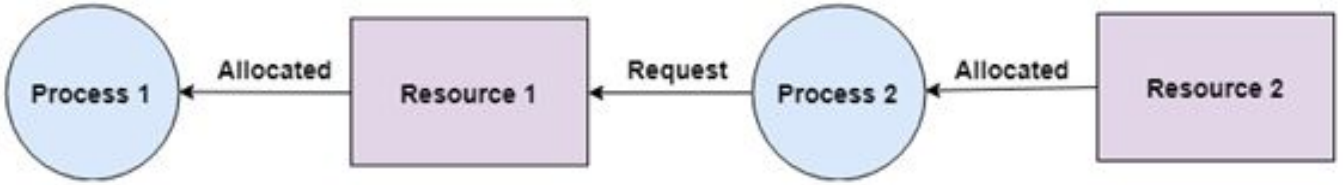


- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes

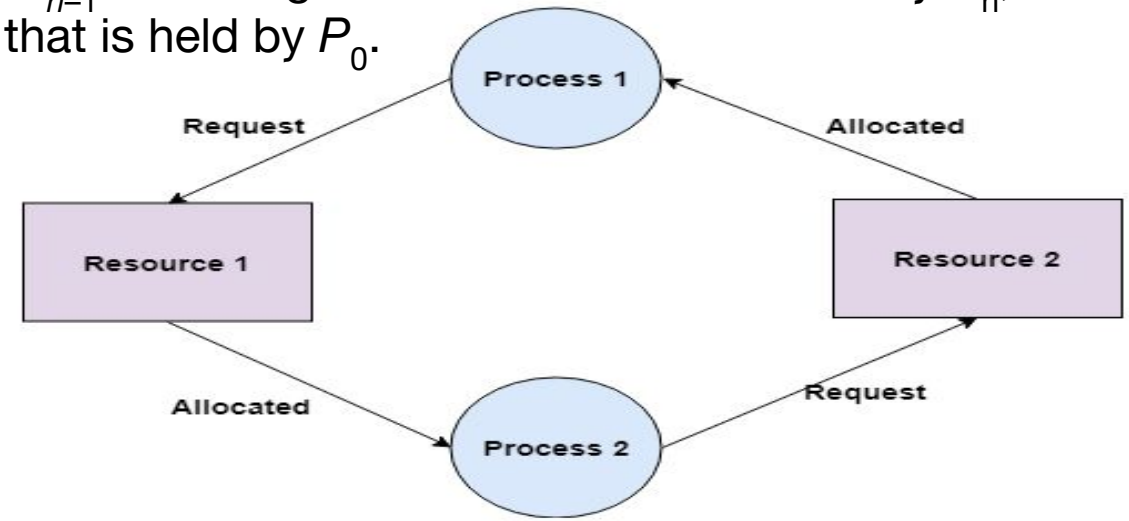


Deadlock Characterization

- **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 visual (mathematical) way to determine if a deadlock has or may occur.

  G = (V, E) The graph contains nodes and edges.

  V Nodes consist of processes = { P1, P2, P3, ...} and resource types { R1, R2, ...}

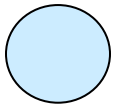
  E Edges are (Pi, Rj) or (Ri, Pj)

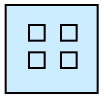
- An arrow from the **process** to **resource** indicates the process is **requesting** the resource. An arrow from **resource** to **process** shows an instance of the resource has been **allocated** to the process.

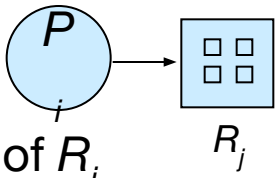
- **request edge** – directed edge $P_i \rightarrow R_j$

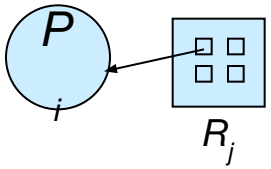
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

- Process 

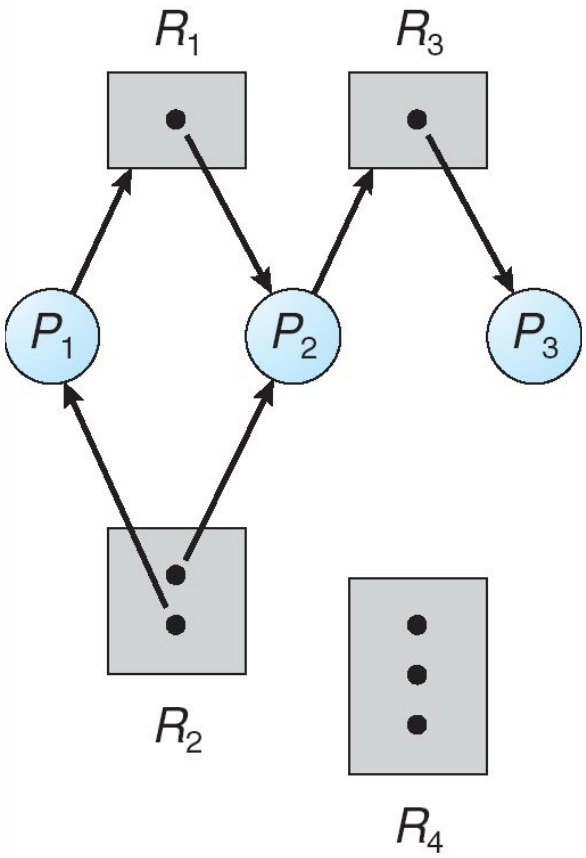
- Resource Type with 4 instances 

- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Process is a circle; resource type is square; dots represent number of instances of resource in type. Request points to square, assignment comes from dot.

