Deadlocks (cont..)

Contents of Lecture:

- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

References for This Lecture:

- *Abraham Silberschatz, Peter Bear Galvin and Greg Gagne, Operating System Concepts, 9th Edition,* ***Chapter 7***

Deadlock Avoidance

- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
  - For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.
  - With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

- The various algorithms that use this approach differ in the amount and type of information required.
  - The simplest and most useful model requires that each process declare the **maximum number of resources** of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
  - A deadlock-avoidance algorithm dynamically examines the **resource-allocation state** to ensure that a **circular-wait condition** can never exist.
  - The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

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

Resource-allocation graph algorithm

- If there is a resource-allocation system with only one instance of each resource type, can use a variant of the resource-allocation graph for deadlock avoidance.

- In addition to the **request** and **assignment edges** already described, introduce a new type of edge, called a **claim edge**.
  - ✓ A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future.
  - ✓ This edge resembles a request edge in direction but is represented in the graph by a **dashed line**.

- **The algorithm:**
  1. Claim edge converts to request edge when a process requests a resource
  2. Request edge converted to an assignment edge when the resource is allocated to the process
  3. When a resource is released by a process, assignment edge reconverts to a claim edge
  4. Resources must be claimed a priori in the system

- To illustrate this algorithm, consider the following resource-allocation graph of Figure 7.7.

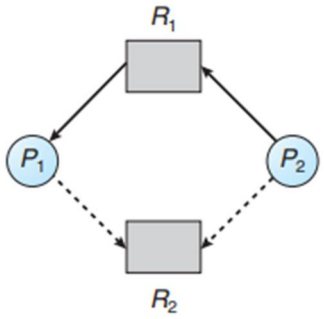


Figure 7.7 Resource-allocation graph for deadlock avoidance.

- Suppose that P2 requests R2. Although R2 is currently free, cannot allocate it to P2, since this action will create a cycle in the graph next figure (Figure 7.8). A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

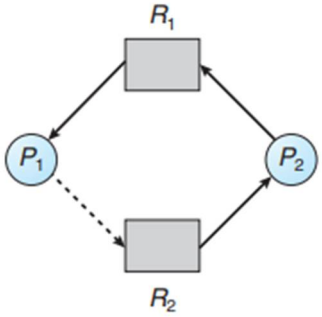


Figure 7.8 An unsafe state in a resource-allocation graph.

Banker's algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
  - ✓ If it will, the resources are allocated;
  - ✓ Otherwise, the process must wait until some other process releases enough resources.

- Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. need the following data structures, where ***n*** is the **number of processes** in the system and ***m*** is the **number of resource types**:

  - ✓ **Available:** A vector of length ***m*** indicates the number of available resources of each type. If Available ***[j] equals k***, then ***k*** instances of resource type ***Rj*** are available.

  - ✓ **Max:** An ***n*** × ***m*** matrix defines the maximum demand of each process. If ***Max[i][j] equals k***, then process ***Pi*** may request at most ***k*** instances of resource type ***Rj*** .

  - ✓ **Allocation:** An **n** × **m** matrix defines the number of resources of each type currently allocated to each process. If ***Allocation[i][j] equals k***, then process ***Pi*** is currently allocated ***k*** instances of resource type ***Rj*** .

  - ✓ **Need:** An **n** × **m** matrix indicates the remaining resource need of each process. If ***Need[i][j] equals k***, then process ***Pi*** may need k more instances of resource type ***Rj*** to complete its task.

    Note that Need[i][j] equals 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) Needi <= Work
   If no such i exists, go to step 4

3. **Work = Work + Allocationi**
   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

- ***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*$_i$ ≤ *Need*$_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
  2. If ***Request*$_i$ ≤ *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 with a Banker

