# 3

# DEADLOCKS

Computer systems are full of resources that can only be used by one process at a time. Common examples include printers, tape drives, and slots in the system's internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file system table slot will invariably lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a CD. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the CD recorder first and is also granted it. Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks can also occur across machines. For example, many offices have a local area network with many computers connected to it. Often devices such as scanners, CD recorders, printers, and tape drives are connected to the network as shared resources, available to any user on any machine. If these devices can be reserved remotely (i.e., from the user's home machine), the same kind of deadlocks can occur as described above. More complicated situations can cause deadlocks involving three, four, or more devices and users.

Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one's record, we also have a deadlock. Thus deadlocks can occur on hardware resources or on software resources.

In this chapter, we will look at deadlocks more closely, see how they arise, and study some ways of preventing or avoiding them. Although this material is about deadlocks in the context of operating systems, they also occur in database systems and many other contexts in computer science, so this material is actually applicable to a wide variety of

multiprocess systems. A great deal has been written about deadlocks. Two bibliographies on the subject have appeared in *Operating Systems Review* and should be consulted for references (Newton, 1979; and Zobel, 1983). Although these bibliographies are old, most of the work on deadlocks was done well before 1980, so they are still useful.

# 3.1 RESOURCES

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as **resources**. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). A computer will normally have many different resources that can be acquired. For some resources, several identical instances may be available, such as three tape drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can be used by only a single process at any instant of time.

## 3.1.1 Preemptable and Nonpreemptable Resources

Resources come in two types: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 32 MB of user memory, one printer, and two 32-MB processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished with the computation, it exceeds its time quantum and is swapped out.

Process *B* now runs and tries, unsuccessfully, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither can proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.

A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD, CD recorders are not preemptable at an arbitrary moment.

In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another. Thus our treatment will focus on nonpreemptable resources.

The sequence of events required to use a resource is given below in an abstract form.

1.  Request the resource.
2.  Use the resource.
3.  Release the resource.

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes, it is as good as blocked, because it cannot do any useful work. In our further treatment, we will assume that when a process is denied a resource request, it is put to sleep.

The exact nature of requesting a resource is highly system dependent. In some systems, a **request** system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the operating system knows about are special files that only one process can have open at a time. These are opened by the usual **open** call. If the file is already in use, the caller is blocked until its current owner closes it.

## 3.1.2 Resource Acquisition

For some kinds of resources, such as records in a database system, it is up to the user processes to manage resource usage themselves. One possible way of allowing user management of resources is to associate a semaphore with each resource. These semaphores are all initialized to 1. Mutexes can be used equally well. The three steps listed above are then implemented as a **down** on the semaphore to acquire the resource, using the resource, and finally an **up** on the resource to release it. These steps are shown in Fig. 3-1(a).

```
typedef int semaphore;              typedef int semaphore;
semaphore resource_1;               semaphore resource_1;
                                    semaphore resource_2;

void process_A(void)                void process_A(void)
{                                   {
    down(&resource_1);                  down(&resource_1);
    use_resource_1();                   down(&resource_2);
    up(&resource_1);                    use_both_resources();
}                                       up(&resource_2);
                                        up(&resource_1);
                                    }
```

<div align="center">(a)                                    (b)</div>

**Figure 3-1.** Using a semaphore to protect resources. (a) One resource. (b) Two resources.

Sometimes processes need two or more resources. They can be acquired sequentially, as shown in Fig. 3-1(b). If more than two resources are needed, they are just acquired one after another.

So far, so good. As long as only one process is involved, everything works fine. Of course, with only one process, there is no need to formally acquire resources, since there is no competition for them.

