PART 6: Deadlock

6.1 Introduction

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a ***deadlock***.

6.1.1 System Model

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:
1. *Request*. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. *Use*. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. *Release*. The process releases the resource

If the system has m of resource R, with n processes competing for them. Each process may need m/n of resource R.

6.1.2 Deadlock Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set { P0, P1, ..., Pn } of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

6.3 Resource-Allocation Graph

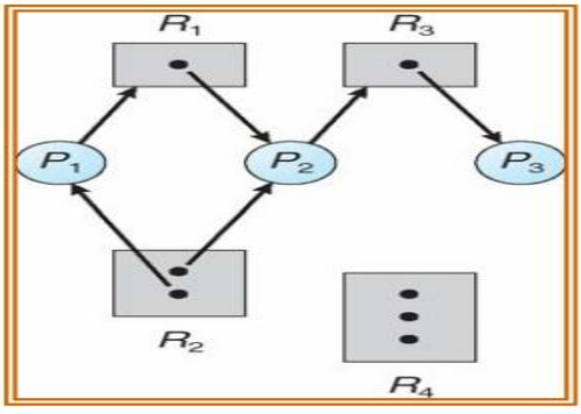
Deadlocks can be described more precisely in terms of a directed graph called a system *resource-allocation graph*. This graph consists of a set of vertices V and a set of edges E .This graph used to describe deadlock more precisely. It consist of:

1. set of processes P={P1, P2, ..., Pn}
2. set of resources R={ R1, R2, ..., Rm}
3. set of edges E

- Pi ¯ Rj : denote that process Pi request Rj.

- Rj ¯ Pi : denote that resource Rj allocates to process Pi.

Ex: Let we have the following Resource Allocation Graph:



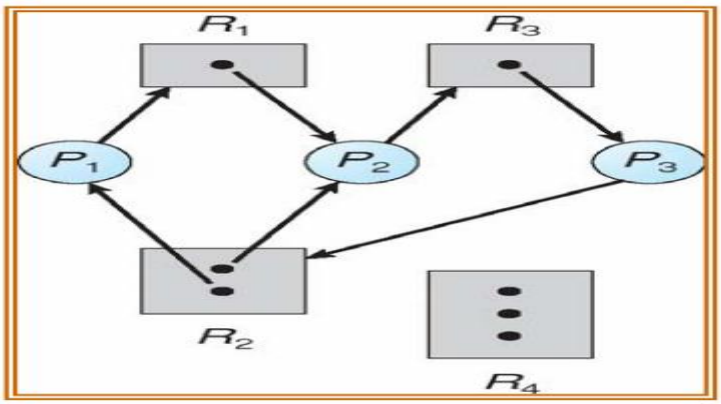
- The sets of P, R, and E are:
  - $P=\{P_1, P_2, P_3\}$
  - $R=\{R_1, R_2, R_3, R_4\}$
  - $E=\{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - One instance of R_1.
  - Two instances of R_2.
  - One instance of R_3.
  - Three instances of R_4.

• Process States
- Process P1 is holding an instance of resource R2, and is waiting for instance of R1.
- Process P2 is holding an instance of R1 and R2, and is waiting for instance of R3.
- Process P3 is holding an instance of R3.

Notes:

1. If the graph contains no cycle, then no process in the system is deadlocked.
2. If the graph contains a cycle, then:
   - if each resource type has exactly one instance, then deadlock occur.
   - If each resource has several instances, then deadlock may occur.

- Now consider that P_3 request an instance of R_2 ($P_3 \rightarrow R_2$)



At this point, two cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

So, processes P_1, P_2, and P_3 are deadlocked. P_2 is waiting for R_3, which is held by P_3. P_3 is waiting for either P_1 or P_2 to release R_2. P_1 is waiting P_2 to release R_1

6.4 Deadlock Handling

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

6.4.1 Deadlock Prevention

As we noted, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

-Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several processes.

-Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system,

a. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.
b. An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

- *No preemption*

To ensure that this condition does not hold, we can use the following protocol:

> If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
> Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

- *circular-wait*

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of

all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let R = {R1, R2, ..., Rm } be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R → N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F (tape drive) = 1

F (disk drive) = 5

F (printer) = 12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type —say, Ri. After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj). Note also that if several instances of the same resource type are needed, a single request for all of them must be issued.

