Chapter 5
DEADLOCK

5.1 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 of which consists of some number of identical instances. Memory space, CPU cycles, files, and **1 / 0** devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type ***CPU*** has two instances. Similarly, the resource type *printer* may have five instances.

If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

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 only has two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: If the request cannot be granted immediately (for example, 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.

Examples are the **request** and **release device, open** and **close file,** and **allocate** and **free memory** system calls. Request and release of other resources can be accomplished through the *wait* and *signal* operations on semaphores.Therefore, for each use, the operating system checks to make sure that the using process has requested and been allocated the resource. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by only 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 resource (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors). However, other types of events may result in deadlocks.

To illustrate a deadlock state, we consider a system with three tape drives. Suppose that there are three processes, each holding one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving processes competing for the same resource type.

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

7.2 Deadlock Characterization

It should be obvious that deadlocks are undesirable. In a deadlock, processesnever finish executing and system resources are tied up, preventing other jobs from ever starting. Before we discuss the various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks.

7.2.1 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: There must exist a process that is holding at least one resource and is 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: There must exist a set {P_0, P_1, ..., P,) of waiting processes such that P_o is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P2, ..., 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 $P_{0.}$

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 7.4, however, that it is useful to consider each condition separately.

5.2.2 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. The set of vertices V is partitioned into two different types of nodes P = {P1, P2, ..., P,), the set consisting of all the active processes in the system, and R = {Rl, R2, ..., Rm), the set consisting of all resource types in the system.

A directed edge from process ***Pi*** to resource type Rj is denoted by *Pi* + Ri; it signifies that process Pi requested an instance of resource type R, and is currently waiting for that resource. A directed edge from resource type ***Rj*** to process Pi is denoted by R, + Pi; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi -+ Ri is called a request edge; a directed edge R, -, Pi is called an assignment edge.

Pictorially, we represent each process Pi as a circle, and each resource type Ri as a square. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge points to only the square Xi, whereas an assignment edge must also designate one of the dots in the square.

When process Pi requests an instance of resource type R,, 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, and as a result the assignment edge is deleted.

The resource-allocation graph shonw in Figure 5.1 depicts the following situation.
0 The sets ***P,*** R, 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 R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type ***R4***

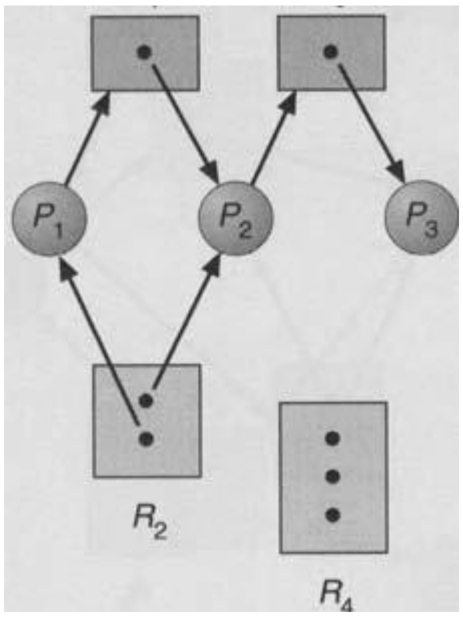


Figure 7.1 Resource-allocation graph

Process states:

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

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains 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. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock 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, let us return to the resource-allocation graph depicted in Figure 7.1. Suppose that process *P3* requests an instance of resource type *R2*. Since no resource instance is currently available, a request edge PJ + *R2* is added to the graph (Figure 5.2). **At** this point, two minimal cycles exist in the system:

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

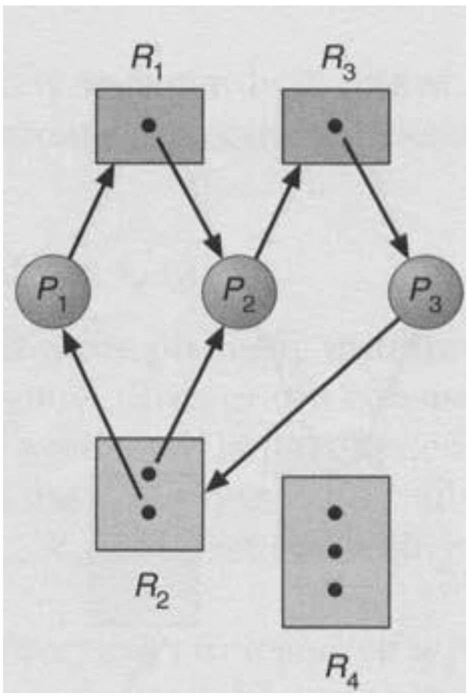


Figure 5.2 Resource-allocation graph with a deadlock.