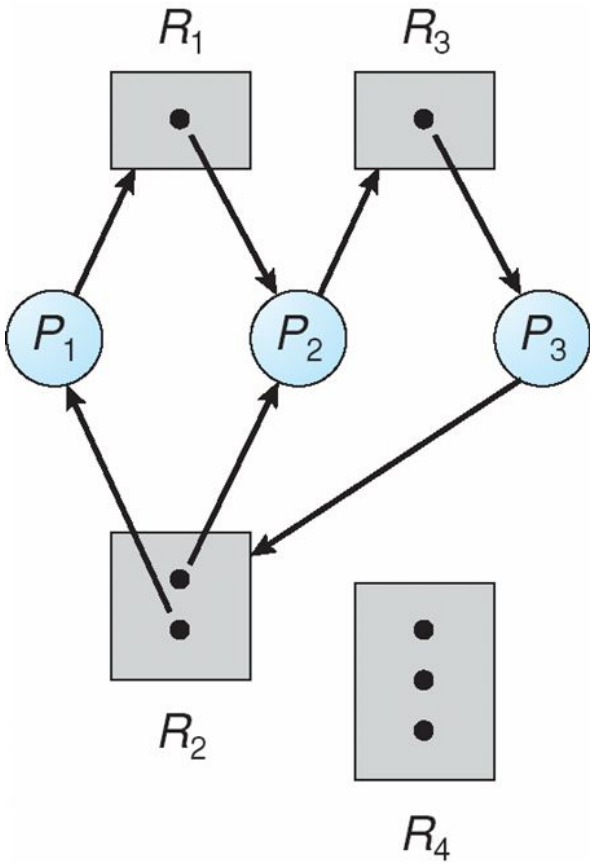
Example of a Resource Allocation Graph



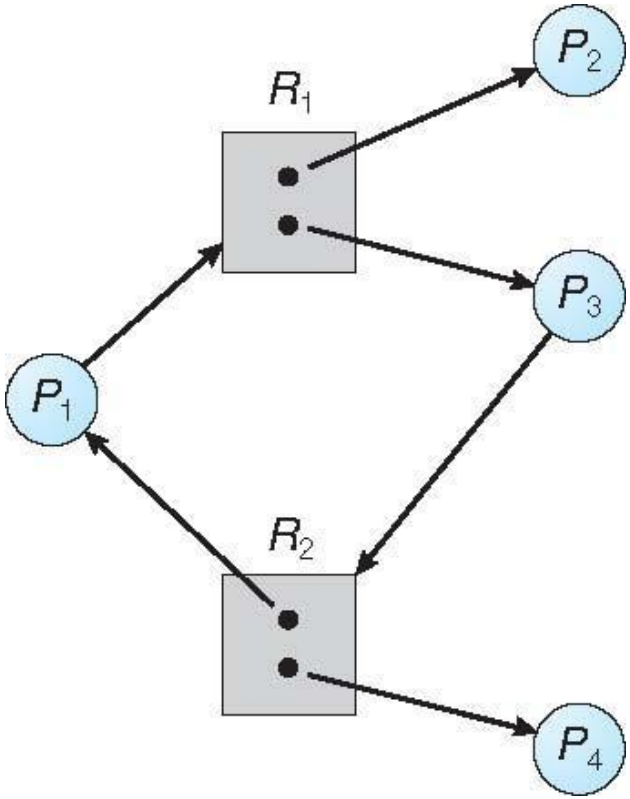
The resource-allocation graph shown in depicts the following situation. • The sets P, R, and E: ◦

- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



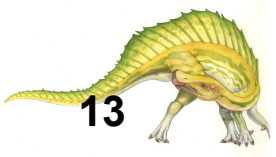
Basic Facts

- 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

HOW TO HANDLE DEADLOCKS – GENERAL STRATEGIES

There are three methods: **Strategy**

- Ignore Deadlocks: ← **Most Operating systems do this!!**
- Ensure deadlock **never** occurs using either
  - **Prevention** : Prevent any one of the 4 conditions from happening.
  - **Avoidance:** Allow all deadlock conditions but calculate cycles about to happen and stop dangerous operations..
- **Allow** deadlock to happen. This requires using both:
  - **Detection:** Know a deadlock has occurred.
  - **Recovery** : Regain the resources.



Deadlock Prevention

Deadlock Prevention

Try to prevent any one of the four necessary conditions for a deadlock

- Restrain the ways request can be made

① Mutual Exclusion
② Hold & Wait
③ Circular wait
④ Non preemptive resources

Deadlock Prevention

Violating mutual exclusion

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

Mutual Exclusion = While a process is using a resource no other process can use the same resource

Problem

- Some resources inherently require mutual exclusion
  - Printers
  - Files for updates
  - CD-Rs

Deadlock Prevention

Violating hold & wait

① Require all processes to acquire all the needed resources at once

(a) Resource request does not have to be at the beginning of a process

(b) If any of the required resource is not available, drop everything (then try it again later)

(c) Any process can not make a request for any new resource, while the process currently holds some resource

② A process does NOT have to request all the resources at once

(a) Whenever an additional resource is needed, drop all what are possessed by this process and request everything at once

(b) If any process steals your resource(s) when you temporarily have to wait until all resources become available

Deadlock Prevention (Cont.)

Violating non-preemption

"Attacking Non-Preemption"

= Allow preemption

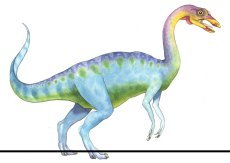
= You process can loose your resource at any time

- 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 resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Problem

- The same problem we had for "mutual exclusion"

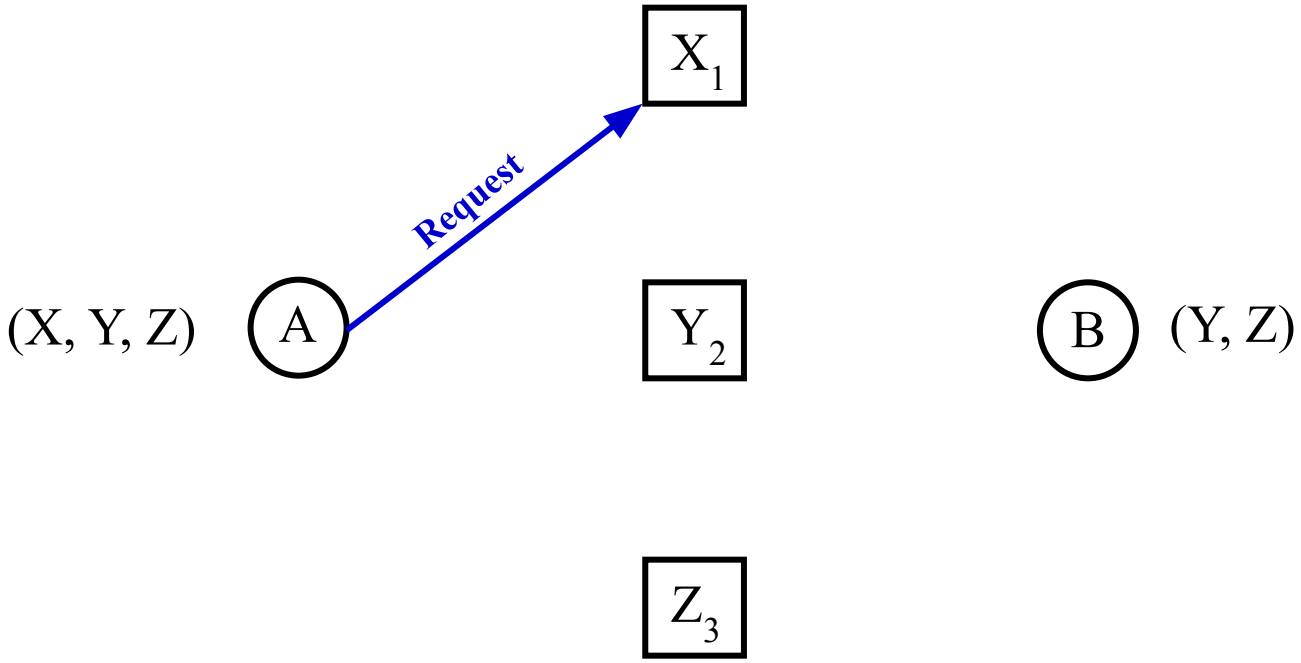
➡ No one will be happy in loosing a CD-R drive in the middle of burning a CD