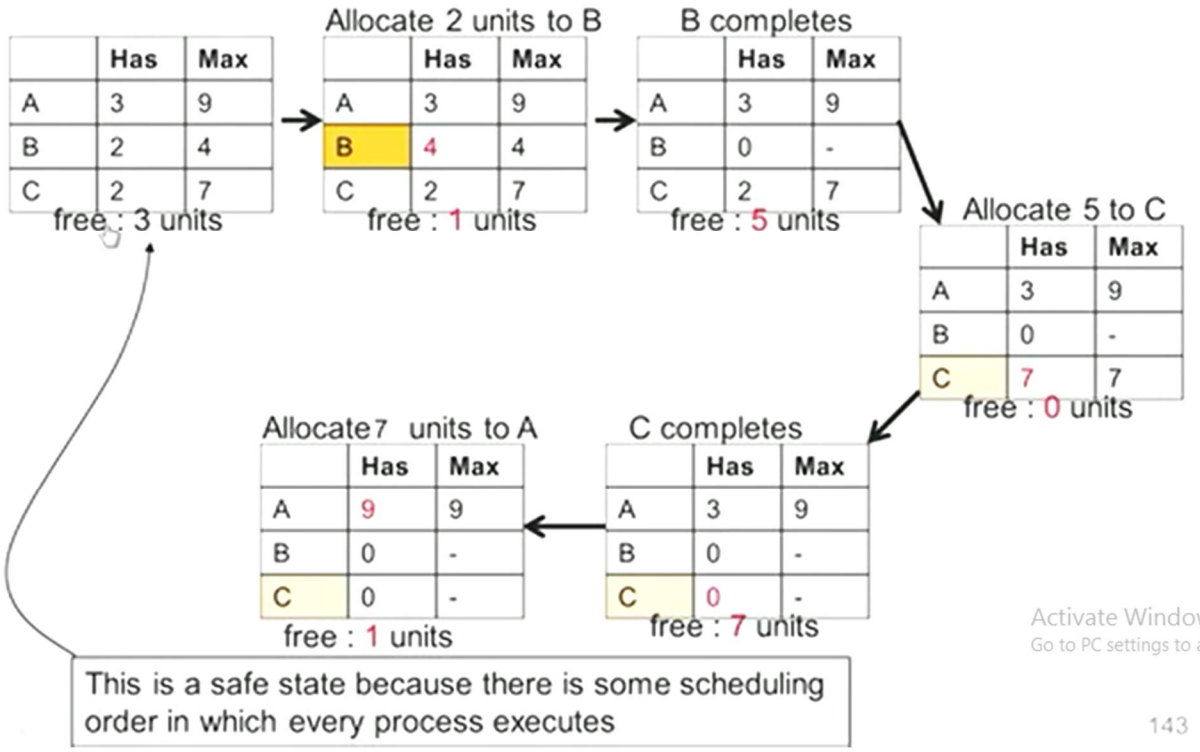
- Consider a banker with 3 clients (A, B, C)
- Each client has certain limits (totaling 20 units)
- The banker know that max credits will not be used at once, so keeps only 10 units

	Has	Max
A	3	9
B	2	4
C	2	7

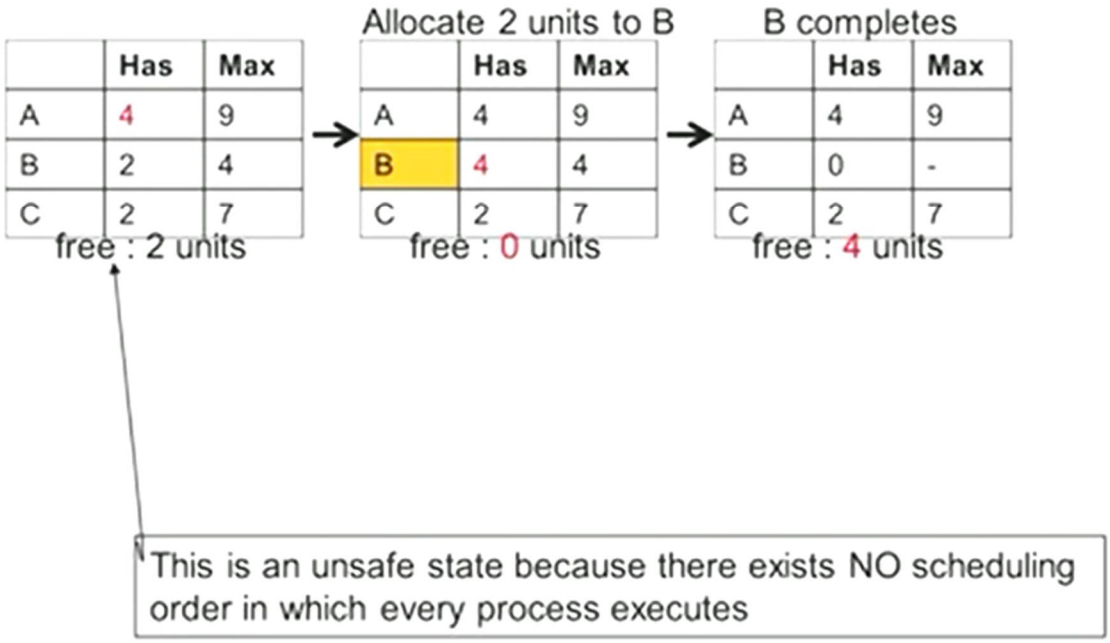
Total: 10 units free: 3 units

- Client declare maximum credits in advance. The banker can allocate credits provided no unsafe state is reached.

