

UNIT 4 – DEADLOCKS

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**.

SYSTEM MODEL

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/O devices are examples of resource types.

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. The number of resources requested may not exceed the total number of resources available in the system.

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, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources or logical resources

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.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION

Necessary 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 non-sharable 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 $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , \dots , P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_0 .

Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called **System Resource-Allocation Graph**

The graph consists of a set of vertices V and a set of edges E . The set of vertices V is partitioned into two different types of nodes:

- $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system.
- $R = \{R_1, R_2, \dots, R_m\}$ the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process P_i .

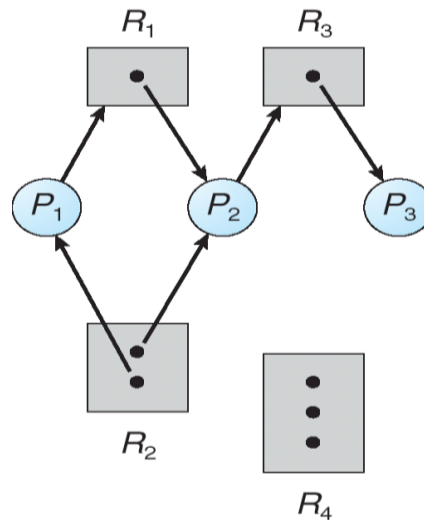
- A directed edge $P_i \rightarrow R_j$ is called a Request Edge.
- A directed edge $R_j \rightarrow P_i$ is called an Assignment Edge.

Pictorially each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, each instance is represented as a dot within the rectangle.

A request edge points to only the rectangle R_j , whereas an assignment edge must also designate one of the dots in the rectangle.

When process P_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure depicts the following situation.



The sets P, K and E:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1 .
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3 .
- Process P_3 is holding an instance of R_3 .

If the graph does contain a cycle, then a deadlock may exist.

- If each resource type has exactly **one instance**, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- If each resource type has **several instances**, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, the resource-allocation graph depicted in below figure:

Suppose that process P_3 requests an instance of resource type R_2 . Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. At this point, two minimal cycles exist in the system:

1. $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
2. $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

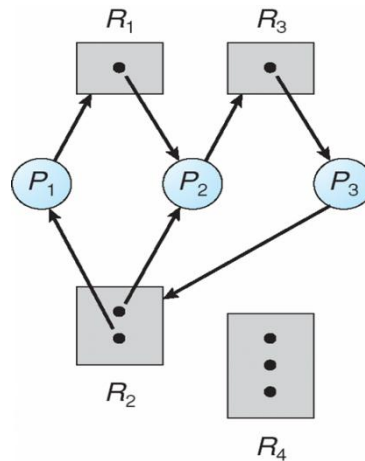


Figure: Resource-allocation graph with a deadlock.

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Consider the resource-allocation graph in below Figure. In this example also have a cycle:

P1->R1->P3->R2->P1

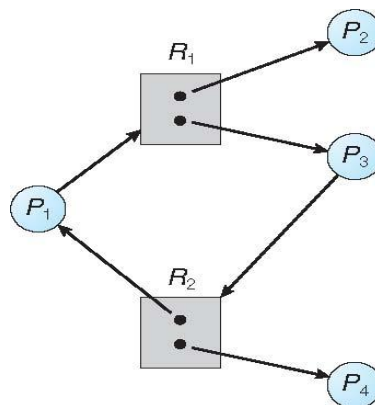


Figure 7.4 Resource-allocation graph with a cycle but no deadlock

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

The deadlock problem can be handled in one of three ways:

1. Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. Allow the system to enter a deadlocked state, detect it, and recover.
3. Ignore the problem altogether and pretend that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme.

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock-avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to detect and recover from deadlocks, then the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

DEADLOCK PREVENTION

Deadlock can be prevented by ensuring that at least one of the four necessary conditions cannot hold.

Mutual Exclusion

- The mutual-exclusion condition must hold for non-sharable resources. Sharable resources, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Ex: Read-only files is 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.
- Deadlocks cannot prevented by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, then guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- Another 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.

Ex:

- 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 again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

The two main disadvantages of these protocols.

1. Resource utilization may be low, since resources may be allocated but unused for a long period.
2. Starvation is possible.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.

To ensure that this condition does not hold, the following protocols can be used:

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, 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.

If a process requests some resources, first check whether they are available. If they are, allocate them.

If they are not available, check whether they are allocated to some other process that is waiting for additional resources. If so, 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

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, let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types. Assign a unique integer number to each resource type, which allows to compare two resources and to determine whether one precedes another in ordering. Formally, it is defined as a one-to-one function

$F: R \rightarrow N$, where N is the set of natural numbers.

Example: if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

$$F(\text{tape drive}) = 1$$

$$F(\text{disk drive}) = 5$$

$$F(\text{printer}) = 12$$

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 - R_i . After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.

DEADLOCK AVOIDANCE

- To avoid deadlocks an additional information is required about how resources are to be requested. With the 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.
- The various algorithms that use this approach differ in the amount and type of information required. The simplest 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. Such an algorithm defines the *deadlock-avoidance approach*.

Safe State

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. A system is in a safe state only if there exists a safe sequence.

Safe sequence: A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$.

In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks as shown in figure. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe states

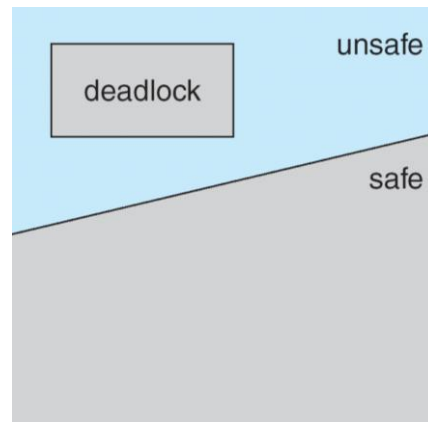


Figure: Safe, unsafe, and deadlocked state spaces.