Deadlock Prevention (Cont.)

Violating Circular-Wait

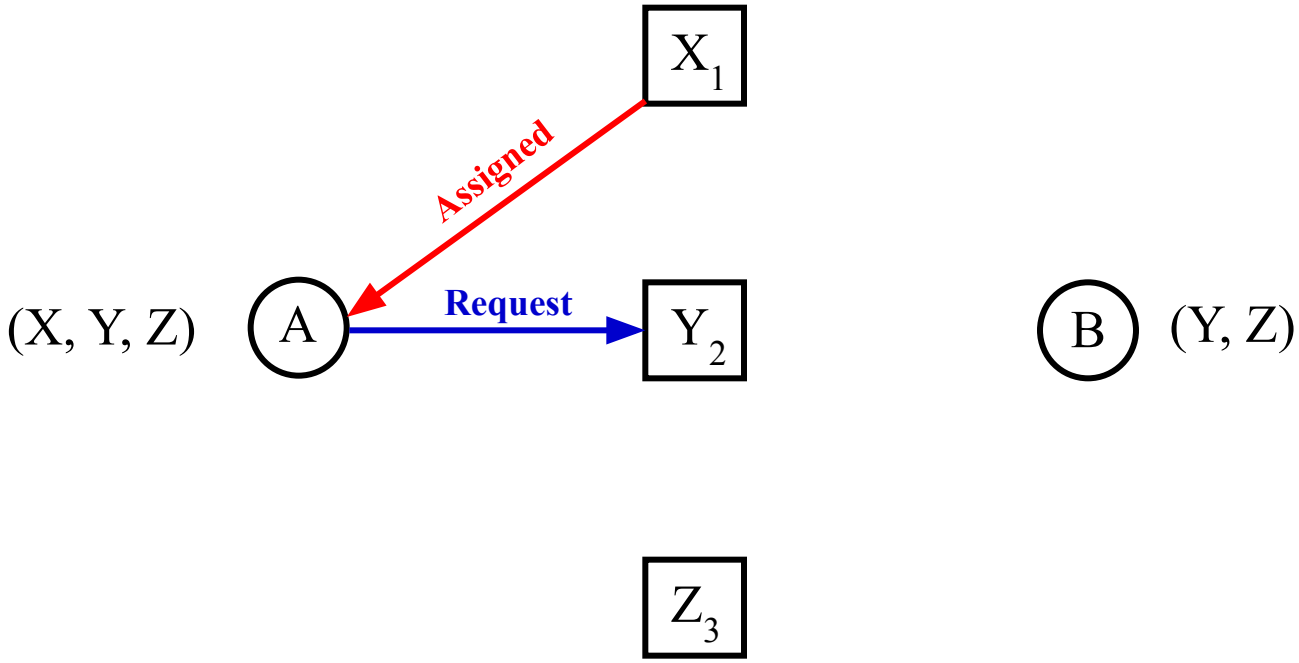
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

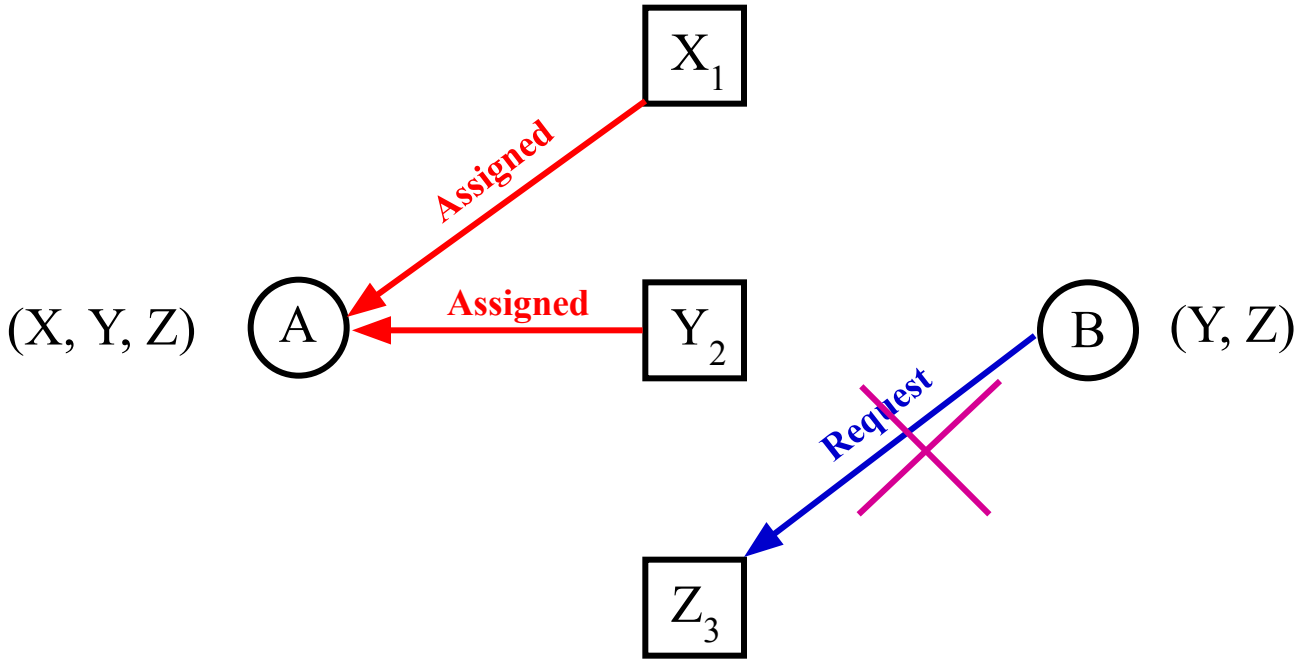
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