Now let us consider a situation with two processes, *A* and *B*, and two resources. Two scenarios are depicted in Fig. 3-2. In Fig. 3-2(a), both processes ask for the resources in the same order. In Fig. 3-2(b), they ask for them in a different order. This difference may seem minor, but it is not.

```
typedef int semaphore;                    typedef int semaphore;
semaphore resource_1;                      semaphore resource_1;
semaphore resource_2;                      semaphore resource_2;

void process_A(void) {                     void process_A(void) {
    down(&resource_1);                         down(&resource_1);
    down(&resource_2);                         down(&resource_2);
    use_both_resources( );                     use_both_resources( );
    up(&resource_2);                           up(&resource_2);
    up(&resource_1);                           up(&resource_1);
}                                          }

void process_B{void) {                     void process_B{void) {
    down(&resource_1);                         down(&resource_2);
    down(&resource_2);                         down(&resource_1);
    use_both_resources( );                     use_both_resources();
    up(&resource_2);                           up(&resource_1);
    up(&resource_1);                           up(&resource_2);
}                                          }
           (a)                                        (b)
```

**Figure 3-2.** (a) Deadlock-free code. (b) Code with a potential deadlock.

In Fig. 3-2(a), one of the processes will acquire the first resource before the other one. That process will then successfully acquire the second resource and do its work. If the other process attempts to acquire resource 1 before it has been released, the other process will simply block until it becomes available.

In Fig. 3-2(b), the situation is different. It might happen that one of the processes acquires both resources and effectively blocks out the other process until it is done. However, it might also happen that process *A* acquires resource 1 and process *B* acquires resource 2. Each one will now block when trying the acquire the other one. Neither process will ever

run again. This situation is a deadlock.

Here we see how what appears to be a minor difference in coding style— which resource to acquire first—turns out to make the difference between the program working and the program failing in a hard-to-detect way. Because deadlocks can occur so easily, a lot of research has gone into ways to deal them. This chapter discusses deadlocks in detail and what can be done about them.

# 3.2 INTRODUCTION TO DEADLOCKS

Deadlock can be defined formally as follows:

> *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software.

## 3.2.1 Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.

2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.

3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once? Can a process hold a resource and ask for another? Can resources be preempted? Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions.

## 3.2.2 Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 3-3(a), resource $R$ is currently assigned to process A.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 3-3(b), process $B$ is waiting for resource $S$. In Fig. 3-3(c) we see a deadlock: process $C$ is waiting for resource $T$, which is currently held by process $D$. Process $D$ is not about to release resource $T$ because it is waiting for resource $U$, held by $C$. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is $C$–$T$–$D$–$U$–$C$.
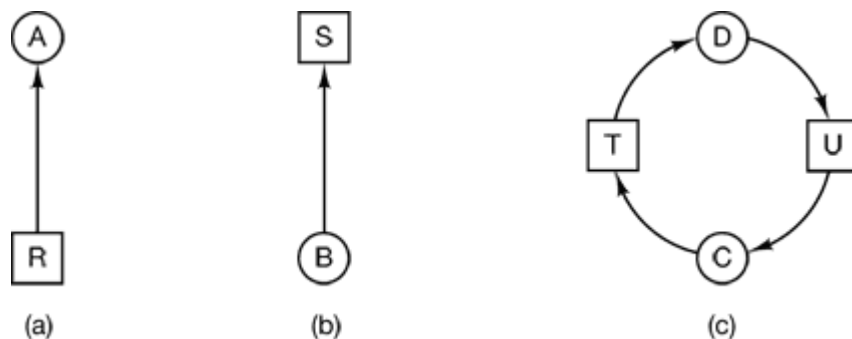


