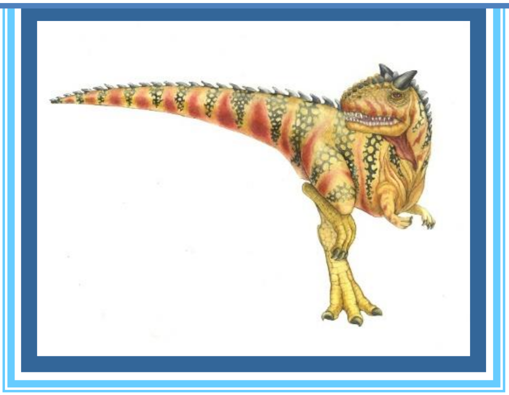
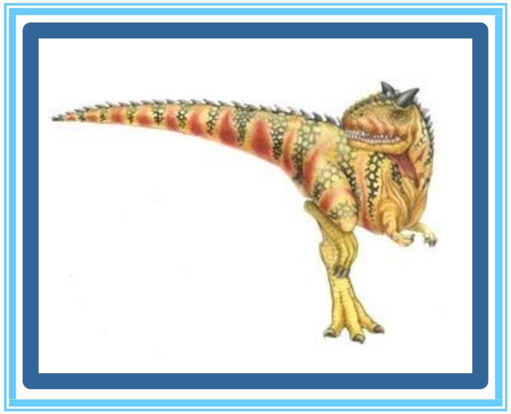
# CSE-309 Operating Systems

Mohammad Shariful Islam
Lecturer, Department of CSE
Mobile: 01747612143
Email: sharifulruhan@gmail.com

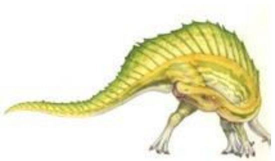


# Chapter 7:  Deadlocks



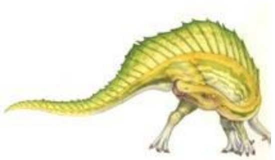
# Chapter 7: Deadlocks

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



# Chapter 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



# Deadlock

- ***Deadlock*** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
- Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other.
- Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).

# Deadlock (cont.)

- For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

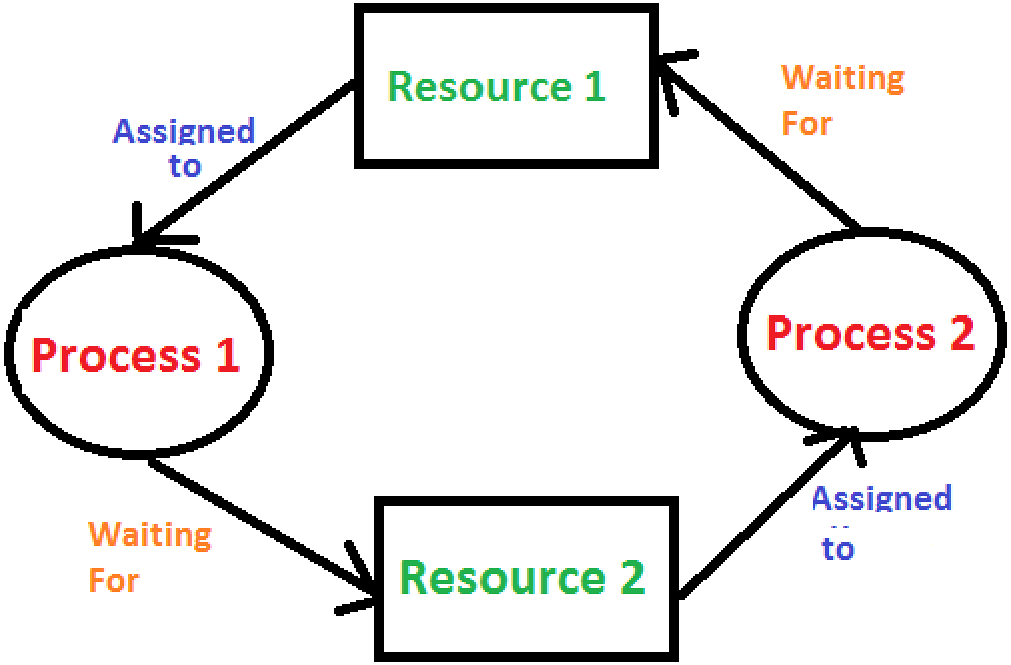
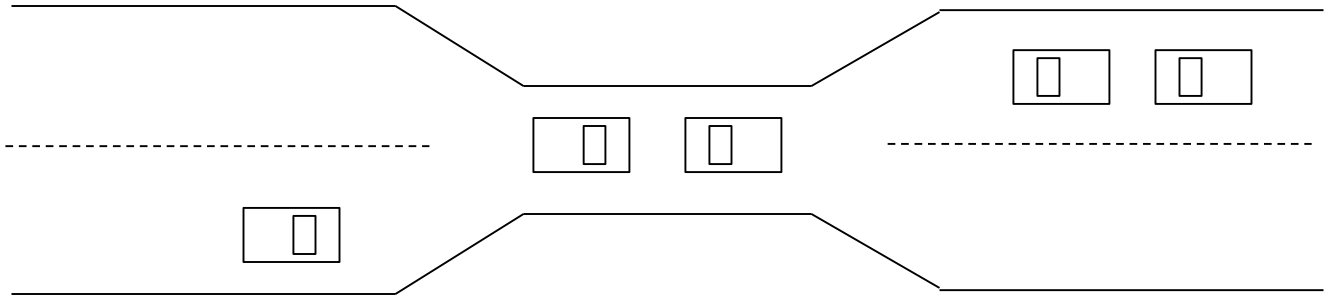


Figure : Deadlock in OS

# Bridge Crossing Example



- Traffic only in one direction like Half duplex data communication.
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

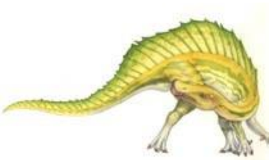
# System Model

- Resource types $R_1, R_2, . . ., R_m$

  *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**

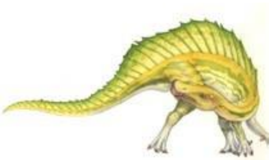


# Deadlock Characterization

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

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



# Resource-Allocation Graph

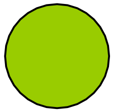
A set of vertices $V$ and a set of edges $E$.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

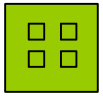
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