Resource-Allocation-Graph Algorithm

- If a resource-allocation system have only one instance of each resource type, then a variant of the resource-allocation graph is used for deadlock avoidance.
- In addition to the request and assignment edges, a new type of edge is introduced, called a claim edge.
- A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j 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 P_i requests resource R_j , the claim edge $P_i \rightarrow R_j$ is converted to a request edge. When a resource R_j is released by P_i the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

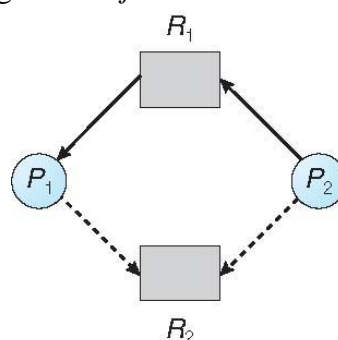


Figure: Resource-allocation graph for deadlock avoidance.

Note that the resources must be claimed a priori in the system. That is, before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

Now suppose that process P_i requests resource R_j . The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph.

There is need to check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, consider the resource-allocation graph as shown above. Suppose that P_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle in the graph.

A cycle, indicates that the system is in an unsafe state. If P_1 requests R_2 , and P_2 requests R_1 , then a deadlock will occur.

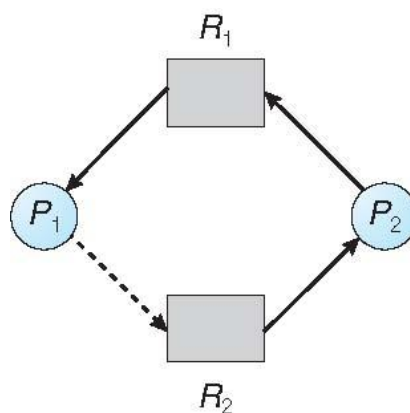


Figure: An unsafe state in a resource-allocation graph

Banker's Algorithm

The Banker's algorithm is applicable to a resource allocation system with multiple instances of each resource type.

- 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.

To implement the banker's algorithm the following data structures are used.

Let n = number of processes, and m = number of resources types

Available: A vector of length m indicates the number of available resources of each type. If $\text{available}[j] = k$, there are k instances of resource type R_j available.

Max: An $n \times m$ matrix defines the maximum demand of each process. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j

Allocation: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j

Need: An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Safety Algorithm

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 $i = 0, 1, \dots, n-1$

2. Find an index i such that both:

(a) Finish[i] = false

(b) $\text{Need}_i \leq \text{Work}$

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

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.

Let Request _{i} be the request vector for process P_i . If Request _{i} [j] == k , then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

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. Have the system 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 $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Example

Consider a system with five processes P_0 through P_4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0 the following snapshot of the system has been taken:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	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*

	<u>Need</u>
	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 currently in a safe state. Indeed, the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $\text{Request}_1 = (1, 0, 2)$. Decide whether this request can be immediately granted.

Check that $\text{Request} \leq \text{Available}$

$$(1, 0, 2) \leq (3, 3, 2) \Rightarrow \text{true}$$

Then pretend that this request has been fulfilled, and the following new state is arrived.

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.

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

Single Instance of Each Resource Type

- If all resources have only a single instance, then define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph.
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q .

Example: In below Figure, a resource-allocation graph and the corresponding wait-for graph is presented.

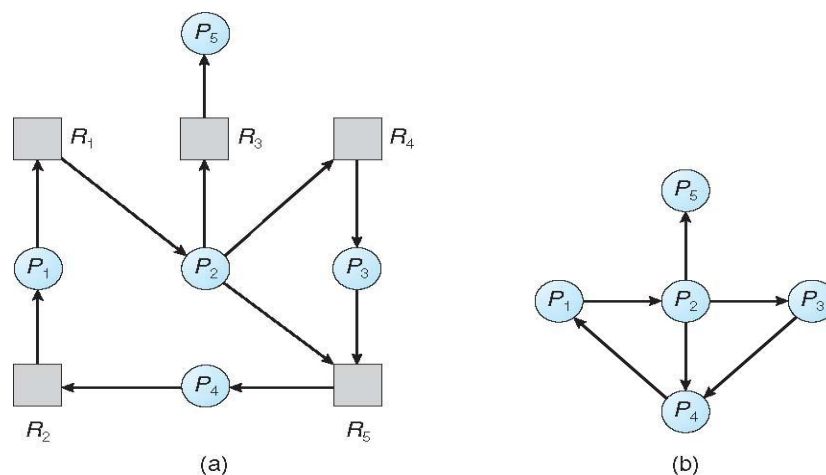


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.

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.

Several Instances of a Resource Type

A deadlock detection algorithm that is applicable to 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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $Request[i][j]$ equals k , then process P_i is requesting k more instances of resource type R_j .

Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n , respectively Initialize:
 - (a) $Work = Available$
 - (b) For $i = 1, 2, \dots, n$, 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
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 $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

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 that, at time T_0 , the following resource-allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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	

After executing the algorithm, Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i

Suppose now that process P_2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	<u>Request</u>
	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

The system is now deadlocked. Although we can reclaim the resources held by process P_0 , the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3 , and P_4 .

Detection-Algorithm Usage

The detection algorithm can be invoked on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

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

The system recover from the deadlock automatically. There are two options for breaking a deadlock : (i) Abort one or more processes to break the circular wait.
(ii) To preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

1. **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
2. **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. 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.
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, we 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, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.
3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?