**Figure 3-3.** Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, $A$, $B$, and $C$, and three resources $R$, $S$, and $T$. The requests and releases of the three processes are given in Fig. 3-4(a)-(c). The operating system is free to run any unblocked process at any instant, so it could decide to run $A$ until $A$ finished all its work,
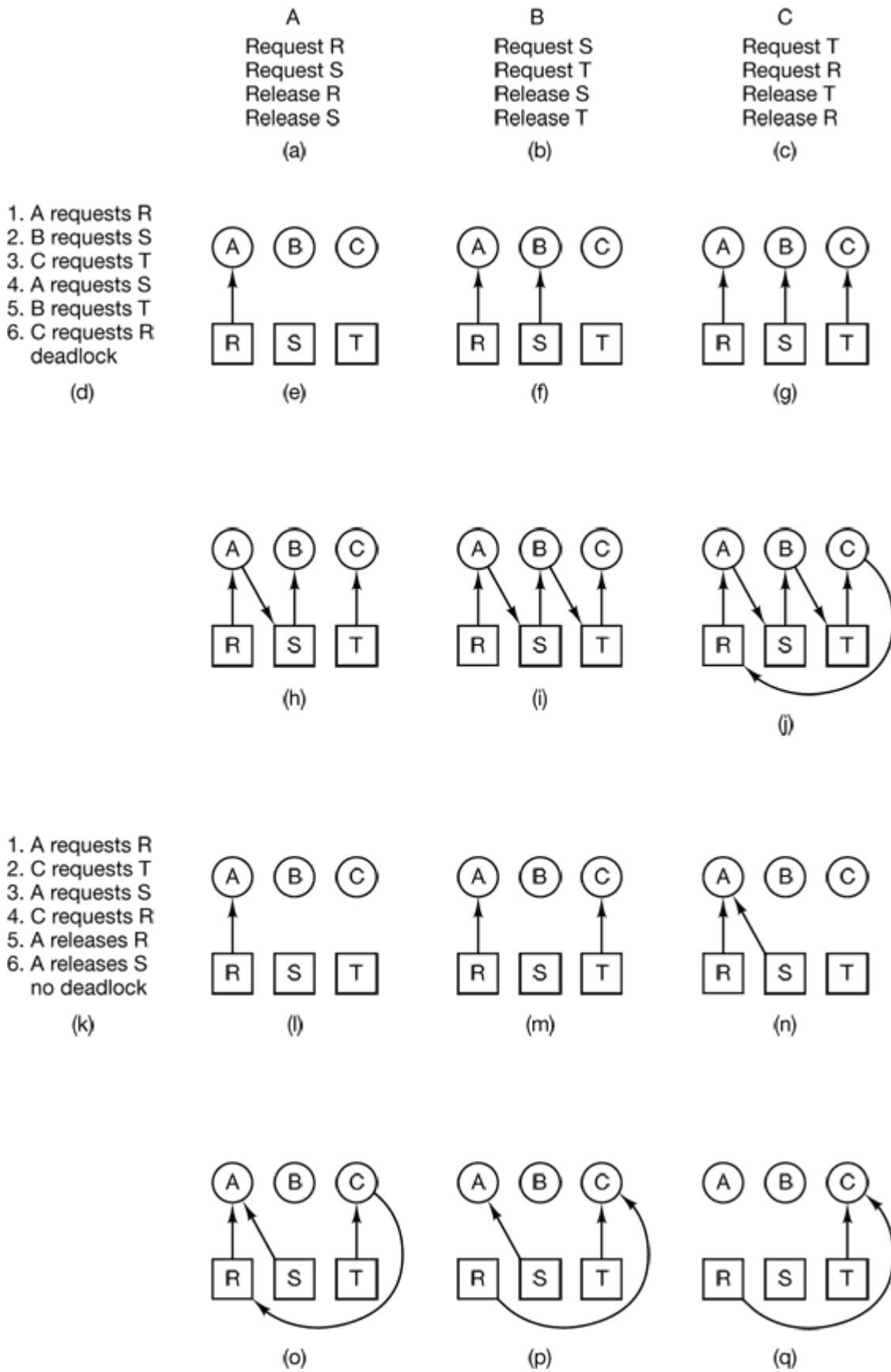
then run *B* to completion, and finally run C.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 3-4(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 3-4(e)-(j), After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 3-4(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 3-4(j).

However, as we have already mentioned, the operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe. In Fig. 3-4, if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig. 3-4(k) instead or Fig. 3-4(d). This sequence leads to the resource graphs of Fig. 3-4(l)-(q), which do not lead to deadlock.

After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* should eventually block when requesting *T* no deadlock can occur. *B* will just wait until *C* is finished.

A
Request R
Request S
Release R
Release S
(a)

B
Request S
Request T
Release S
Release T
(b)

C
Request T
Request R
Release T
Release R
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock
(d)

(e)

(f)

(g)

(h)

(i)

(j)

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock
(k)

(l)

(m)

(n)

(o)

(p)

(q)

**Figure 3-4.** An example of how deadlock occurs and how it can be avoided.

Later in this chapter we will study a detailed algorithm for making allocation decisions that do not lead to deadlock. For the moment, the point to understand is that resource graphs are a tool that let us see if a given request/release sequence leads to deadlock. We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock. Although our treatment of resource graphs has been for the case of a single resource of each type, resource graphs can also be generalized to handle multiple resources of the same type (Holt, 1972). In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.