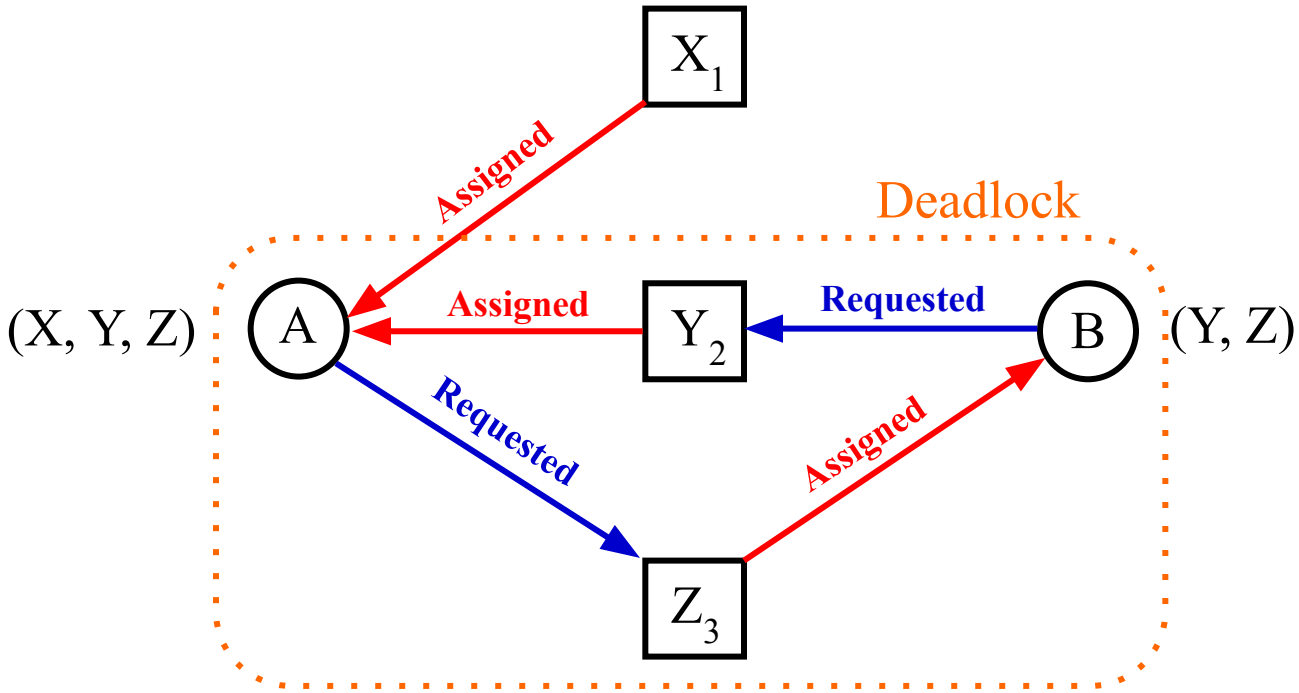
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

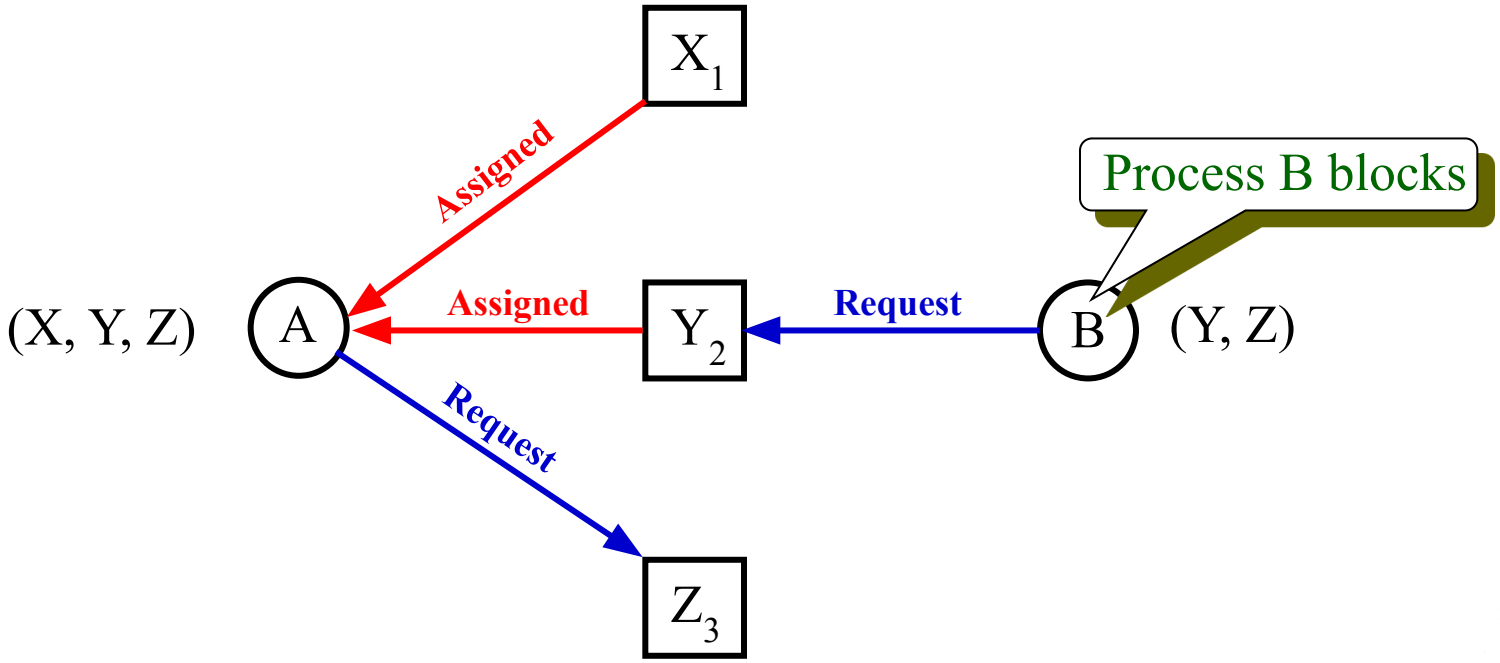
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