6.4.2 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. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

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. In the following sections, we explore two deadlock-avoidance algorithms.

Safe State **:**

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a *safe sequence*. A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj , with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all P j have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, a so on. If no such sequence exists, then the system state is said to be *unsafe.The unsaved may be not deadlock*

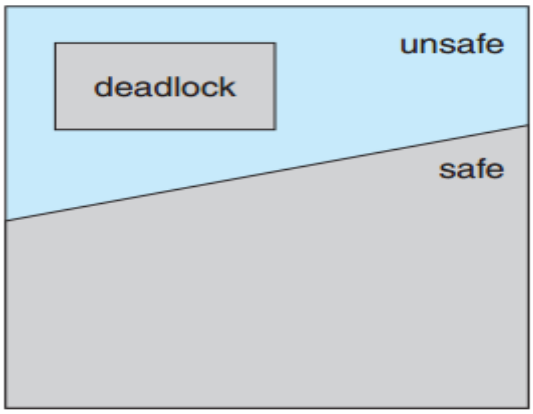


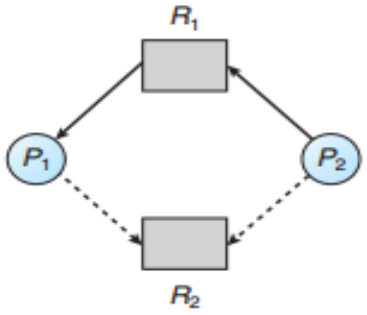
Figure 5.1 Safe, unsafe, and deadlocked state spaces.

Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only *one instance* of each resource type, we can use a variant of the resource-allocation graph for *deadlock avoidance*. In addition to the request and assignment edges already described, we 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. When process Pi requests resource Rj , the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a *claim edge* Pi → Rj.

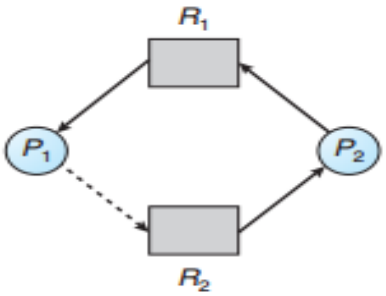
Before process Pi starts executing, all its *claim edges* must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge Pi → Rj to be added to the graph only if all the edges associated with process Pi are claim edges.

To illustrate this algorithm, we consider the resource-allocation graph of bellow Figure :



Resource-allocation graph for deadlock avoidance.

Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph (Figure Bellow) :



An unsafe state in a resource-allocation graph.

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

Banker's Algorithm

This deadlock avoidance algorithm . This algorithm is commonly known as the b
Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

• Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available.

• Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
• Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
• Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task : Need [i,j] = Max[i,j] – Allocation [i,j].

a.Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

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 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.

EXAMPL: Lets have 5 processes (P0- P4); 3 resource types: A (10 instances), B (5instances), and C (7 instances).

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

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

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Safety algorithm:

If Need<=Available then
Process is executed
New available =available+ allocation
Else
Do not execute and go forward

P0	Need≥ available 7 3 4≥3 3 2	P0 is NOT execute
P1	Need≤ available 1 2 2≤ 3 3 2 New available=3 3 2+2 0 0=5 3 2	P1 is execute
P2	Need≥ available 6 0 0 ≥5 3 2	P2 is NOT execute
P3	Need≤ available 0 1 1 ≤5 3 2 New available=5 3 2+2 1 1=7 4 3	P3 is execute
P4	Need≤ available 4 3 1≤ 7 4 3 New available=7 4 3+0 0 2=7 4 5	P4 is execute
P0	Need≤ available 7 3 4≤ 7 4 5 New available= 7 4 5+0 1 0 =7 5 5	P4 is execute
P2	Need≤ available 6 0 0 ≤7 5 5 New available=7 5 5+3 0 2=10 5 7	P2 is execute

- The system is in a safe state since the sequence < *P*1, *P*3, *P*4, *P*2, *P*0> satisfies safety criteria.

b.Resource-Request Algorithm for Process Pi

we describe the algorithm for determining whether requests can be safely granted.Let **Request**i be the request vector for process Pi . If **Request**i [j] == k, then process Pi wants k instances of resource type R j. When a request for resources is made by process Pi, the following actions are taken:

1. If **Request**i ≤ **Need**i, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If **Request**i ≤ **Available,** go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = **Available**–**Request**i ;
Allocationi = **Allocation**i + **Request**i ;
Needi = **Need**i –**Request**i ;

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for **Request**i **,** and the old resource-allocation state is restored.