Safe state



Unsafe state



Example

- To illustrate the use of the banker's algorithm, consider the following:
  - ✓ 5 processes P0 through P4;
  - ✓ 3 resource types:
    - A (10 instances), B (5instances), and C (7 instances)
- Snapshot at time T0:

	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	

- The content of the matrix ***Need*** is defined to be ***Max − Allocation*** and is as:

	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 < **P1, P3, P4, P2, P0**> satisfies safety criteria.
- Suppose now that process **P1** requests one additional instance of resource type A and two instances of resource type C, so **Request1 = (1,0,2)**.

- To decide whether this request can be immediately granted, first check that **Request1 ≤ Available** that is, that (1,0,2) ≤ (3,3,2), which is true. then pretend that this request has been fulfilled, and we arrive at the following new state:

	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 < **P1, P3, P4, P0, P2**> satisfies safety requirement

Exercise:

- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detection

- If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
  - ✓ An algorithm that examines the state of the system to determine whether a deadlock has occurred (detection algorithm).
  - ✓ An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- If all resources have only a single instance, then can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for graph**.
  - ✓ Obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

- An edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
  - ✓ An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resourceallocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq .
  - ✓ Next figure (Figure 7.9), present a resource-allocation graph and the corresponding wait-for graph.

- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.

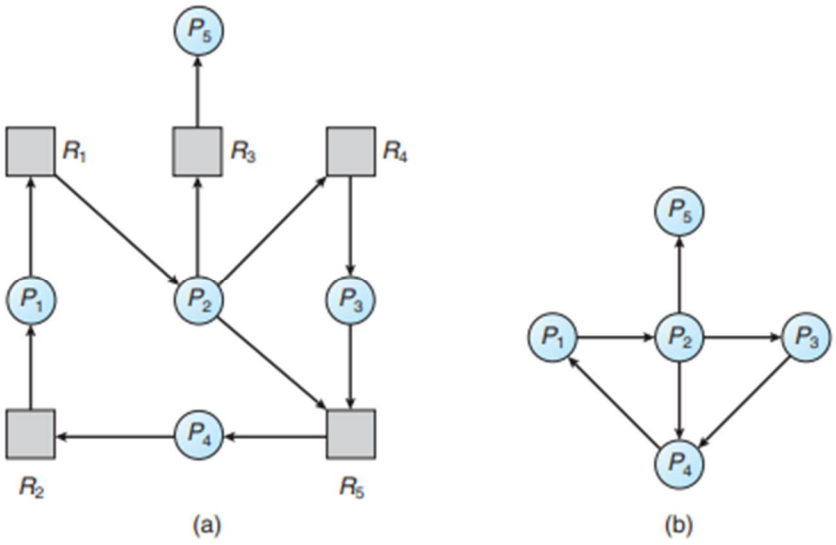


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Several Instances of a Resource Type

- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:
  - ✓ **Available:** A vector of length **m** indicates the number of available resources of each type.
  - ✓ **Allocation:** An **n × m** matrix defines the number of resources of each type currently allocated to each process.
  - ✓ **Request:** An **n × m** matrix indicates the current request of each process. If **Request[i][j] equals k**, then process **Pi** is requesting **k** more instances of resource type **Rj**

Detection algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively. Initialize:
   Work = **Available**.
   For i = 0, 1, ..., n–1,
   if **Allocationi ≠ 0**,
   then **Finish[i] =false**.
   Otherwise, **Finish[i] = true**.