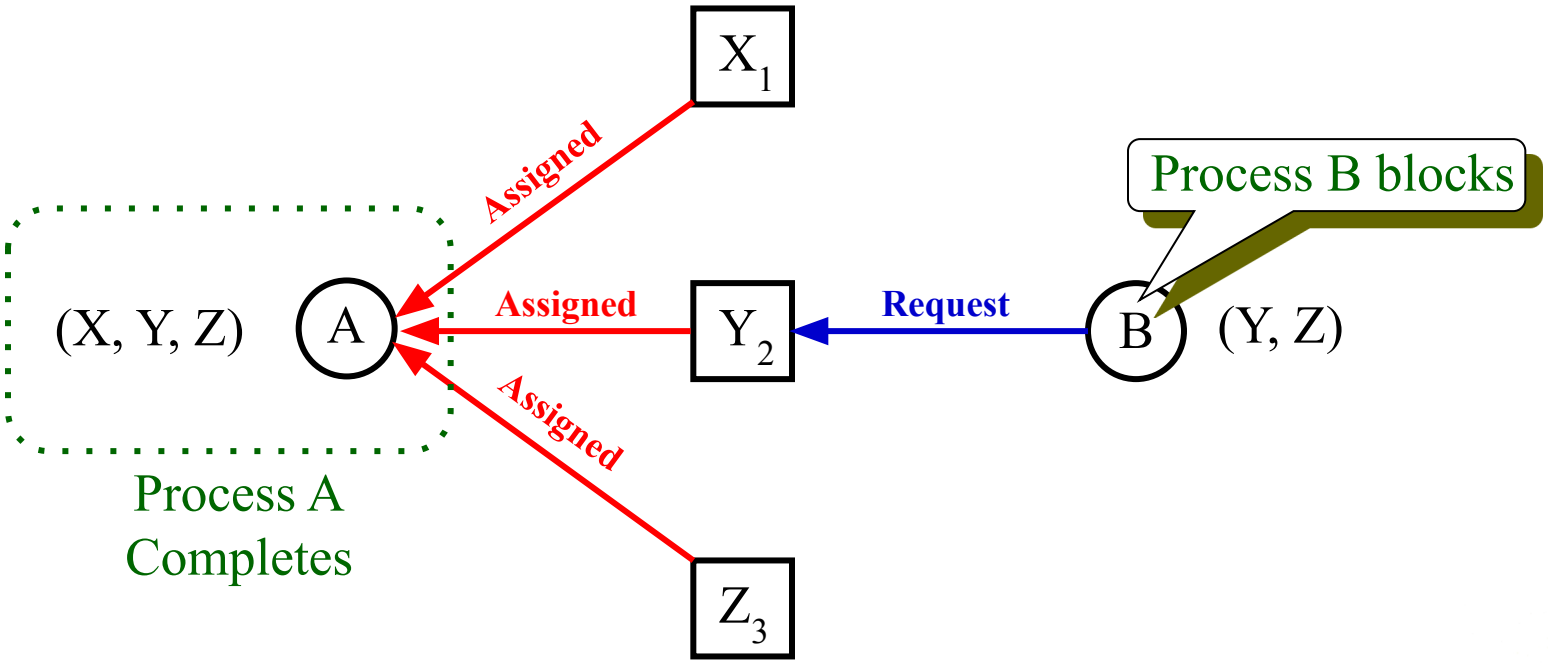
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

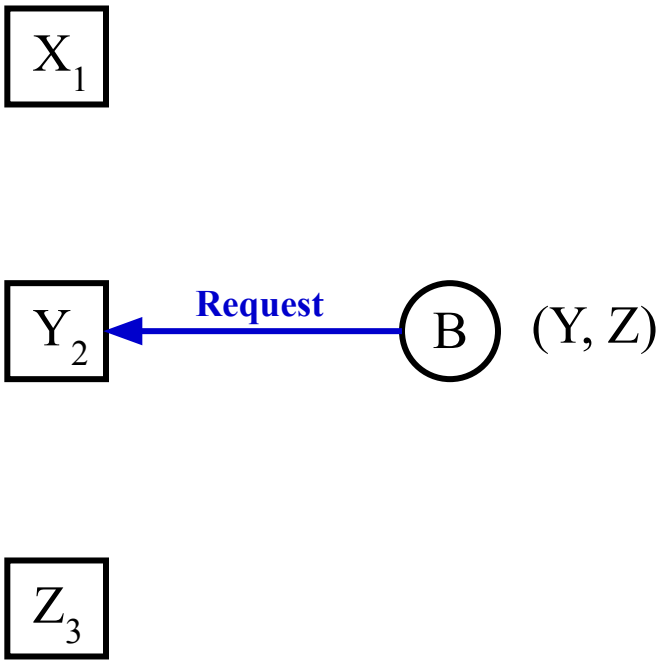
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

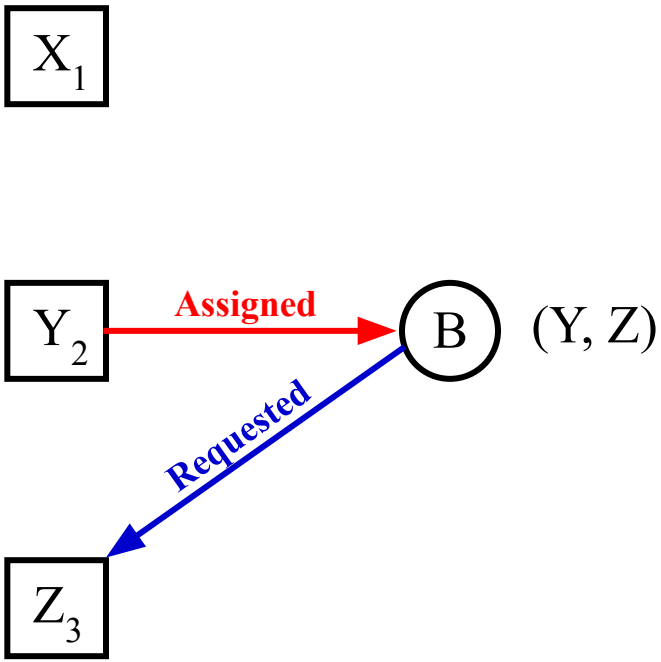
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