Processes P_1, P_2, and P3 are deadlocked. Process P2 is waiting for the resource *R3,* which is held by process P3. Process P3, on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.

Now consider the resource-allocation graph in Figure **7.3.** In this example, we also have a cycle

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

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

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with the deadlock problem.

5.3 Methods for Handling Deadlocks

Principally, there are three different methods for dealing with the deadlock problem:

We can use a protocol to ensure that the system will never enter a deadlock state.

- We can allow the system to enter a deadlock state and then recover.

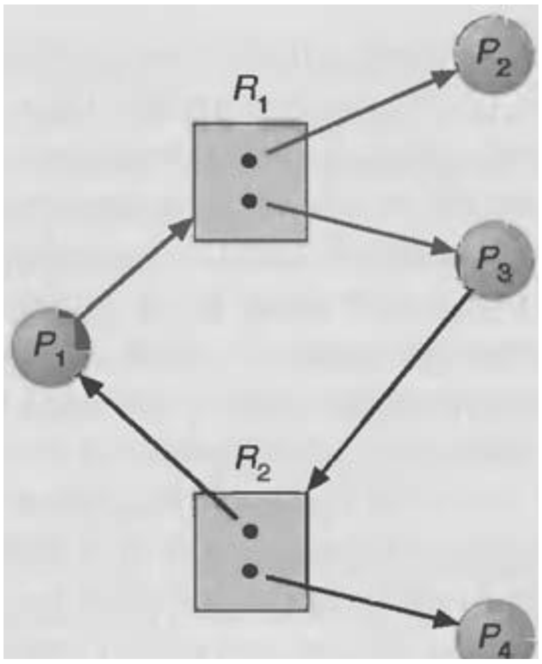


Figure 5.3 Resource-allocation graph with a cycle but no deadlock.

We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX. We shall elaborate briefly on each method. Then, in Sections 5.4 to 5.8, we shall present detailed algorithms.

To ensure that deadlocks never occur, the system can use either a deadlockprevention or a deadlock-avoidance scheme. ***Deadlock prevention*** is a set of methods for ensuring that at least one of the necessary conditions (Section 5.2.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 7.4.

Deadlock avoidance, on the other hand, 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, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed. We discuss these schemes in Section 5.5.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 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 (if a deadlock has indeed occurred). If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where 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 the deterioration of the system performance, because resources are being held by processes that cannot run, and because more and more processes, as they make requests for resources, enter a deadlock state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method does not seem to be a viable approach to the deadlock problem, it is nevertheless used in some operating systems. In many systems, deadlocks occur infrequently (say, once per year); thus, it is cheaper to use this method instead of the costly deadlock prevention, deadlock avoidance, or deadlock detection and recovery methods that must be used constantly. Also, there are circumstances in which the system is in a frozen state without it being in a deadlock state. As an example of this situation, consider a real-time process running at the highest priority (or any process running on a non-preemptive scheduler) and never returning control to the operating system.

5.4 Deadlock Prevention

As we noted in Section 5.2.1, 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. Let us elaborate on this approach by examining each of the four necessary conditions separately.

5.4.1 Mutual Exclusion

The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, 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, it is not possible to prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable.

5.4.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must 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. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, 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 tape drive to a disk file, sorts the disk 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 tape drive, disk file, and the 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 tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape 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.

There are two main disadvantages to these protocols. First, *resource utilization* may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, *starvation* is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

5.4.3 No Preemption

The third necessary condition is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process that is holding some resources requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. That is, these resources are implicitly released. 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 available, 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 not either available or 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 while it was waiting. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

5.4.4 Circular Wait

One way to ensure that the circular-wait 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.