# Resource-Allocation Graph (Cont.)

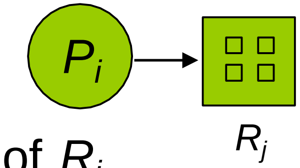
- Process



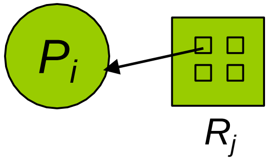
- Resource Type with 4 instances



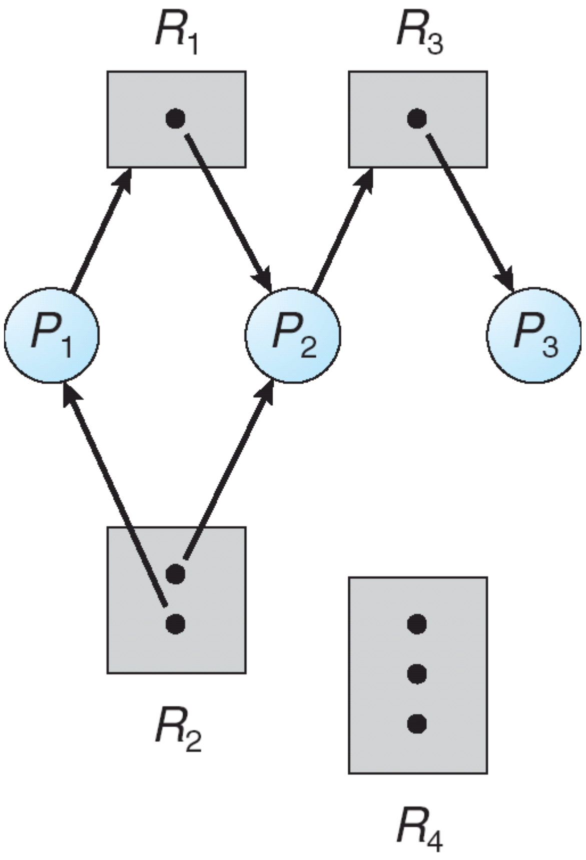
- $P_i$ requests instance of $R_j$



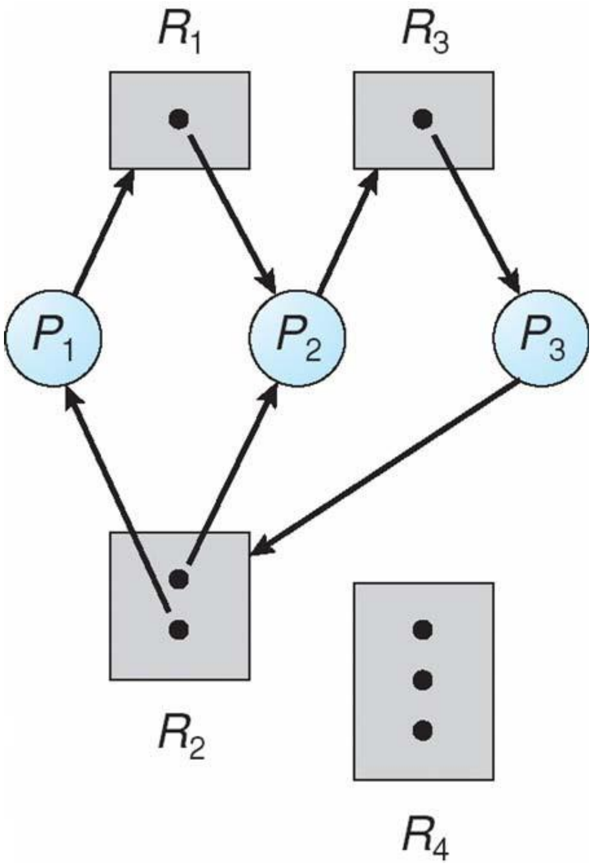
- $P_i$ is holding an instance of $R_j$



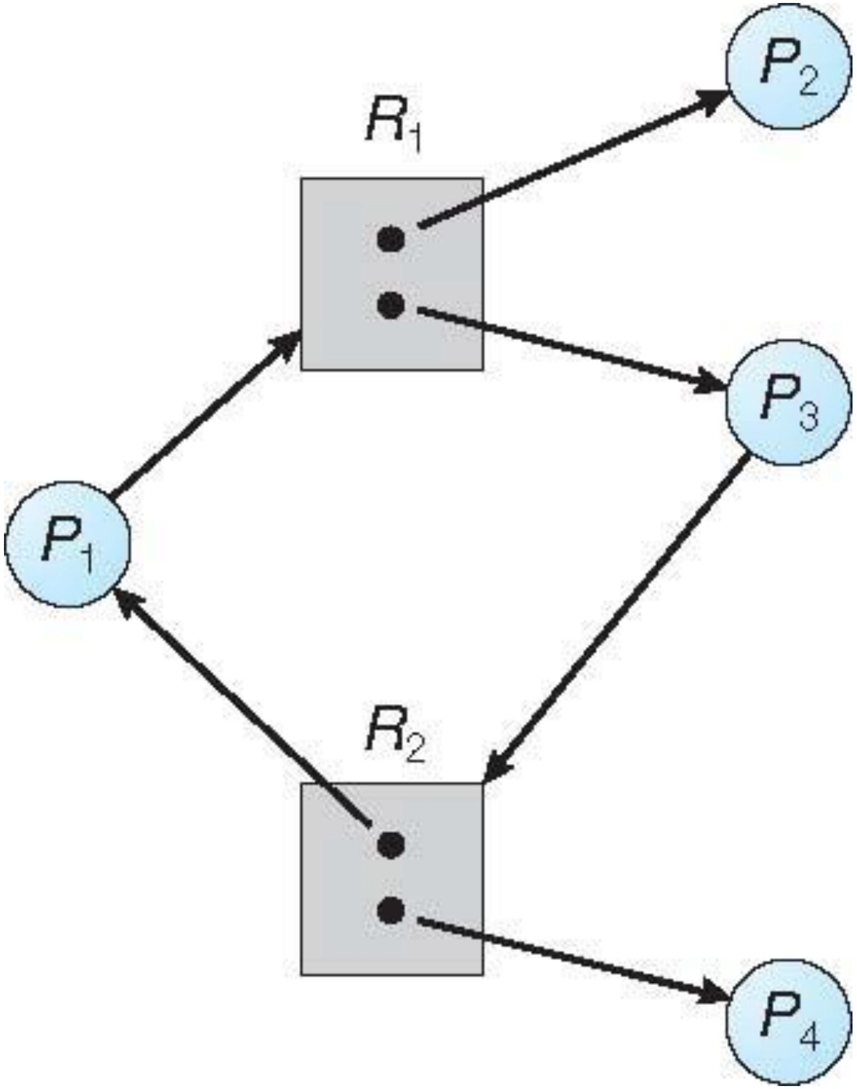
# Example of a Resource Allocation Graph



# Resource Allocation Graph With A Deadlock



# Graph With A Cycle But No Deadlock



# Exercise

Consider a system has one DVD drive, two printers, one CPU and three files (for storing information). Suppose there are three processes, i.e., $P_1$, $P_2$, and $P_3$. Process $P_1$ is holding an instance of resource type printer and is waiting for an instance of resource type DVD drive. Process $P_2$ is holding an instance of resource type DVD drive and an instance of printer and is waiting for an instance of resource type CPU. Process $P_3$ is holding an instance of resource type CPU and is waiting for an instance of resource type printer.