Ex: *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 1	6 0 0	
*P*3	2 1 1	0 1 1	
*P*4	0 0 2	4 3 1	

1. Executing safety algorithm shows that sequence < *P*1, *P*3, *P*4, *P*0, *P*2> satisfies safety requirement.
2. Can request for (3,3,0) by *P*4 be granted?
3. Can request for (0,2,0) by *P*0 be granted?

ANSWER:

2. CAN REQUEST FOR (3 ,3,0)FOR P4 BE GRANTD?
   (3,3,0)>=(4 3 1) THEN P4 NO GRANTED
3. CAN REQUEST FOR (0 ,2,0) FOR P0 BE GRANTD?
   (0 ,2,0) <=(4 3 1) AND (0 ,2,0) <=(2 , 3 ,0) THEN P0 CAN GRANTED AND:

   Then

Available = (2 , 3 ,0) –(0 ,2,0) = (2 ,1 ,0) ;
Allocation of P0 = (0 ,1,0)+ (0 ,2,0) = (2 ,3 ,0) ; ;
Need of P0 = (7 , 4, 3) –(0 ,2,0) = (7 ,2,3);

6.4.3 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:

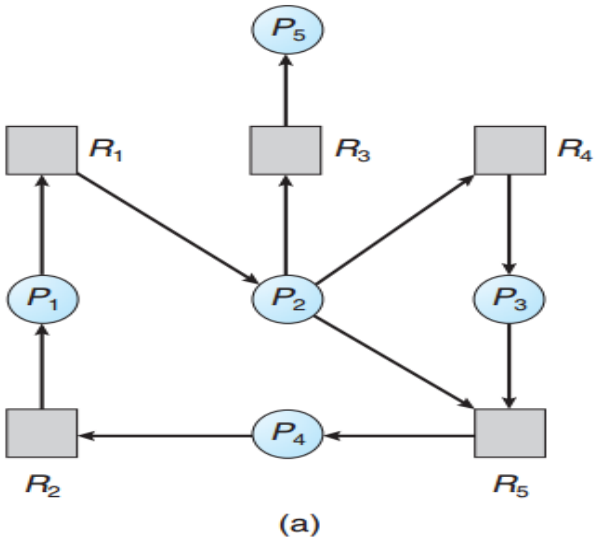
• An algorithm that examines the state of the system to determine whether a deadlock has occurred
• An algorithm to recover from the deadlock

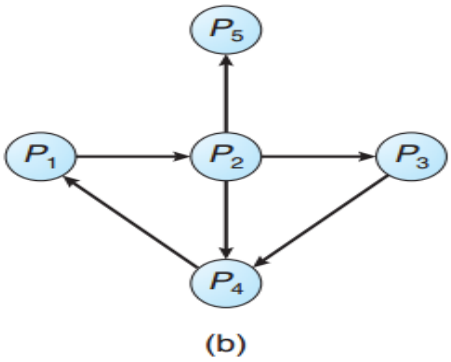
In the following discussion, we elaborate on these two requirements as they pertain to systems with only a *single instance* of each resource type, as well as to systems with *several instances* of each resource type.

1.Single Instance of each Resource Type(*wait-for graph)*

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

(n 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 resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq).
In the following Figure , we present a resource-allocation graph and the corresponding wait-for graph.





(a) Resource-allocation graph.

(b) Corresponding wait-for graph.

1.Several Instance of each Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with *multiple instances* of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. 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 *R j*.

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed:

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.

For example: To illustrate this algorithm, we consider a system with five processes *P*0 through *P*4 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 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	

In this, Work = [0, 0, 0] &
Finish = [false, false, false, false, false]

1- i=0 is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
   Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].
2- i=2 is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
   Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3]
   Finish = [true, false, true, false, false].
3- i=1 is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
   Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &
   Finish = [true, true, true, false, false].
4- i=3 is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
   Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
   Finish = [true, true, true, true, false].
5- i=4 is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
   Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
   Finish = [true, true, true, true, true].

Since Finish is a vector of all true it means there is no deadlock in this example.we will find that the sequence <*P*0, *P*2, *P*1, *P*3, *P*4> results in *Finish*[*i*] == *true* for all *i*.