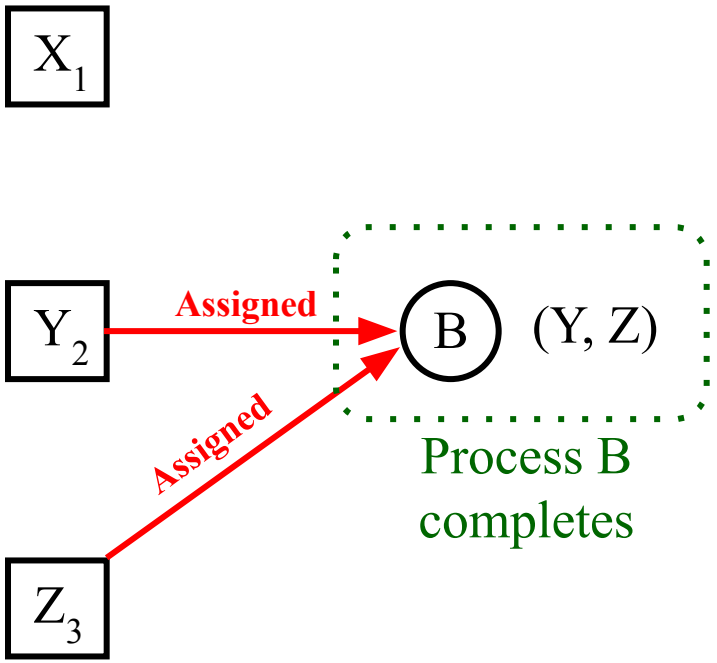
- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



Deadlock Prevention (Cont.)

Violating Circular-Wait

- Assign unique ID for every non-sharable resource in a system
- Every process must start requesting resources in either ascending or descending order of resource IDs



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

Deadlock Avoidance

The two states defined in Deadlock Avoidance

① Safe State

A system is in "safe state" if there is at least one assignment schedule to complete all the processes without deadlock and without any process releasing their resources

(deadlock will never happen even in the worst case)

⇨ If a system is in safe state, deadlock can always be avoided

② Unsafe State

If a system is not in safe state, the system must be in "unsafe state"

(deadlock can happen in the worst case)

⇨ If a system is in unsafe state, deadlock may not be avoided

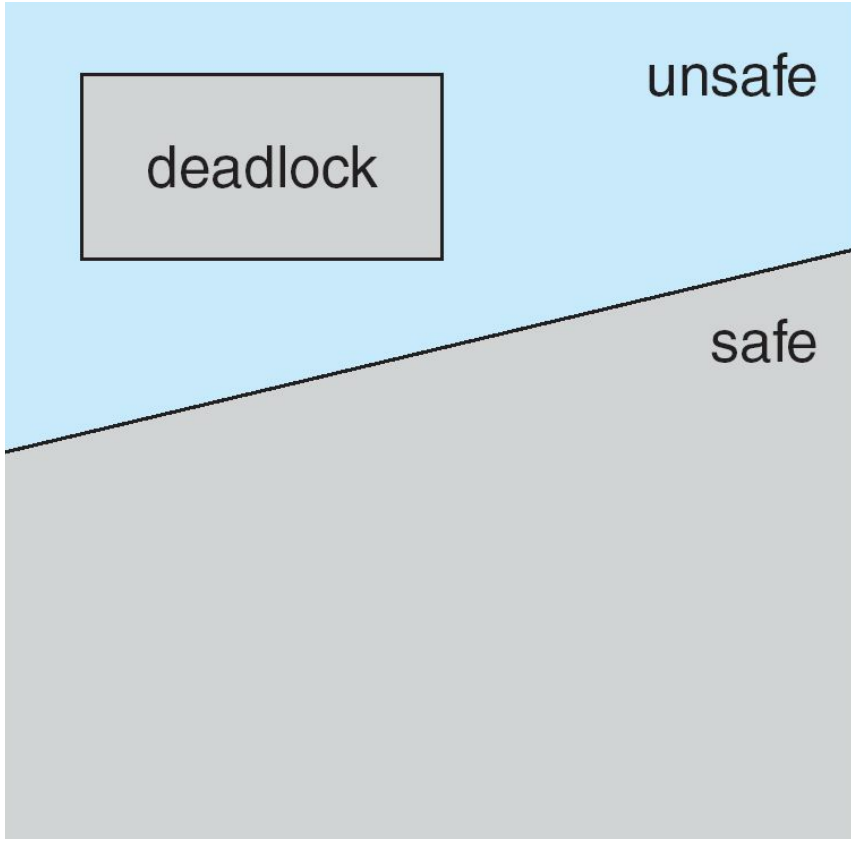
Safe State

- 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

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



Deadlock Avoidance

Process	Assigned	MAX. Need
A	3	9
B	2	4
C	2	7

- If a process gets all required resources then, the process becomes COMPLTED

Requests Pending

The number of requests a process made, but not granted because OS does not have the requested resources

Process Status

- If a process has at least one pending request, process is BLOCKED
- Otherwise a process is RUNNING

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available

4

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

But Not Deadlock yet

If Process A requests 4 more instances of this resource

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	4	BLOCKED
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Deadlock = all (3) processes blocked waiting for additional resource

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

If Process A requests 4 more instances of this resource

Unsafe state = no one can complete if no one releases resource

This is "unsafe state"

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3 7	9	0 4	RUNNING BLOCKED
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 0 4

But Not Deadlock yet

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

If Process A requests 1 more instances of this resource

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	7	9	1	BLOCKED
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 0

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

If Process B requests 2 more instances of this resource

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	7	9	1	BLOCKED
B	2	4	2	BLOCKED
C	2	7	0	RUNNING

Available: 0

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

Now Deadlock

If Process C requests 2 more instances of this source

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	7	9	1	BLOCKED
B	2	4	2	BLOCKED
C	2	7	2	BLOCKED

Available: 0

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

This is "safe state"

If Process B requests 2 more instances of this resource

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	4	4	0	RUNNING
C	2	7	0	RUNNING

Available: 2

Deadlock Avoidance

Assume: 11 instances of the resource R exist

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	2	4	0	RUNNING
C	2	7	0	RUNNING

Available: 4

This is "safe state"