✓ **Show these resources in a resource-allocation graph.**

# 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

# Methods for Handling Deadlocks

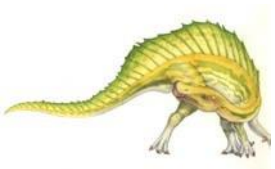
Methods for handling deadlock:

There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

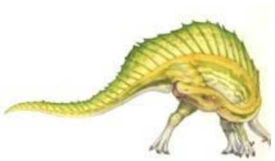


# Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible



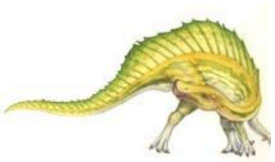
# Deadlock Prevention (Cont.)

- **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
  - 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
- **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

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

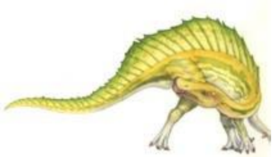


# Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

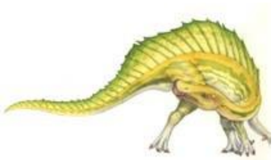
- System is in **safe state** if there exists a sequence <$P_1$, $P_2$, …, $P_n$> of ALL the processes in the systems such that for each $P_i$, the resources that $P_i$ can still request can be satisfied by currently available resources + resources held by all the $P_j$, with $j < I$

- 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

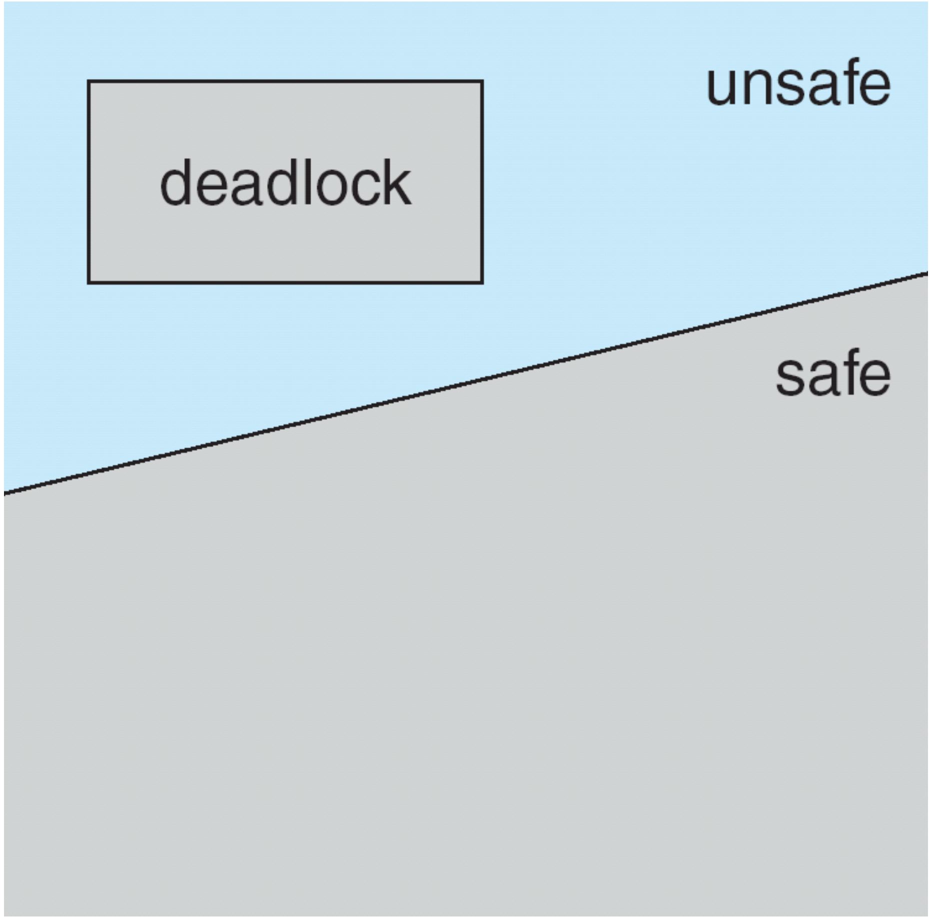


# Basic Facts

- 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

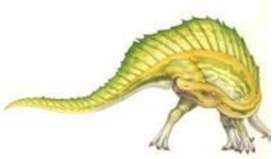
- **Claim edge** $P_i \rightarrow R_j$ indicated that process $P_j$ may request resource $R_j$; represented by a dashed line

- Claim edge converts to request edge when a process requests a resource

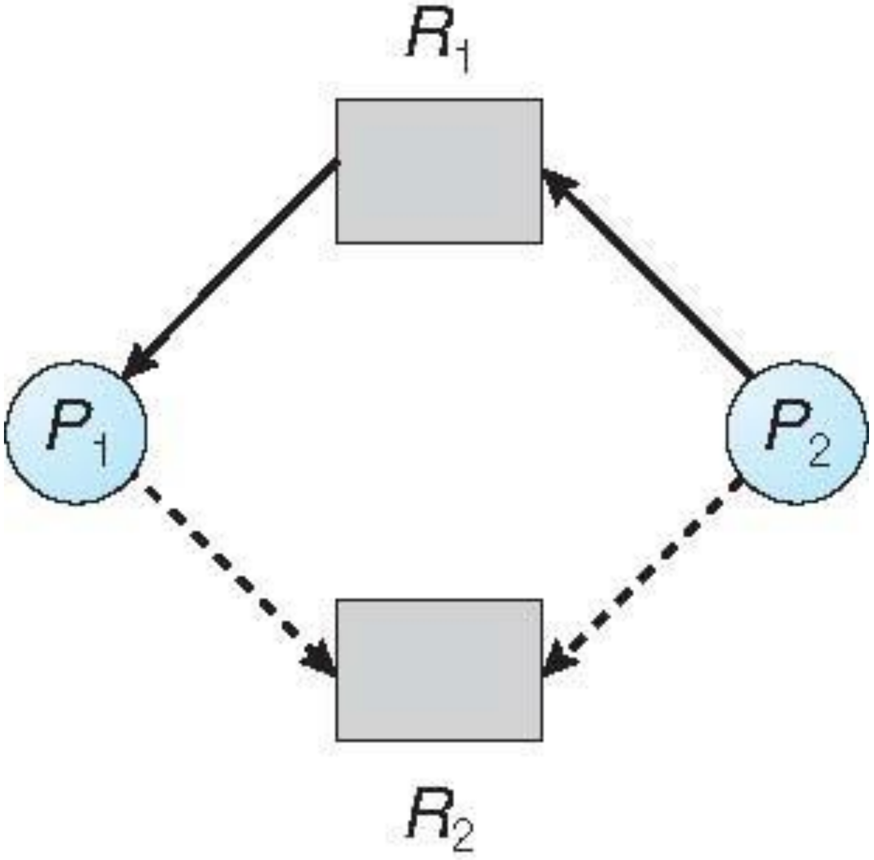
- Request edge converted to an assignment edge when the resource is allocated to the process