Let R = {R1, R2, ..., R,} 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 Ri if and only if F(R,) > F(Ri). If several instances of the same resource type are needed,a single request for all of them must be issued. 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, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) ***1*** F(R,). If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be {Po, PI, ..., P,), where ***Pi*** is waiting for a resource Xi, which is held by process Pi+1. (Modulo arithmetic is used on the indexes, so that P, is waiting for a resource R, held by Po.) Then, since process Pi+l is holding resource Ri while requesting resource Ri+l, we must have F(Ri) < F(Ri+l), for all i. But this condition means that F(Ro) < F(R1) < ... < F(R,,) < F(Ro). By transitivity, F(Ro) < F(Ro), which is impossible. Therefore, there can be no circular wait. Note that, the function F should be defined according to the normal order of usage of the resources in a system. For example, since the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive) < F(printer).

5.5 Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 7.4, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

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, we might be told that process ***P*** will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The various algorithms 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 a priori information, for each process, about the maximum number of resources of each type that may be requested, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defies the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

5.5.1 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 <PI, PL, ..., ***Pn>*** is a safe sequence for the current allocation state if, for each Pi, the resources that ***Pi*** can still request can be satisfied by the currently available resources plus the resources held by all the ***Pi,*** with j < i. In this situation, if the resources that process ***Pi*** needs are not immediately available, then ***Pi*** can wait until all ***Pi*** have finished. When they have finished, ***Pi*** can obtain all of its

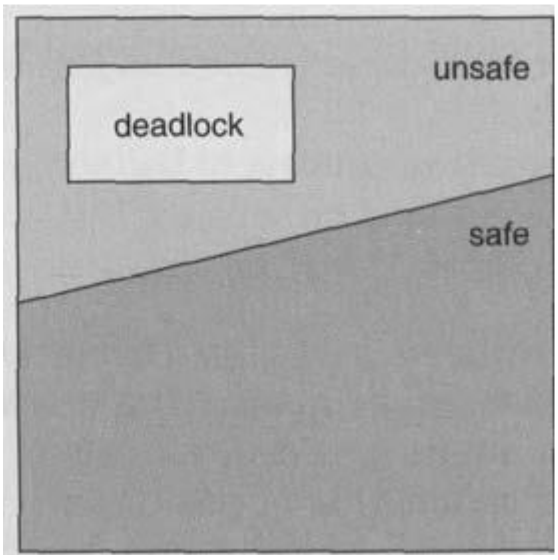


Figure 5.4 Safe, unsafe, and deadlock state spaces.

needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, *Pi+l* 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 deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 7.4). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

To illustrate, we consider a system with 12 magnetic tape drives and **3** processes: *PO, PI,* and *P2.* Process *Po* requires 10 tape drives, process *PI* may need as many as 4, and process *P2* may need up to 9 tape drives. Suppose that, at time *to,* process *Po* is holding 5 tape drives, process *PI* is holding 2, and process *P2* is holding 2 tape drives. (Thus, there are **3** free tape drives.)

	Maximum Needs	Current Needs
Po	*10*	***5***
P1	4	2
P2	9	2

At time *to,* the system is in a safe state. The sequence <*PI,* Po, *P2*> satisfies the safety condition, since process *PI* can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process *Po* can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process *P2* could get all its tape drives and return them (the system will then have all 12 tape drives available).

Note that it is possible to go from a safe state to an unsafe state. Suppose that, at time *tl,* process *P2* requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process *PI* can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process ***Po*** is allocated ***5*** tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process Po must wait. Similarly, process ***P2*** may request an additional 6 tape drives and have to wait, resulting in a deadlock.

Our mistake is in granting the request from process ***P2*** for 1 more tape drive. If we had made ***P2*** wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock situation. Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

Note that, in this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock-avoidance algorithm.

5.5.2 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in Section 7.2.2 can be used for deadlock avoidance.

In addition to the request and assignment edges, 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 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 Ri is released by ***Pi,***th e assignment edge Rj + ***Pi***i s reconverted to a claim edge ***Pi*** --+ Rj. We note that the resources must be claimed a priori in the system. That is, 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.

Suppose that process ***Pi*** requests resource Rj. The request can be granted only if converting the request edge ***Pi*** --+ R, to an assignment edge Rj + ***Pi*** does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of ***n2*** 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. Therefore, process ***Pi*** will have to wait for its requests to be satisfied.