2. Detection and recovery. Let deadlocks occur, detect them, and take action.

3. Dynamic avoidance by careful resource allocation.

4. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

We will examine each of these methods in turn in the next four sections.

# 3.3 THE OSTRICH ALGORITHM

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all[†]. Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.

To make this contrast more specific, most operating systems potentially suffer from deadlocks that are not even detected, let alone automatically broken. The total number of processes in a system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 (sub)processes. After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing—a deadlock. The probability of this happening is minuscule, but it *could* happen. Should we abandon processes and the fork call to eliminate the problem?

The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up. Swap space on the disk is another limited resource. In fact, almost every table in the operating system represents a finite resource. Should we abolish all of these because it might happen that a collection of *n* processes might each claim 1/*n* of the total, and then each try to claim another one?

Most operating systems, including UNIX and Windows, just ignore the problem on the assumption that most users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything. If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes, as we will see shortly. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom. Under these conditions, general solutions are hard to find.

# 3.4 DEADLOCK DETECTION AND RECOVERY

A second technique is detection and recovery. When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact. In this section we will look at some of the ways deadlocks can be detected and some of the ways recovery from them can be handled.

## 3.4.1 Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later using a different method.

For such a system, we can construct a resource graph of the sort illustrated in Fig. 3-3. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

As an example of a more complex system than the ones we have looked at so far, consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.

3. Process $C$ holds nothing but wants $S$.

4. Process $D$ holds $U$ and wants $S$ and $T$.

5. Process $E$ holds $T$ and wants $V$.

6. Process $F$ holds $W$ and wants $S$.

7. Process $G$ holds $V$ and wants $U$.

The question is: "Is this system deadlocked, and if so, which processes are involved?"

To answer this question, we can construct the resource graph of Fig. 3-5(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 3-5 (b). From this cycle, we can see that processes $D$, $E$, and $G$ are all deadlocked. Processes $A$, $C$, and $F$ are not deadlocked because $S$ can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.
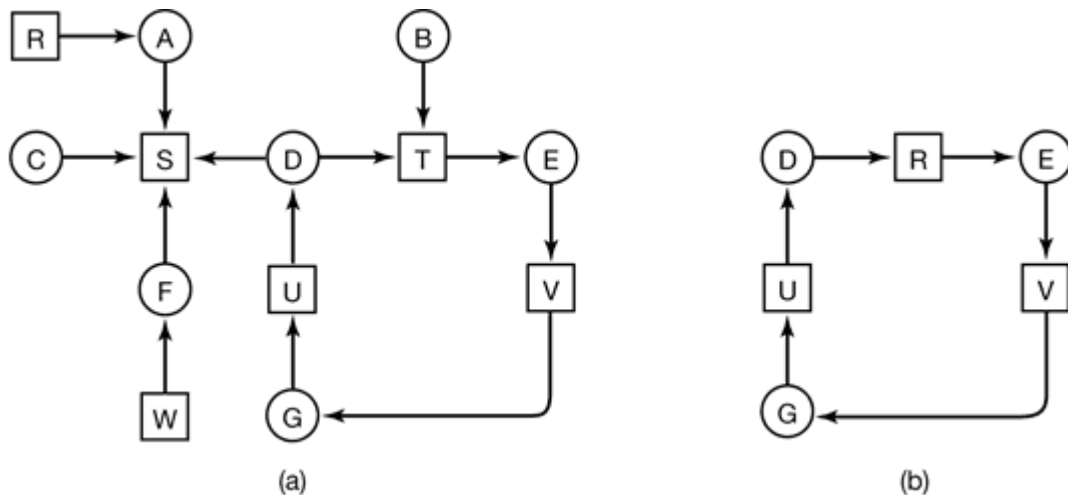


Figure 3-5. (a) A resource graph. (b) A cycle extracted from (a).

Although it is relatively simple to pick out the deadlocked processes by eye from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known. Below we will give a simple one that inspects a graph and terminates either when it has found a cycle or when it has shown that none exist. It uses one data structure, $L$, a list of nodes. During the algorithm, arcs will be marked to indicate that they have already been inspected, to prevent repeated inspections.