- When a resource is released by a process, assignment edge reconverts to a claim edge

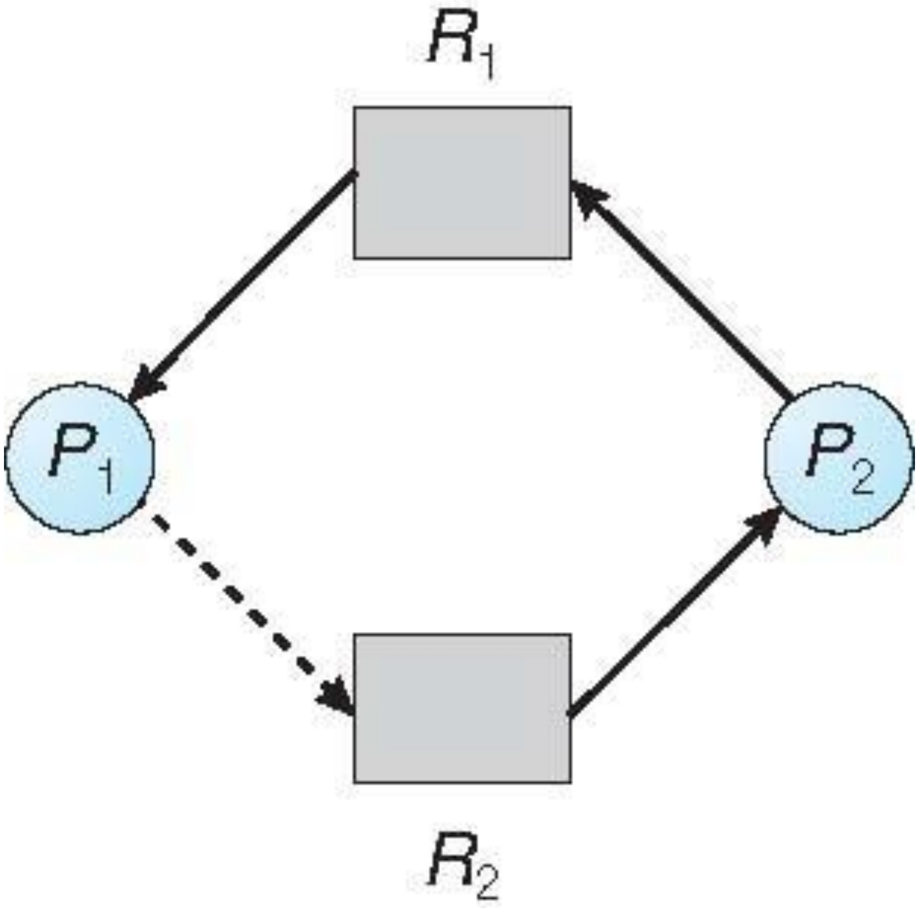
- Resources must be claimed *a priori* in the system



# Resource-Allocation Graph



# Unsafe State In Resource-Allocation Graph



# 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
- 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

$$Need\ [i,j] = Max[i,j] – Allocation\ [i,j]$$

# 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)* $Need_i \leq Work$

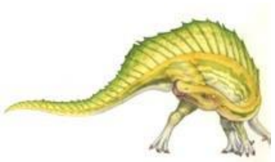
   If no such *i* exists, go to step 4

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* = request vector for process $P_i$. If $Request_i\,[\,\mathbf{j}\,] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$

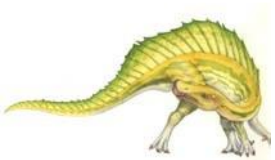
1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available
3. Pretend to allocate requested resources to $P_i$ by modifying the state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

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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



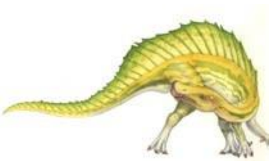
# Example of Banker's Algorithm

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

  *A* (10 instances), *B* (5 instances), 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.)

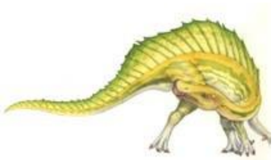
**Question 1. What will be the content of the Need matrix?**