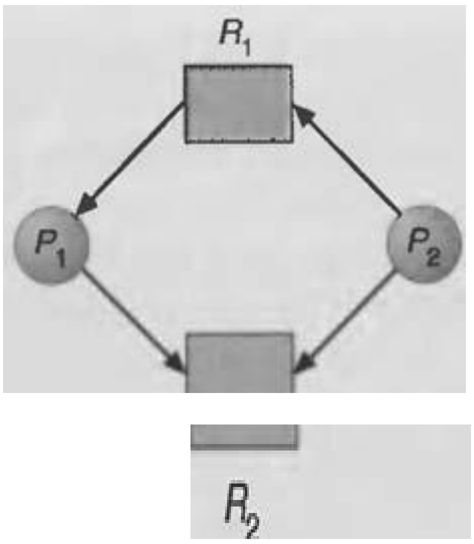


Figure 5.5 Resource-allocation graph for deadlock avoidance.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 5.5. 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 5.6). A cycle indicates that the system is in an unsafe state. If PI requests R2, and P2 requests R1, then a deadlock will occur.

7.5.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the ***banker's algorithm.*** The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

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. Let *n* be the number of processes in the system and ***m*** be the number of resource types. We need the following data structures:

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

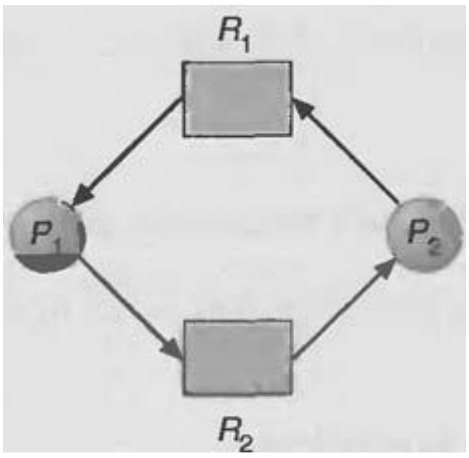


Figure 5.6 An unsafe state in a resource-allocation graph.

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

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

- ***Need:*** An ***n x m*** matrix indicates the remaining resource need of each process. If ***Need[i,j]*** = ***k,*** then ***Pi*** may need ***k*** more instances of resource type ***Rj*** to complete its task. Note that ***Need[i,j]*** = ***Max[i,j]*** - ***Allocation[i,j].***

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, let us establish some notation. Let X and ***Y*** be vectors of length ***n.*** We say that X 5 ***Y*** if and only if ***X[i]*** 5 ***Y[i]*** for all ***i*** = 1, 2, ..., ***n.*** For example, if X = (1, 7, 3, 2) and ***Y*** = (0,3,2,1), thenYIX.Y<XifY<XandY#X.

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as ***Allocationi*** and ***Needi,*** respectively. The vector ***Allocationi*** specifies the resources currently allocated to process ***Pi;*** the vector ***Needi*** specifies the additional resources that process ***Pi*** may still request to complete its task.

5.5.3.1 Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n,*** respectively. Initialize ***Work*** := ***Available*** and ***Finish[i] :=false*** for ***i*** = 1,2, ..., ***n.***

2. Find an ***i*** such that both
   a. ***Finish[i] =false***
   b. ***Needi*** 5 ***Work***

I£ 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.

This algorithm may require an order of ***m*** x ***n2*** operations to decide whether a state is safe.

5.5.3.2 Resource-Request Algorithm

Let ***Requesti*** be the request vector for process ***Pi.*** If ***Requesti[j]*** = ***k,*** then process ***Pi*** wants ***k*** instances of resource type ***Rj.*** When a request for resources is made by process ***Pi,*** the following actions are taken:

1. If ***Requesti*** *5* ***Needi,*** 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 - Requesti;***
Allocationi := ***Allocationi*** + ***Requesti;***
Needi := ***Needi - Requesti;***

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 ***Requesti*** and the old resource-allocation state is restored.