The algorithm operates by carrying out the following steps as specified:

1. For each node, $N$ in the graph, perform the following 5 steps with $N$ as the starting node.

2. Initialize $L$ to the empty list, and designate all the arcs as unmarked.

3. Add the current node to the end of $L$ and check to see if the node now appears

in $L$ two times. If it does, the graph contains a cycle (listed in $L$) and the algorithm terminates.

4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.

5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.

6. We have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

What this algorithm does is take each node, in turn, as the root of what it hopes will be a tree, and does a depth-first search on it. If it ever comes back to a node it has already encountered, then it has found a cycle. If it exhausts all the arcs from any given node, it backtracks to the previous node. If it backtracks to the root and cannot go further, the subgraph reachable from the current node does not contain any cycles. If this property holds for all nodes, the entire graph is cycle free, so the system is not deadlocked.

To see how the algorithm works in practice, let us use it on the graph of Fig. 3-5(a). The order of processing the nodes is arbitrary, so let us just inspect them from left to right, top to bottom, first running the algorithm starting at $R$ then successively, $A$, $B$, $C$, $S$, $D$, $T$, $E$, $F$, and so forth. If we hit a cycle, the algorithm stops.

We start at $R$ and initialize $L$ to the empty list. Then we add $R$ to the list and move to the only possibility, $A$, and add it to $L$, giving $L = [R, A]$. From $A$ we go to $S$, giving $L = [R, A, S]$. $S$ has no outgoing arcs, so it is a dead end, forcing us to backtrack to $A$. Since $A$ has no unmarked outgoing arcs, we backtrack to $R$, completing our inspection of $R$.

Now we restart the algorithm starting at $A$, resetting $L$ to the empty list. This search, too, quickly stops, so we start again at $B$. From $B$ we continue to follow outgoing arcs until we get to $D$, at which time $L = [B, T, E, V, G, U, D]$. Now we must make a (random) choice. If we pick $S$ we come to a dead end and backtrack to $D$. The second time we pick $T$ and update $L$ to be $[B, T, E, V, G, U, D, T]$, at which point we discover the cycle and stop the algorithm.

This algorithm is far from optimal. For a better one, see (Even, 1979). Nevertheless, it demonstrates that an algorithm for deadlock detection exists.
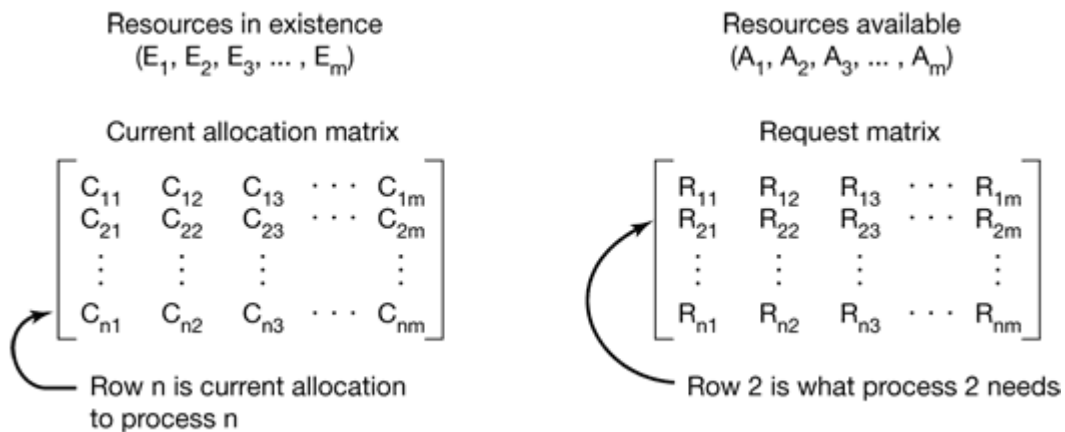
## 3.4.2 Deadlock Detection with Multiple Resource of Each Type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock

among $n$ processes. $P_1$ through $P_n$. Let the number of resource classes be $m$, with $E_1$ resources of class 1, $E_2$ resources of class 2, and generally, $E_i$ resources of class $i$ ($1 < i < m$). $E$ is the **existing resource vector**. It gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.

At any instant, some of the resources are assigned and are not available. Let $A$ be the **available resource vector**, with $A_i$ giving the number of instances of resource $i$ that are currently available (i.e., unassigned). If both of our two tape drives are assigned, $A_1$ will be 0.