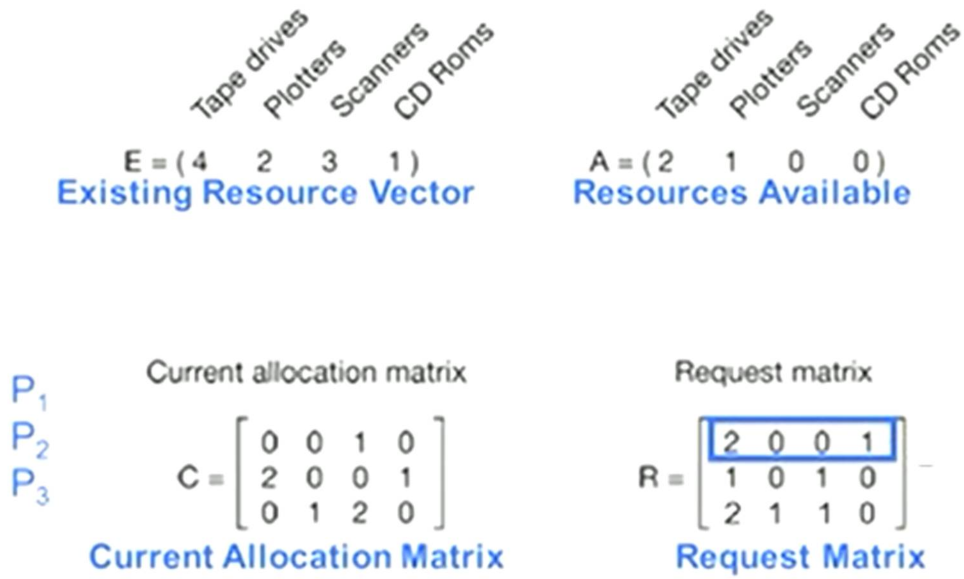
2. Find an index i such that both
   a. **Finish[i]** == **false**
   b. **Requesti** ≤ **Work**
   If no such i exists, go to step 4.

3. **Work** = **Work** + **Allocationi**
   Finish[i] = **true**
   Go to step 2.

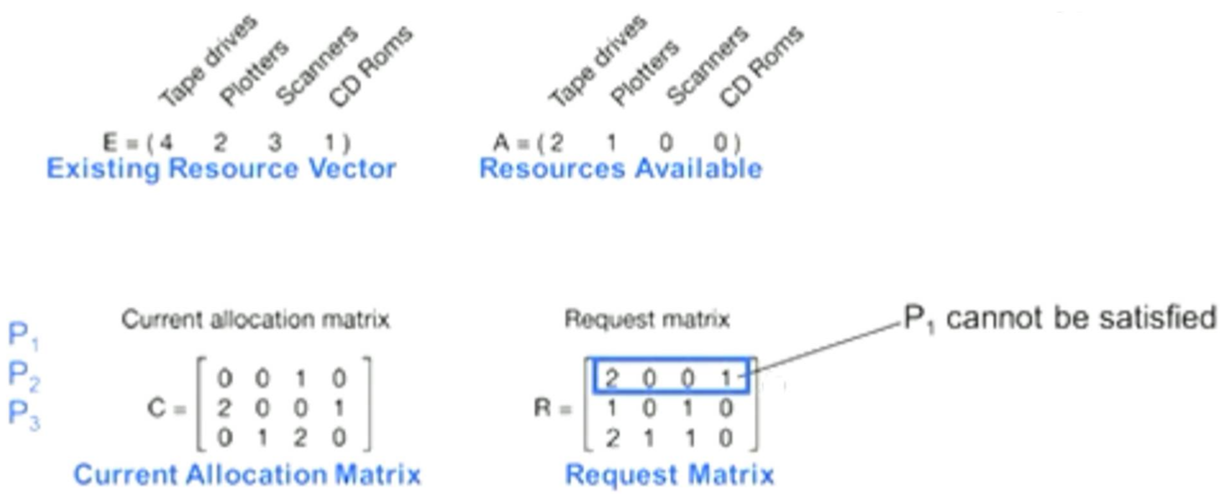
4. If **Finish[i]** == **false** for some **i,** $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if **Finish[i]** == **false**, then process **Pi** is deadlocked.

- Algorithm requires an order of **O(m x n2)** operations to detect whether the system is in deadlocked state