5.5.3.3 An Illustrative Example

Consider a system with five processes ***Po*** through ***P4*** and three resource types ***A, B, C.*** Resource type ***A*** has 10 instances, resource type ***B*** has **5** instances, and resource type ***C*** has **7** instances. Suppose that, at time *To,* the following snapshot of the system has been taken:

	Allocation	***Max***	***Available***
	A B C	***A B C***	***A B C***
Po	0 10	**753**	**3 3** 2
P1	200	**3** 2 2	
PL	302	902	
P3	211	222	
p4	002	**4 3 3**	

The content of the matrix Need is defied to be Max - Allocation and is

	Need
	ABC
Po	743
P1	122
PL	600

P3	011
P	431

We claim that the system is currently in a safe state. Indeed, the sequence <PI, P3, P4, PL, Po> satisfies the safety criteria.

Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request, = (1,0,2). To decide whether this request can be immediately granted, we first check that Requestl *5* Available (that is, (1,0,2) *5* (3,3,2)), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	Need	Available
	ABC	ABC	ABC
Po	010	743	230
P1	3 0 2	020	
PL	302	600	
P3	211	011	
p4	002	431	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <PI, P3, P4, PO, P2> satisfies our safety requirement. Hence, we can immediately grant the request of process PI.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available. A request for (0,2,0) by PO cannot be granted, even though the resources are available, since the resulting state is unsafe.

5.6 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 must 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. At this point, however, let us note that a detection and recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

5.6.1 Single Instance of Each Resource Type

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 nodes of type resource and collapsing the appropriate edges. More precisely, an edge from Pi to Pi in a wait-for graph implies that process Pi is waiting for process Pi 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 --+ & and *Rq* + Pj for some resource *Rq*. For example, in Figure 7.7, we 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 graphcontains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically to *invoke an algorithm* that searches for a cycle in the graph.

An algorithm to detect a cycle in a graph requires an order of *n2* operations, where *n* is the number of vertices in the graph.

5.6.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-detection algorithm that we describe next is applicable to to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 5.5.3):

Available: A vector of length *m* indicates the number of available resources of each type.

Allocation: An *n x m* matrix defies the number of resources of each type currently allocated to each process.