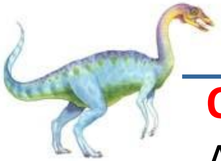
**Need= Max - Allocation**

Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

| 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 |

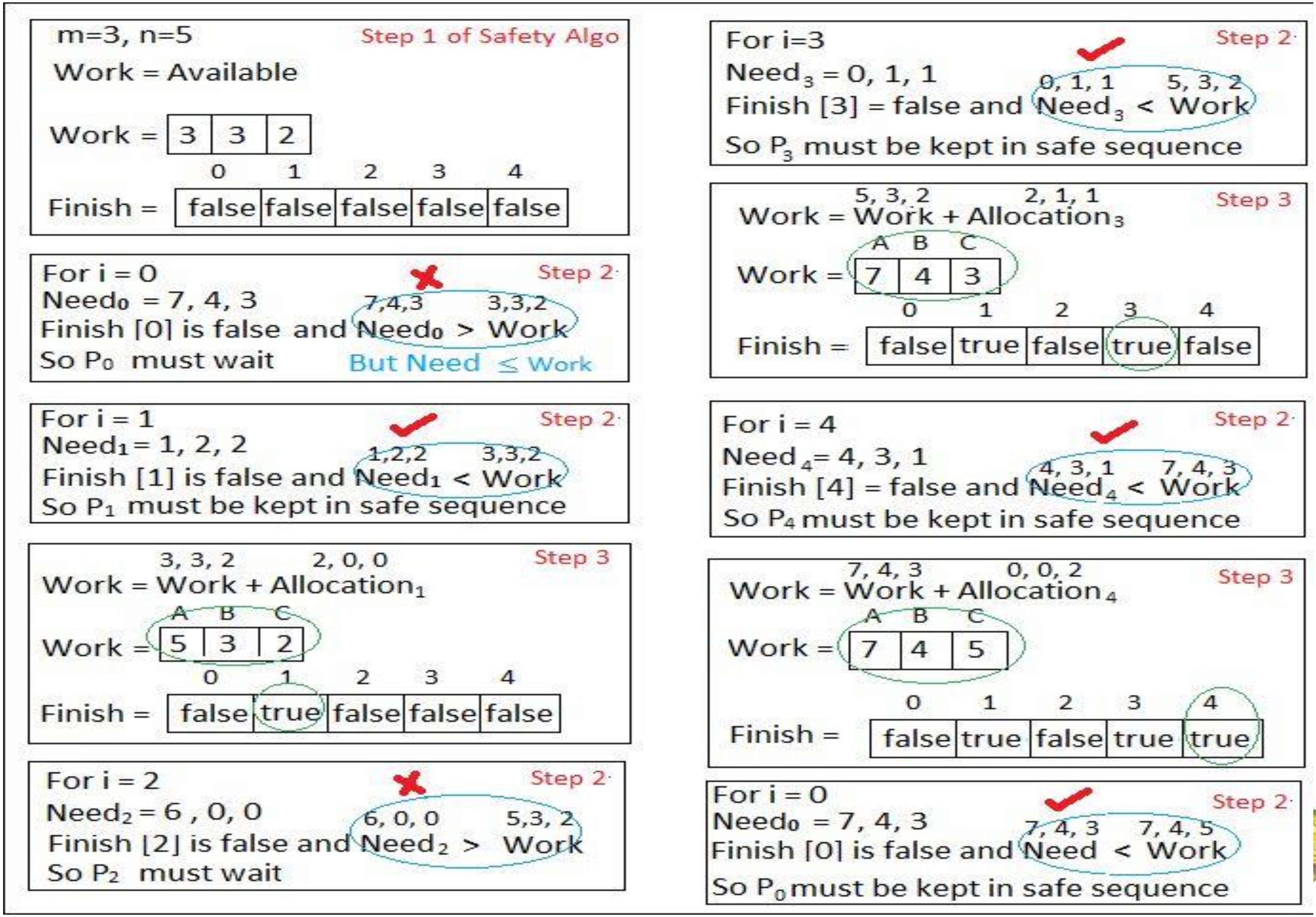




# Example (Cont.)

**Question 2.** **Is the system in safe state? If Yes, then what is the safe sequence?**

Applying the Safety algorithm on the given system,



m=3, n=5 Step 1 of Safety Algo

Work = Available

Work =

|   |   |   |
|---|---|---|
| 3 | 3 | 2 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | false | false | false | false |

For i = 0 Step 2

$Need_0 = 7, 4, 3$ ✗

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$

So $P_0$ must wait But Need ≤ Work

For i = 1 Step 2

$Need_1 = 1, 2, 2$ ✓

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$

So $P_1$ must be kept in safe sequence

Step 3

Work = $\overset{3,\,3,\,2}{Work} + \overset{2,\,0,\,0}{Allocation_1}$

Work =

| A | B | C |
|---|---|---|
| 5 | 3 | 2 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | true | false | false | false |

For i = 2 Step 2

$Need_2 = 6, 0, 0$ ✗

Finish [2] is false and $\overset{6,\,0,\,0}{Need_2} > \overset{5,3,\,2}{Work}$

So $P_2$ must wait

For i=3 Step 2

$Need_3 = 0, 1, 1$ ✓

Finish [3] = false and $\overset{0,\,1,\,1}{Need_3} < \overset{5,\,3,\,2}{Work}$

So $P_3$ must be kept in safe sequence

Step 3

Work = $\overset{5,\,3,\,2}{Work} + \overset{2,\,1,\,1}{Allocation_3}$

Work =

| A | B | C |
|---|---|---|
| 7 | 4 | 3 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | true | false | true | false |

For i = 4 Step 2

$Need_4 = 4, 3, 1$ ✓

Finish [4] = false and $\overset{4,\,3,\,1}{Need_4} < \overset{7,\,4,\,3}{Work}$

So $P_4$ must be kept in safe sequence

Step 3

Work = $\overset{7,\,4,\,3}{Work} + \overset{0,\,0,\,2}{Allocation_4}$

Work =

| A | B | C |
|---|---|---|
| 7 | 4 | 5 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | true | false | true | true |

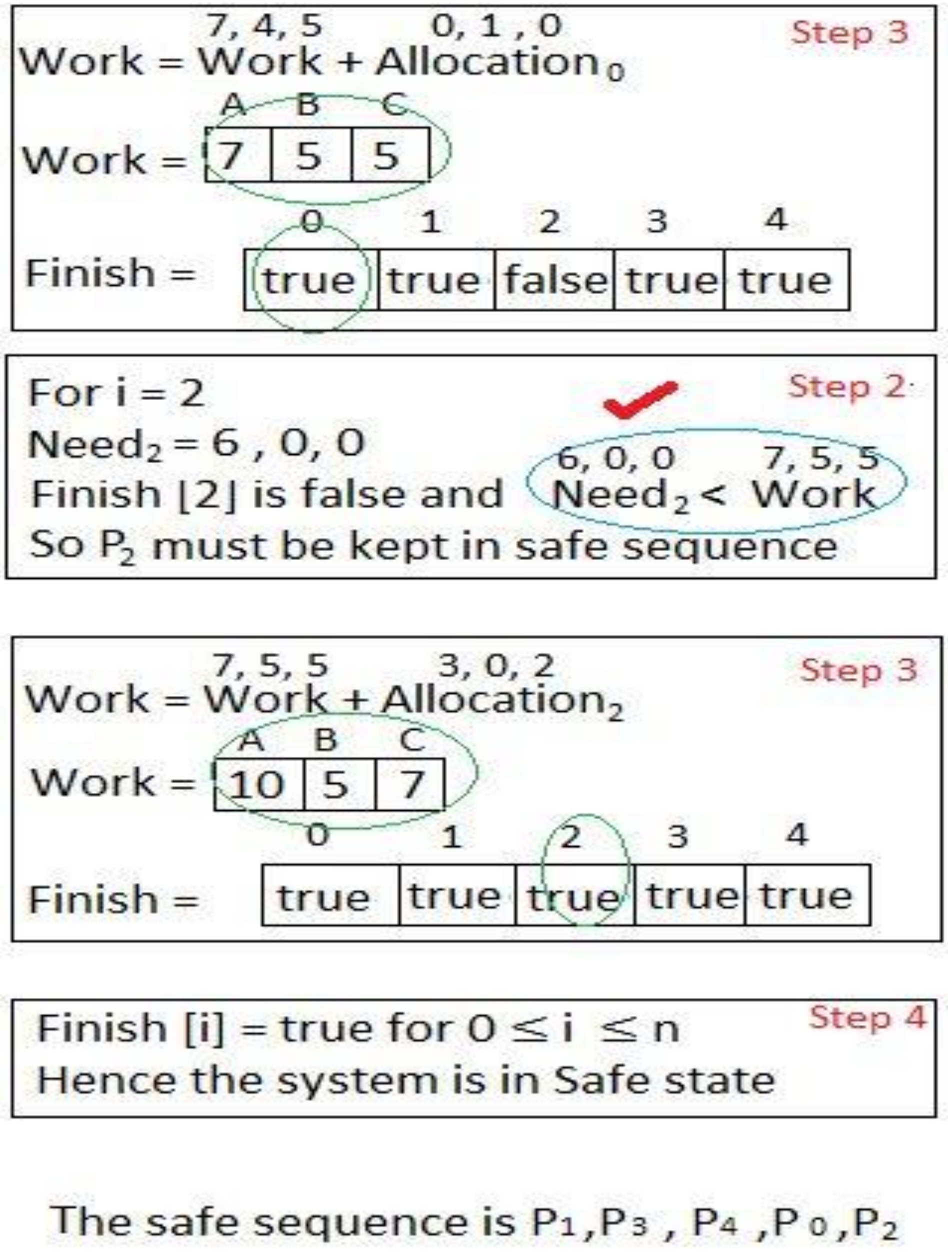
For i = 0 Step 2

$Need_0 = 7, 4, 3$ ✓

Finish [0] is false and $\overset{7,\,4,\,3}{Need} < \overset{7,\,4,\,5}{Work}$

So $P_0$ must be kept in safe sequence

# Example (Cont.)



Step 3

$\text{Work} = \overset{7,\,4,\,5}{\text{Work}} + \overset{0,\,1\,,\,0}{\text{Allocation}_0}$