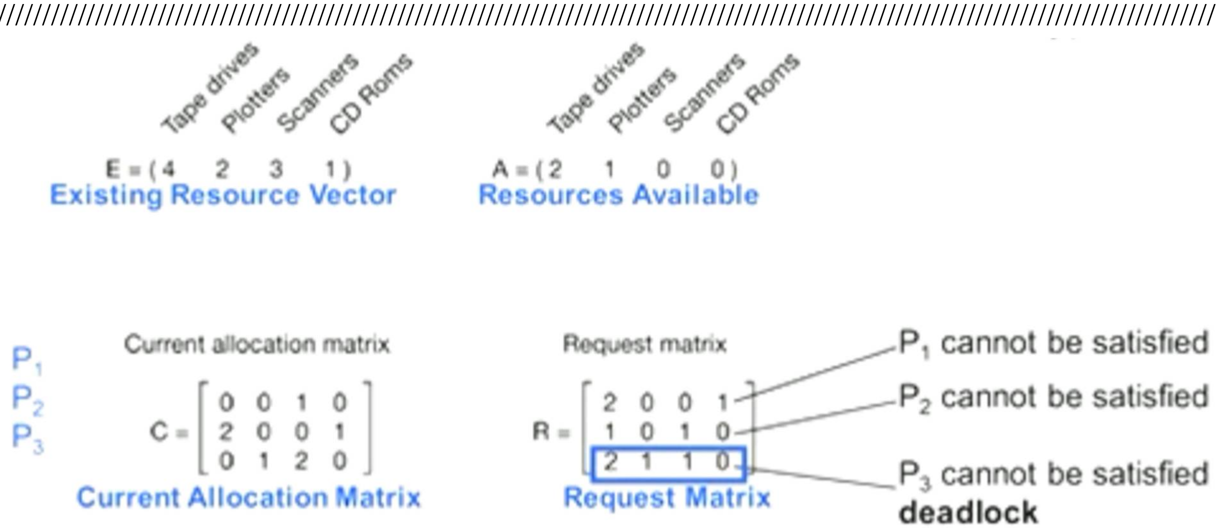
Example



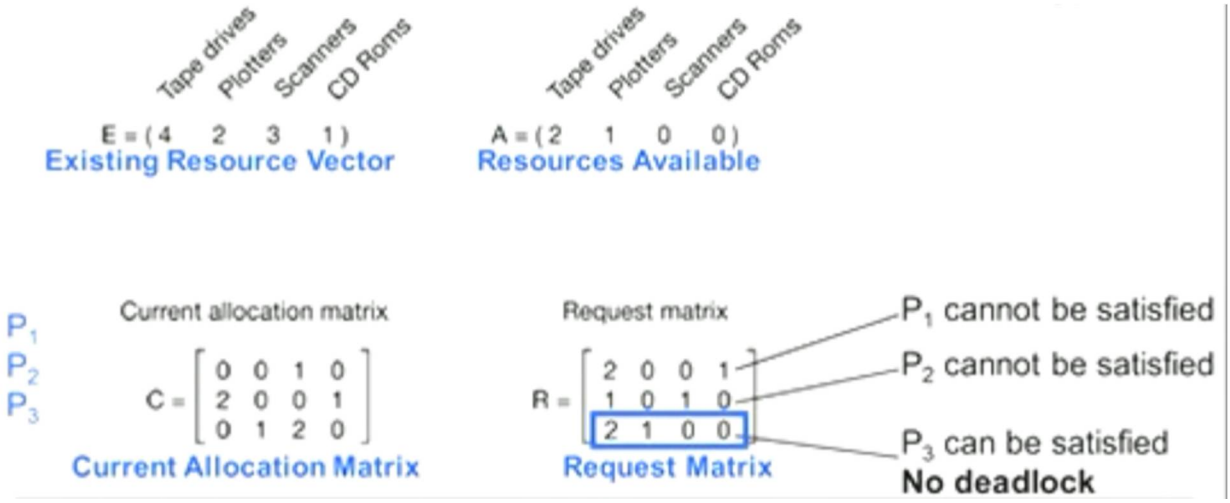
- ❖ Process Pi holds Ci resources and requests Ri resources, where i = 1 to 3
- ❖ Check if there is any sequence of allocation by which all current request can be met. If so, there is no deadlock.



//



- If **P3** change the request to (2 1 0 0) as following. P3 can be satisfied, so, no deadlock.



Example

- Consider a system with five processes P0 through P4 and three resource types A, B, and C.
- Resource type A has seven instances, resource type B has two instances, and resource type C has six instances.
- Suppose that, at time T0, we have the following resource-allocation state:

	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	

- Claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, will find that the sequence <P0, P2, P3, P1, P4> results in **Finish[i] == true** for all i.

- Suppose now that process P2 makes one additional request for an instance of type C. The **Request** matrix is modified as follows:

	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

- A deadlock exists, consisting of processes P0, P1, P2, P3, and P4.