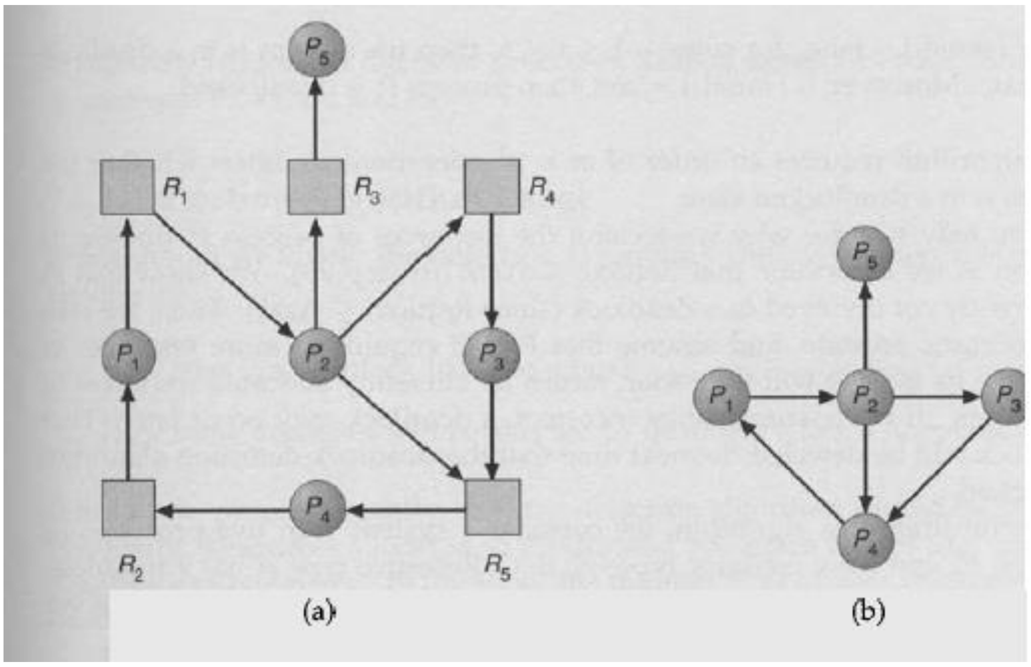
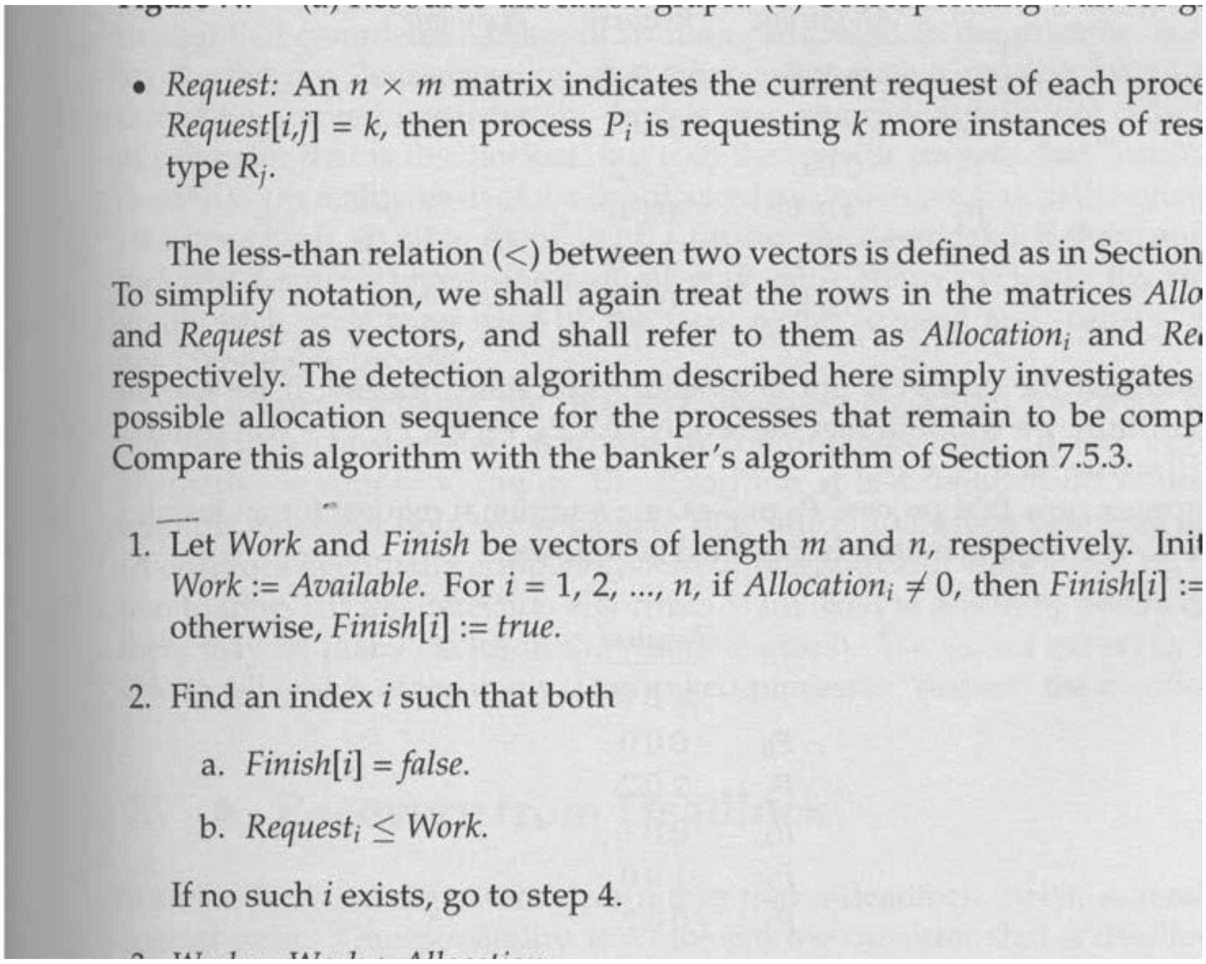


Figure: 5. 7 a) Resource allocation graph(.b) Corresnondinn wait-for graph



- *Request:* An $n \times m$ matrix indicates the current request of each proce *Request*$[i,j] = k$, then process P_i is requesting k more instances of res type R_j.