Work =

| A | B | C |
|---|---|---|
| 7 | 5 | 5 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| true | true | false | true | true |

Step 2

For i = 2

$\text{Need}_2 = 6\,,\,0,\,0$

Finish [2] is false and $\overset{6,\,0,\,0}{\text{Need}_2} < \overset{7,\,5,\,5}{\text{Work}}$ ✓

So $P_2$ must be kept in safe sequence

Step 3

$\text{Work} = \overset{7,\,5,\,5}{\text{Work}} + \overset{3,\,0,\,2}{\text{Allocation}_2}$

Work =

| A | B | C |
|---|---|---|
| 10 | 5 | 7 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| true | true | true | true | true |

Step 4

Finish [i] = true for $0 \leq i \leq n$

Hence the system is in Safe state

The safe sequence is $P_1, P_3\,,\,P_4\,,P_0\,,P_2$

# Resource allocation graph

- Following is the Resource Allocation Graph:

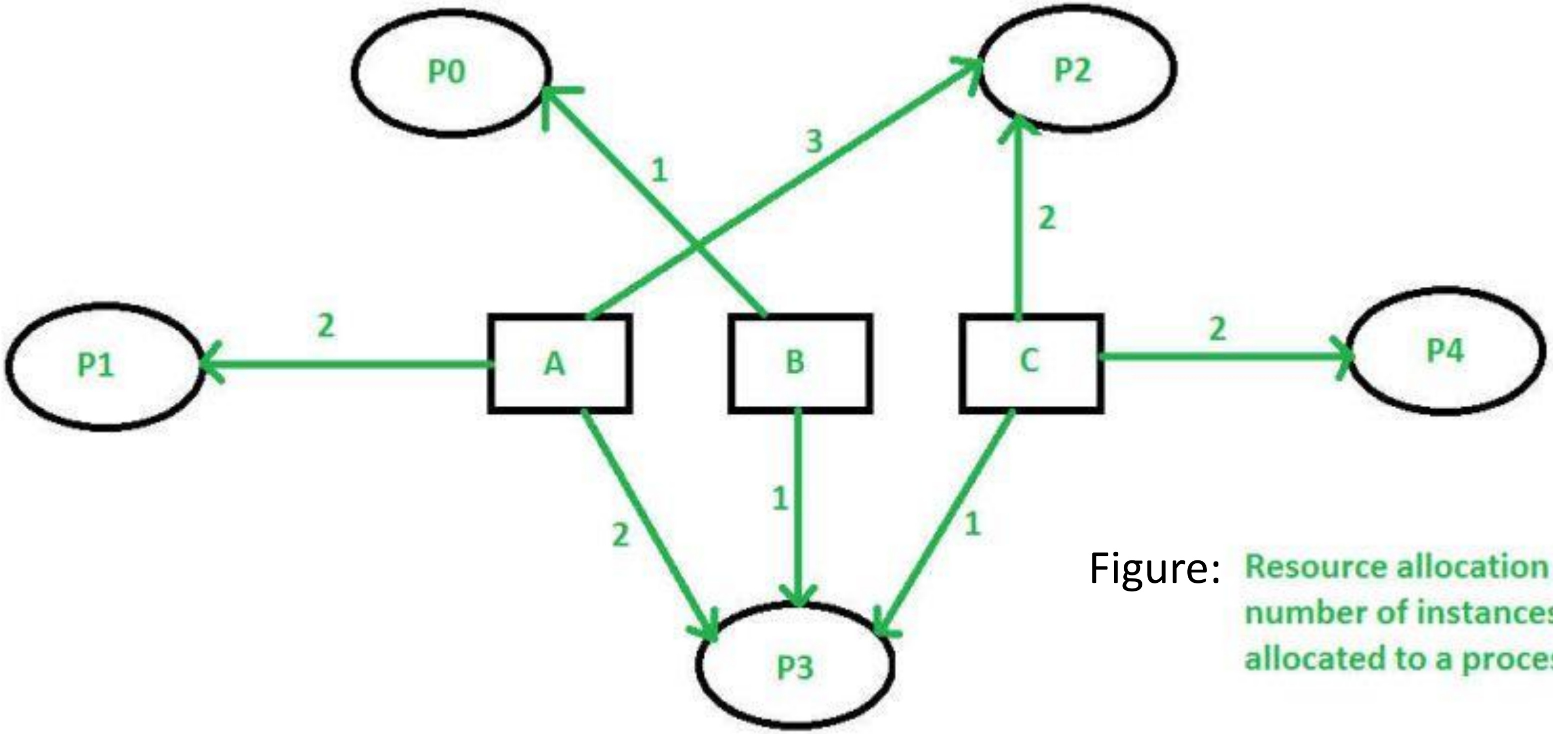
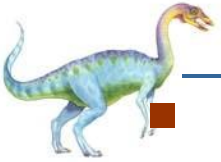


Figure: Resource allocation Graph showing number of instances of a resource allocated to a process.



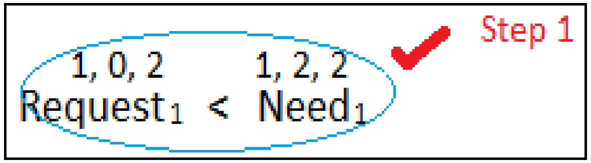
# Example: $P_1$ Request (1,0,2)

- **Question 3. What will happen if process $P_1$ requests one additional instance of resource type A and two instances of resource type C?**

A B C

$Request_1$ = 1, 0, 2

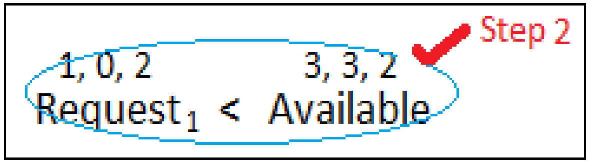
To decide whether the request is granted we use Resource-Request algorithm