If Process B requests 2 more instances of this resource

Process	Assigned	MAX. Need	Requests Pending	Process Status
A	3	9	0	RUNNING
B	0	0	0	COMPLETED
C	2	7	0	RUNNING

Available: 6

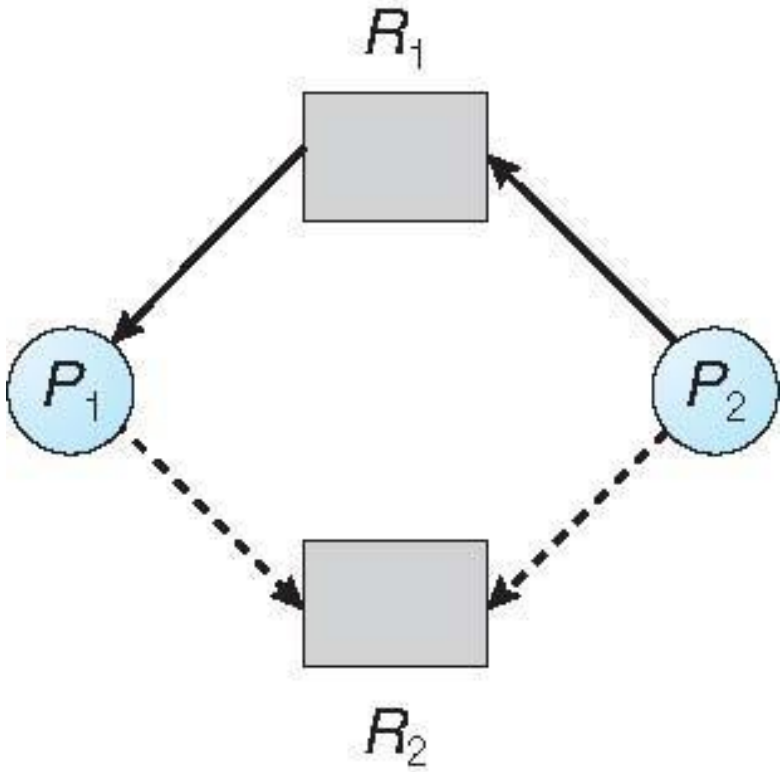
Avoidance Algorithms

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

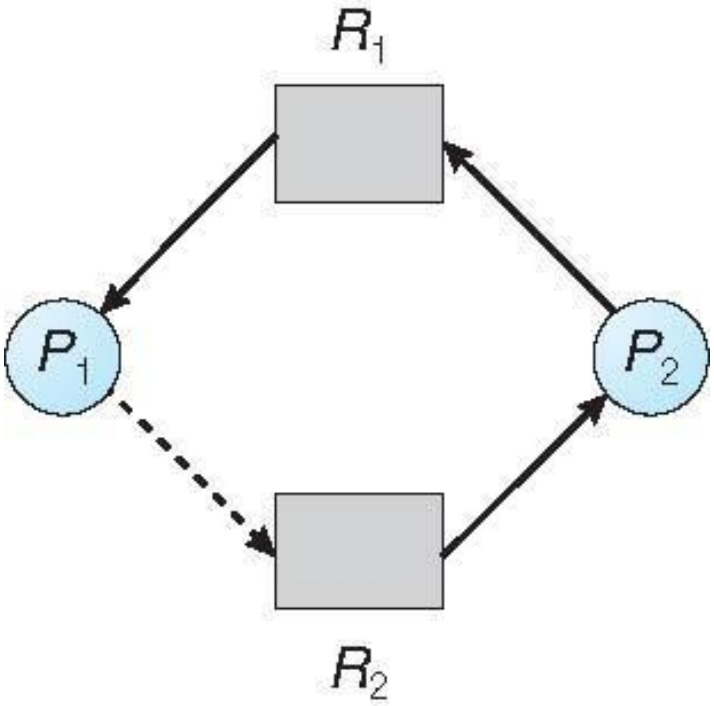
Resource-Allocation Graph Scheme

- **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**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

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

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

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

Example of Banker's Algorithm

Safe Sequence < >

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	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

1. **Work = Available**
2. **Finish[i] = False**
   if *Need* ≤ *Work*
   If no such ***i*** exists, go to step 4
3. ***Work = Work +Allocation***
4. **Finish[i] = True**

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F					

Example of Banker's Algorithm

Safe Sequence < >

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	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

1. Work = **3 3 2**
2. Finish[P_0] = False
   if *Need* ≤ *Work*
   7 4 3 ≤ **3 3 2**
   Condition is False
3. *Work = Work + Allocation*
4. Finish[i] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4	P_5
T/F	F					

Example of Banker's Algorithm

Safe Sequence < P_1 >

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	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

1. Work = 3 3 2
2. Finish[P_1] = False
   if *Need* ≤ *Work*
   1 2 2 ≤ 3 3 2
   Condition is TRUE
3. *Work = Work + Allocation*
   3 3 2 + 2 0 0=5 3 2
4. Finish[P_1] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4	P_5
T/F	F	T				

Example of Banker's Algorithm

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	5 3 2	7 4 3
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

Safe Sequence < P_1 >

1. Work = 5 3 2
2. Finish[0] = False
if *Need* ≤ *Work*
***6 0 0* ≤ 3 3 2**
Condition is FALSE
3. Work = Work + Allocation
***4.* Finish[i] = True**

Finish[P_i]	P_0	P_1	P_2	P_3	P_4	P_5
T/F	F	T	F			

Example of Banker's Algorithm

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	5 3 2	7 4 3
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

Safe Sequence < P_1, P_3, >

1. Work = 5 3 2
2. Finish[P_3] = False
   if *Need* ≤ *Work*
   0 1 1 ≤ 5 3 2
   Condition is TRUE
3. *Work* = *5 3 2 + 2 1 1=7 4 3*
4. Finish[P_3] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	F	T	

Example of Banker's Algorithm

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	7 4 3	7 4 3
P_2	3 0 2	9 0 2		6 0 0
P_4	0 0 2	4 3 3		4 3 1

Safe Sequence < P_1, P_3, P_4 >

1. Work = 7 4 3
2. Finish[P_4] = False
   if *Need* ≤ *Work*
   4 3 1 ≤ 7 4 3
   Condition is TRUE
3. *Work* = 7 4 3 + 0 0 2=7 4 5
4. Finish[P_4] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	F	T	T

Example of Banker's Algorithm

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	7 4 5	7 4 3
P_2	3 0 2	9 0 2		6 0 0

Safe Sequence < P_1, P_3, P_4, P_2 >

1. Work = 7 4 5
2. Finish[P_2] = False
   if *Need* ≤ *Work*
   6 0 0 ≤ 7 4 5
   Condition is TRUE
3. *Work* = 7 4 3+3 0 2=10 4 7
4. Finish[P_2] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	T	T	T

Example of Banker's Algorithm

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

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	10 4 7	7 4 3

1. Work = **10 4 7**
2. Finish[P_0] = False
   if *Need* ≤ *Work*
   7 4 3 ≤ **10 4 7**
   Condition is TRUE
3. *Work*=*10 4 7*+*0 1 0* =*10 5 7*
4. Finish[P_0] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	T	T	T	T	T

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 ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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?