The less-than relation ($<$) between two vectors is defined as in Section To simplify notation, we shall again treat the rows in the matrices *Allo* and *Request* as vectors, and shall refer to them as *Allocation*$_i$ and *Re* respectively. The detection algorithm described here simply investigates possible allocation sequence for the processes that remain to be comp Compare this algorithm with the banker's algorithm of Section 7.5.3.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Init *Work* := *Available*. For $i = 1, 2, ..., n$, if *Allocation*$_i \neq 0$, then *Finish*$[i]$:= 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.

4. If Finish[i] = false, for some i, 1 *5 **i** 5* n, then the system is in a deadlock state. Moreover, if Finish[i] =false, then process Pi is deadlocked.

This algorithm requires an order of ***m*** x n2 operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process Pi (in step 3) as soon as we determine that Requesti *5* Work (in step 2b). We know that Pi is currently not involved in a deadlock (since Requesti *5* Work). Thus, we take an optimistic attitude, and assume that Pi will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time that the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes PO through P4 and three resource types A, B, C. Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time To, we have the following resource-allocation state:

	Allocation	Request	Available
	ABC	ABC	ABC
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	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <Po, P2, P3, PI, P4> will result 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
	ABC
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes PI, P2, P3, and Pq.

7.6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends 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. **In** addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks can come into being only when some process makes a request that cannot be granted immediately. It is possible that this request is the final request that completes a chain of waiting processes. **In** the extreme, we could invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. **In** this case, we can identify not only the set of processes that is deadlocked, but also the specific process that "caused" the deadlock. **(In** reality each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may cause many cycles in the resource graph, each cycle completed by the most recent request, and "caused" by the one identifiable process.

Of course, invoking the deadlock-detection algorithm for every request may incur a considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at less frequent intervals – for example, once per hour, or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and will cause CPU utilizationto drop.) If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph. We would generally not be able to tell which of the many deadlocked processes "caused the deadlock.

5.7 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system ***recover*** from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to pre-empt some resources from one or more of the deadlocked processes.

5.7.1 Process Termination

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

Abort all deadlocked processes:

This method clearly will break the deadlock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed later.

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. Notice that aborting a process may not be easy. If the process was in the midst of updating a file, terminating it in the middle will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the state of the printer to a correct state before proceeding with the printing of the next job.

If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems. The question is basically an economic one; we should abort those processes the termination of which will incur the minimum cost. Unfortunately, the term *minimum* cost is not a precise one. Many factors may determine 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 type 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

5.7.2 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 deadlock process is holding, and the amount of time
a deadlocked 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, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

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? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a ***starvation*** situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.The most common solution is to include the number of rollbacks in the cost factor.

5.8 Combined Approach to Deadlock Handling

Researchers have argued that none of the basic approaches for handling deadlocks (prevention, avoidance, and detection) alone is appropriate for the entire spectrum of resource-allocation problems encountered in operating systems. One possibility is to combine the three basic approaches, allowing the use of the optimal approach for each class of resources in the system. The proposed method is based on the notion that resources can be partitioned into classes that are hierarchically ordered. A resource-ordering technique (Section 5.4.4) is applied to the classes. Within each class, the most appropriate technique for handling deadlocks can be used.

It is easy to show that a system that employs this strategy will not be subjected to deadlocks. Indeed, a deadlock cannot involve more than one class, since the resource-ordering technique is used. Within each class, one of the basic approaches is used. Consequently, the system is not subject to deadlocks. To illustrate this technique, we consider a system that consists of the following four classes of resources:

Internal resources:

Resources used by the system, such as a process control block Central memory: Memory used by a user job Job resources: Assignable devices (such as a tape drive) and files Swappable space: Space for each user job on the backing store

One mixed deadlock solution for this system orders the classes as shown, and uses the following approaches to each class: Internal resources: Prevention through resource ordering can be used, since run-time choices between pending requests are unnecessary.

Central memory: Prevention through preemption can be used, since a job can always be swapped out, and the central memory can be preempted. Job resources: Avoidance can be used, since the information needed about resource requirements can be obtained from the job-control cards. Swappable space: Preallocation can be used, since the maximum storage requirements are usually known.

This example shows how various basic approaches can be mixed within the framework of resource ordering, to obtain an effective solution to the deadlock