Step 1 ✓

1, 0, 2 1, 2, 2

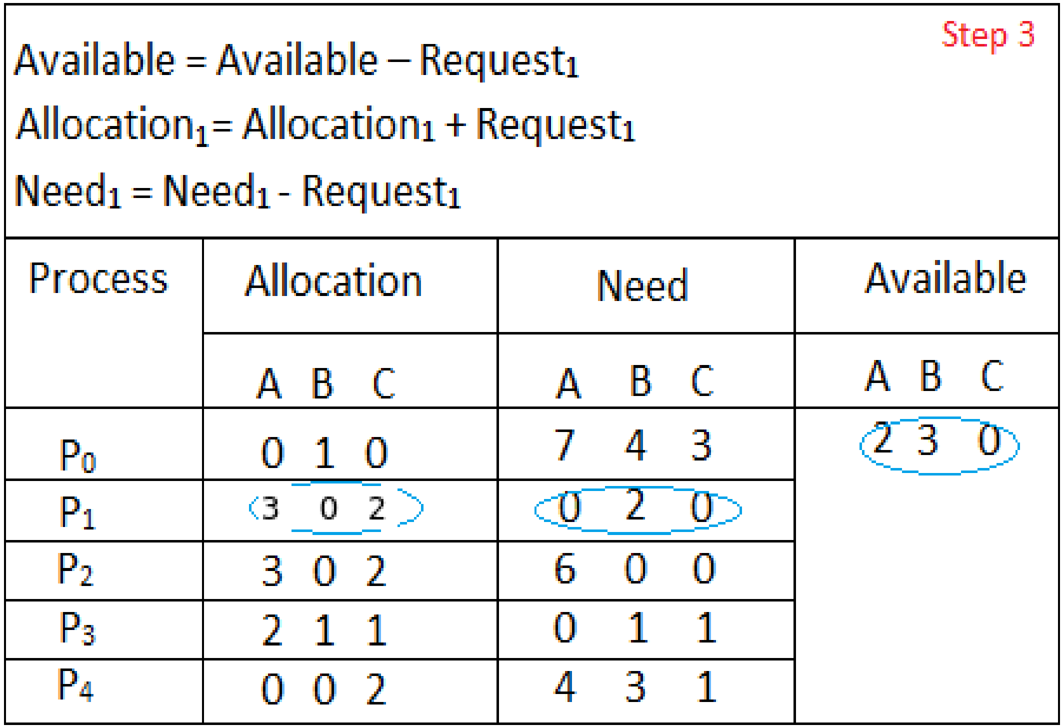
$Request_1 < Need_1$



Step 2 ✓

1, 0, 2 3, 3, 2

$Request_1 <$ Available



Step 3

$Available = Available - Request_1$

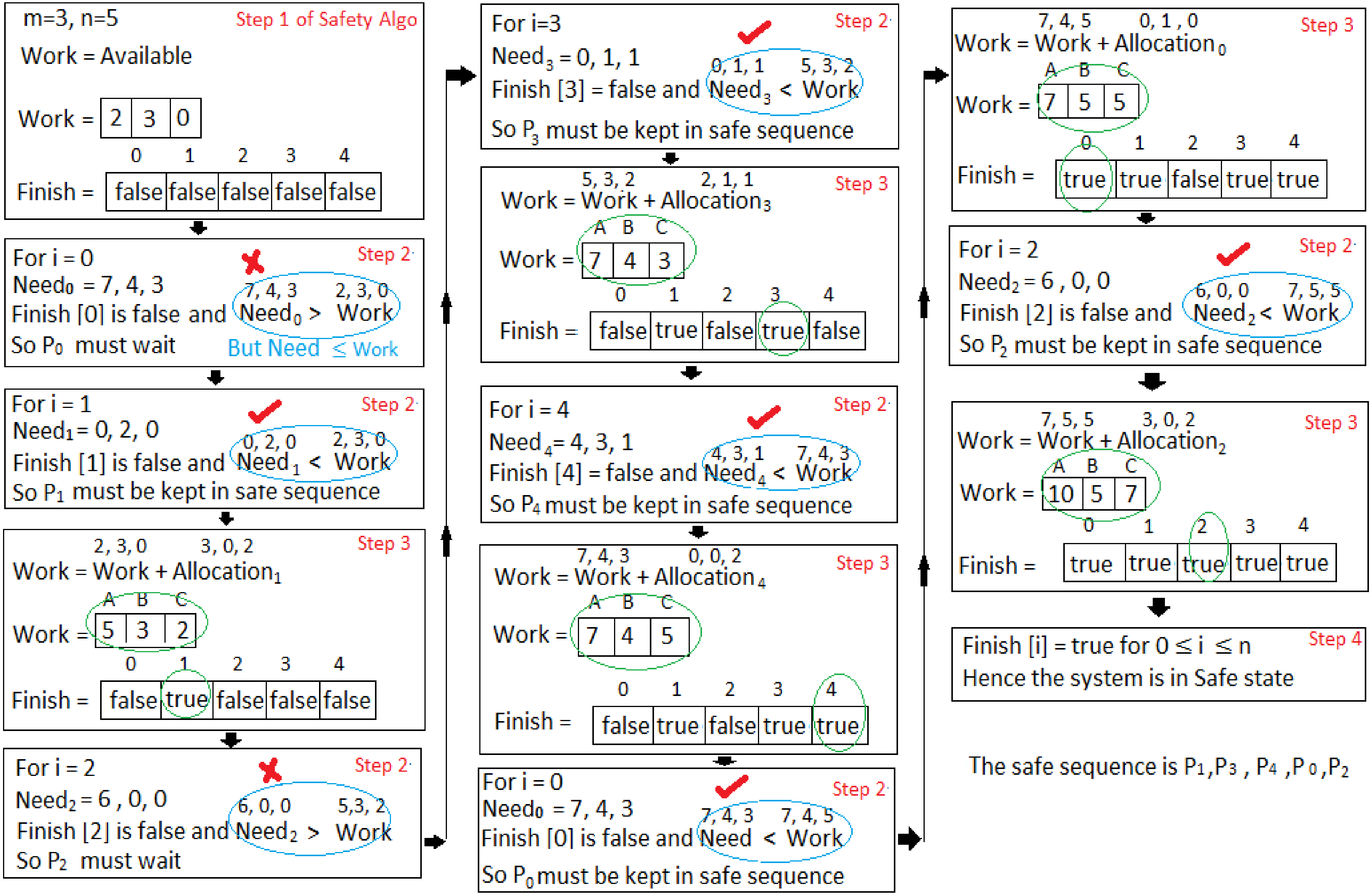
$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

| Process | 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 | |

# Example: $P_1$ Request (1,0,2)

- We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work =

| | | |
|---|---|---|
| 2 | 3 | 0 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | false | false | false | false |

For i = 0 — Step 2 ✗

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,\,4,\,3}{Need_0} > \overset{2,\,3,\,0}{Work}$

So $P_0$ must wait

But Need ≤ Work

For i = 1 — Step 2 ✓

$Need_1$ = 0, 2, 0

Finish [1] is false and $\overset{0,\,2,\,0}{Need_1} < \overset{2,\,3,\,0}{Work}$

So $P_1$ must be kept in safe sequence

Step 3

$\overset{2,\,3,\,0}{Work} = \overset{}{Work} + \overset{3,\,0,\,2}{Allocation_1}$

Work =

| A | B | C |
|---|---|---|
| 5 | 3 | 2 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | true | false | false | false |