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	2 3 0	7 4 3
P_1	3 0 2	2 0 0		0 2 0
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

Safe Sequence < >

1. Work = Available
2. Finish[i] = False
if *Need* ≤ *Work*
If no such ***i*** exists, go to step 4
3. Work = Work +Allocation
***4.* Finish[i] = True**

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F					

Example (Cont.)

Safe Sequence < >

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	2 3 0	7 4 3
P_1	3 0 2	2 0 0		0 2 0
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

1. Work = 2 3 0
2. Finish[P_0] = False
if *Need* ≤ *Work*
7 4 3 ≤ 2 3 0
Condition is FALSE
3. Work = Work +Allocation
***4.* Finish[i] = True**

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F				

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	2 3 0	7 4 3
P_1	3 0 2	2 0 0		0 2 0
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

Safe Sequence $< P_1 >$

1. Work = 2 3 0
2. Finish[P_1] = False
   if *Need* ≤ *Work*
   0 2 0 ≤ 2 3 0
   Condition is TRUE
3. *Work*=2 3 0 + 3 0 2=5 3 2
4. Finish[P_1] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T			

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	5 3 2	7 4 3
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

Safe Sequence <P_1 >

1. Work = 5 3 2
2. Finish[P_2] = False
   if *Need* ≤ *Work*
   6 0 0 ≤ 5 3 2
   Condition is FALSE
3. *Work= Work + Allocation*
4. Finish[*i*] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	F		

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	5 3 2	7 4 3
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

Safe Sequence <P_1, P_3 >

1. Work = 5 3 2
2. Finish[P_3] = False
if *Need* ≤ *Work*
***0 1 1* ≤ 5 3 2**
Condition is TRUE
3. Work= Work + Allocation
5 3 2 + 2 1 1=7 4 3
***4.* Finish[P_3] = True**

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	F	T	

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	7 4 3	7 4 3
P_2	3 0 2	9 0 2		6 0 0
P_4	0 0 2	4 3 3		4 3 1

Safe Sequence $< P_1, P_3, P_4 >$

1. Work = 7 4 3
2. Finish[P_4] = False
   if *Need* ≤ *Work*
   4 3 1 ≤ 7 4 3
   Condition is TRUE
3. *Work= Work + Allocation*
   7 4 3 + 0 0 2=7 4 5
4. Finish[P_4] = True

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	F	T	T

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	7 4 5	7 4 3
P_2	3 0 2	9 0 2		6 0 0

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

1. Work = 7 4 5
2. Finish[P_2] = False
if *Need* ≤ *Work*
6 0 0 ≤ 7 4 5
Condition is TRUE
3. Work= Work + Allocation
7 4 5 + 3 0 2=10 4 7
***4.* Finish[P_2] = True**

Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	F	T	T	T	T

Example (Cont.)

- n = 5 processes P_0 through P_4;
  m = 3 resource types:
  A (10 instances), *B* (5instances), 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	10 4 7	7 4

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

1. Work = 10 4 7
2. Finish[P_0] = False
   if *Need* ≤ *Work*
   7 4 3 ≤ 10 4 7
   Condition is TRUE
3. *Work= Work + Allocation*
   10 4 7 + 0 1 0=10 5 7
4. Finish[P_0] = True

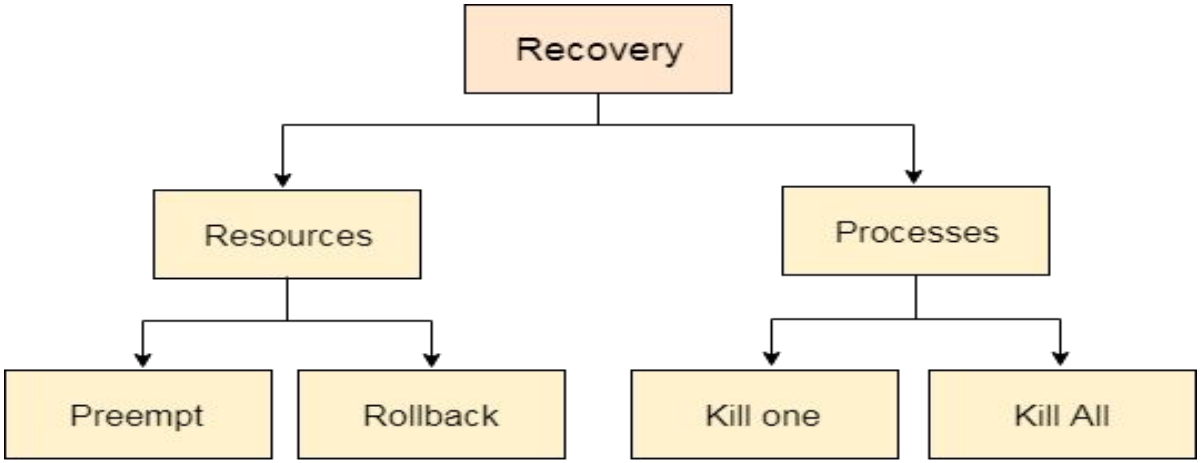
Finish[P_i]	P_0	P_1	P_2	P_3	P_4
T/F	T	T	T	T	T

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.

Recovery from Deadlock: Process Termination

- **Abort all the Deadlocked Processes:**
  Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.
- **Abort one process at a time until deadlock is eliminated:**
  Abort one deadlocked process at a time, until deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we must run deadlock detection algorithm to check whether any processes are still deadlocked.



Recovery from Deadlock: Process Termination

- In which order should we choose to abort?
- Many factors mat affect which process is chosen, including:
  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:**
  We must determine which resources and which processes are to be preempted and the order to minimize the cost.
- **Rollback:**
  We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.
- **Starvation:**
  In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

End of Chapter 7