Now we need two arrays, $C$, the **current allocation matrix**, and $R$, the **request matrix**. The $i$-th row of $C$ tells how many instances of each resource class $P_i$ currently holds. Thus $C_{ij}$ is the number of instances of resource $j$ that are held by process $i$. Similarly, $R_{ij}$ is the number of instances of resource $j$ that $P_i$ wants. These four data structures are shown in Fig. 3-6.

Resources in existence
$(E_1, E_2, E_3, \dots, E_m)$

Resources available
$(A_1, A_2, A_3, \dots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

**Figure 3-6.** The four data structures needed by the deadlock detection algorithm.

An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

In other words, if we add up all the instances of the resource $j$ that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.

The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors $A$ and $B$ to mean that each element of $A$ is less than or equal to the corresponding element of $B$. Mathematically, $A \leq B$ holds if and only if $A_i \leq B_i$ for $1 < i < m$.

Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked.
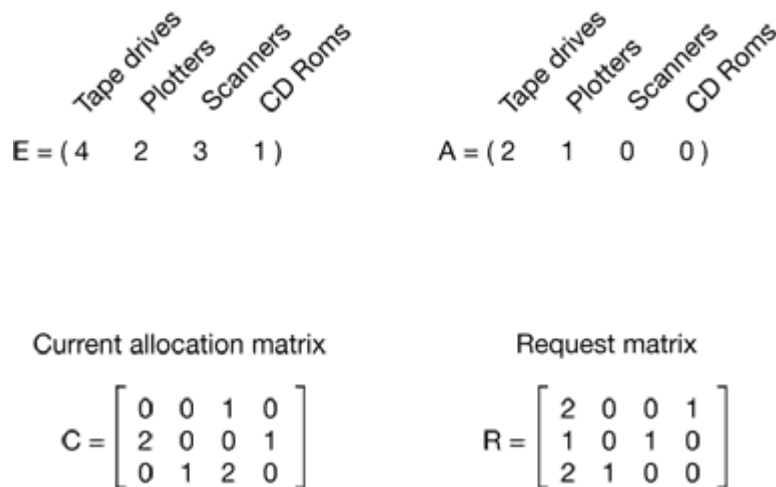
The deadlock detection algorithm can now be given, as follows.

1. Look for an unmarked process, $P_i$, for which the $i$-th row of $R$ is less than or equal to $A$.
2. If such a process is found, add the $i$-th row of $C$ to $A$, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

What the algorithm is doing in step 1 is looking for a process that can be run to completion. Such a process is characterized as having resource demands that can be met by the currently available resources. The selected process is then run until it finishes, at which time it returns the resources it is holding to the pool of available resources. It is then marked as completed. If all the processes are ultimately able to run, none of them are deadlocked. If some of them can never run, they are deadlocked. Although the algorithm is nondeterministic (because it may run the processes in any feasible order), the result is always the same.

As an example of how the deadlock detection algorithm works, consider Fig. 3-7. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanner, and CD-ROM drive. Process 1 has one scanner. Process 2 has two tape drives and a CD-ROM drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the $R$ matrix.

$$E = (4 \quad 2 \quad 3 \quad 1) \qquad\qquad A = (2 \quad 1 \quad 0 \quad 0)$$

Current allocation matrix          Request matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Figure 3-7.** An example for the deadlock detection algorithm.

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied. The first one cannot be satisfied because there is no CD-ROM drive available. The second cannot be satisfied either, because there is no scanner free. Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving

$$A = (2\ 2\ 2\ 0)$$

At this point process 2 can run and return its resources, giving

$$A = (4\ 2\ 2\ 1)$$

Now the remaining process can run. There is no deadlock in the system.

Now consider a minor variation of the situation of Fig. 3-7. Suppose that process 2 needs a CD-ROM drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system is deadlocked.

Now that we know how to detect deadlocks, the question of when to look for them comes up. One possibility is to check every time a resource request is made. This is certain to detect them as early as possible, but it is potentially expensive in terms of CPU time. An alternative strategy is to check every $k$ minutes, or perhaps only when the CPU utilization has dropped below some threshold. The reason for considering the CPU utilization is that if enough processes are deadlocked, there will be few runnable processes, and the CPU will often be idle.