For i = 2 — Step 2 ✗

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,\,0,\,0}{Need_2} > \overset{5,3,\,2}{Work}$

So $P_2$ must wait

For i=3 — Step 2 ✓

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,\,1,\,1}{Need_3} < \overset{5,\,3,\,2}{Work}$

So $P_3$ must be kept in safe sequence

Step 3

Work = $\overset{5,\,3,\,2}{Work}$ + $\overset{2,\,1,\,1}{Allocation_3}$

Work =

| A | B | C |
|---|---|---|
| 7 | 4 | 3 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | true | false | true | false |

For i = 4 — Step 2 ✓

$Need_4$= 4, 3, 1

Finish [4] = false and $\overset{4,\,3,\,1}{Need_4} < \overset{7,\,4,\,3}{Work}$

So $P_4$ must be kept in safe sequence

Step 3

Work = $\overset{7,\,4,\,3}{Work}$ + $\overset{0,\,0,\,2}{Allocation_4}$

Work =

| A | B | C |
|---|---|---|
| 7 | 4 | 5 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| false | true | false | true | true |

For i = 0 — Step 2 ✓

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,\,4,\,3}{Need} < \overset{7,\,4,\,5}{Work}$

So $P_0$ must be kept in safe sequence

Step 3

Work = $\overset{7,\,4,\,5}{Work}$ + $\overset{0,\,1\,,\,0}{Allocation_0}$

Work =

| A | B | C |
|---|---|---|
| 7 | 5 | 5 |

Finish =

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| true | true | false | true | true |

For i = 2 — Step 2 ✓

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,\,0,\,0}{Need_2} < \overset{7,\,5,\,5}{Work}$

So $P_2$ must be kept in safe sequence

Step 3

Work = $\overset{7,\,5,\,5}{Work}$ + $\overset{3,\,0,\,2}{Allocation_2}$

Work =

| A | B | C |
|---|---|---|
| 10 | 5 | 7 |

Finish =

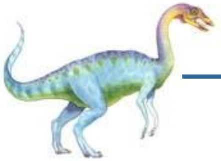
| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| true | true | true | true | true |

Step 4

Finish [i] = true for 0 ≤ i ≤ n

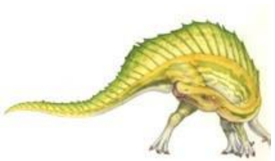
Hence the system is in Safe state

The safe sequence is $P_1$,$P_3$ , $P_4$ ,$P_0$ ,$P_2$



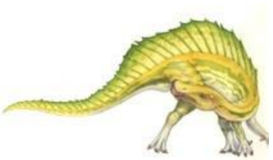
# Example: $P_1$ Request (1,0,2)

- Executing safety algorithm shows that sequence < $P_1$, $P_3$, $P_4$, $P_0$, $P_2$> satisfies safety requirement.
- Hence the new system state is safe, so we can immediately grant the request for process **$P_1$** .
- Can request for (3,3,0) by $P_4$ be granted?
- Can request for (0,2,0) by $P_0$ be granted?



# Deadlock Detection

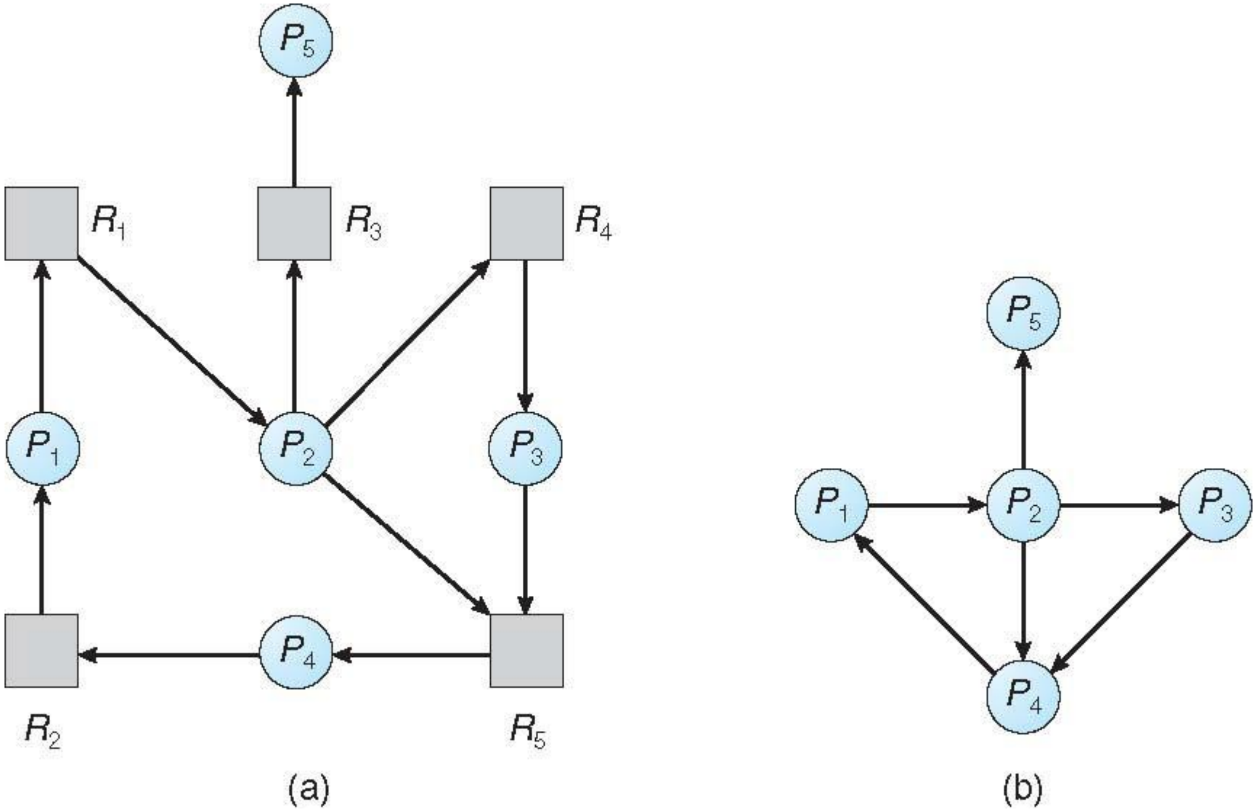
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



# 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, 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 requestof 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) $Request_i \leq Work$

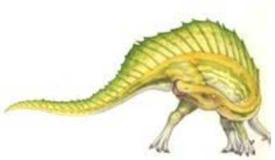
   If no such *i* exists, go to step 4

# 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

**Algorithm requires an order of O($m$ x $n^{2)}$ operations to detect whether the system is in deadlocked state**



# 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

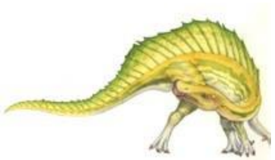
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

# 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, include number of rollback in cost factor



# End of Chapter 7