Recovery from Deadlock

- There are two options for breaking a deadlock.
  - ✓ One is simply to **abort** one or more processes to break the circular wait.
  - ✓ The other is to **preempt** some resources from one or more of the deadlocked processes.

Process Termination

- To eliminate deadlocks by aborting a process, one of two methods used:
  - ✓ Abort all deadlocked processes
  - ✓ Abort one process at a time until the deadlock cycle is eliminated

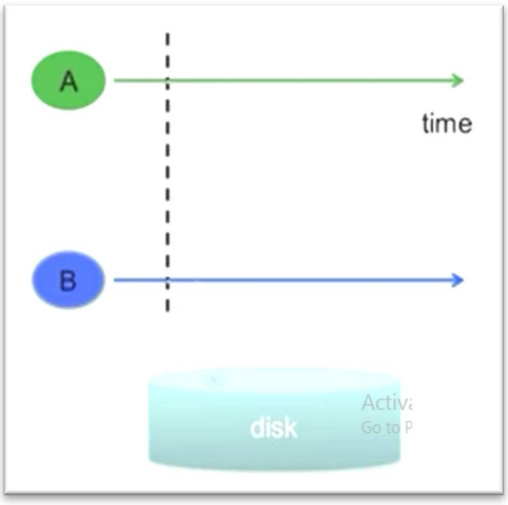
- Many factors may affect which process is chosen, including:
  1. What the priority of the process is
  2. How long the process has computed and how much longer the process will compute before completing its designated task
  3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
  4. How many more resources the process needs in order to complete
  5. How many processes will need to be terminated
  6. Whether the process is interactive or batch

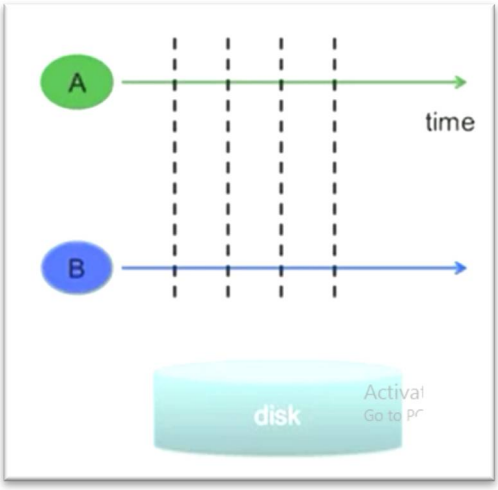
Resource Preemption

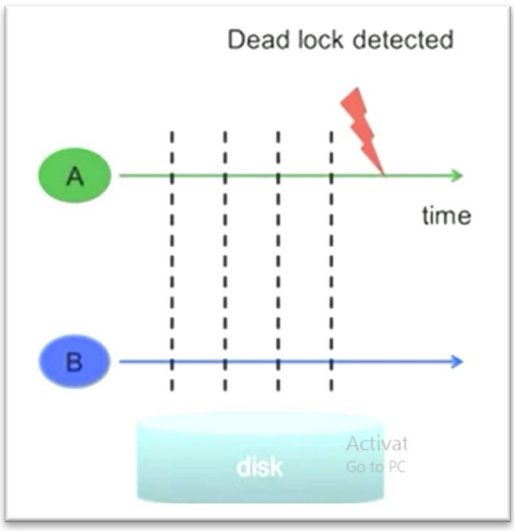
- To eliminate deadlocks using resource preemption, successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

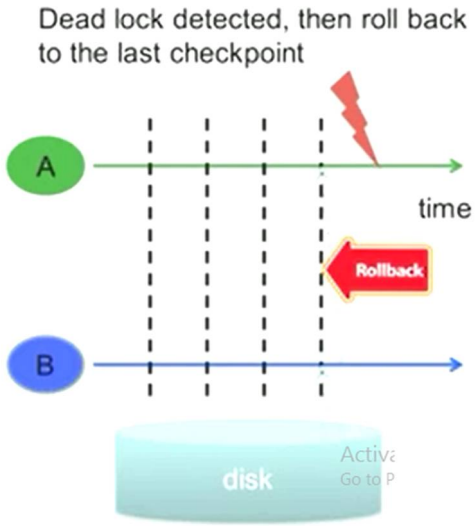
- If preemption is required to deal with deadlocks, then three issues need to be addressed:
  1. **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, must determine the order of preemption to minimize cost.

  2. **Rollback:** Checkpoint states and then rollback.









3. **Starvation:** same process may always be picked as victim, include number of rollback in cost factor