## 3.4.3 Recovery from Deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. In this section we will discuss various ways of recovering from deadlock. None of them are especially

attractive, however.

## Recovery through Preemption

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. In many cases, manual intervention may be required, especially in batch processing operating systems running on mainframes.

For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.

The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

## Recovery through Rollback

If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes **checkpointed** periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process. To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence of checkpoint files are accumulated.

When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

## Recovery through Killing Processes

The crudest, but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.

Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources. In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs. For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed part way through, the first run has no influence on the second run.

On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some record in the database, running it once, killing it, and then running it again will add 2 to the record, which is incorrect.

# 3.5 DEADLOCK AVOIDANCE

In the discussion of deadlock detection, we tacitly assumed that when a process asks for resources, it asks for them all at once (the $R$ matrix of Fig. 3-6). In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe. Thus the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance. In this section we examine ways to avoid deadlock by careful resource allocation.
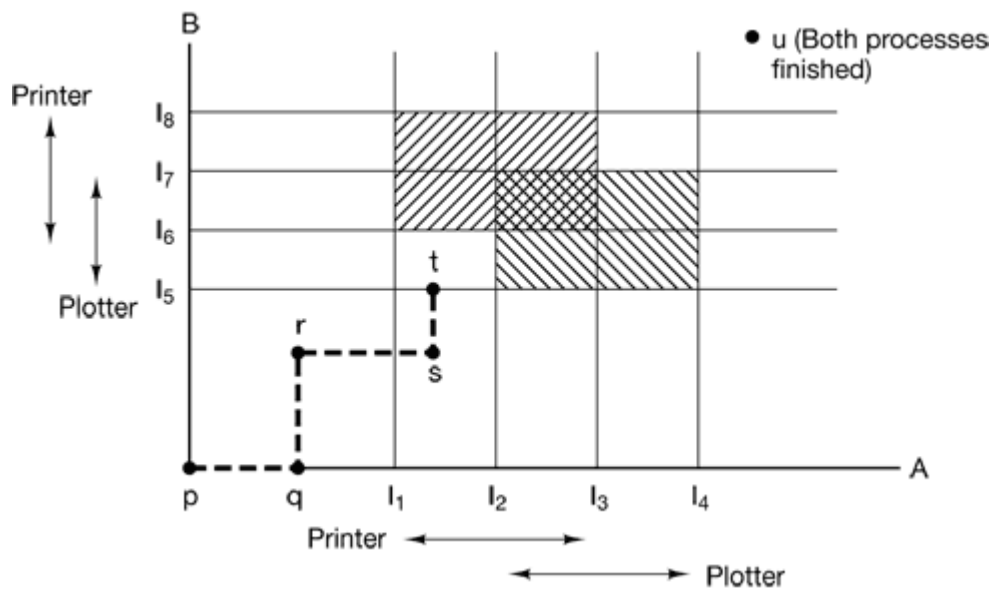
## 3.5.1 Resource Trajectories

The main algorithms for doing deadlock avoidance are based on the concept of safe states. Before describing the algorithms, we will make a slight digression to look at the concept of safety in a graphic and easy-to-understand way. Although the graphical approach does not translate directly into a usable algorithm, it gives a good intuitive feel for the nature of the problem.

In Fig. 3-8 we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process $A$. The vertical axis represents the number of instructions executed by process $B$. At $I_1$ $A$ requests a printer; at $I_2$ it needs a plotter. The printer and plotter are released at $I_3$ and $I_4$, respectively. Process $B$ needs the plotter from $I_5$ to $I_7$ and the printer from $I_6$ to

$I_8$.

Every point in the diagram represents a joint state of the two processes. Initially, the state is at $p$, with neither process having executed any instructions. If the scheduler chooses to run $A$ first, we get to the point $q$, in which $A$ has executed some number of instructions, but $B$ has executed none. At point $q$ the trajectory becomes vertical, indicating that the scheduler has chosen to run $B$. With a single processor, all paths must be horizontal or vertical, never diagonal. Furthermore, motion is always to the north or east, never to the south or west (processes cannot run backward).

When $A$ crosses the $I_1$ line on the path from $r$ to $s$, it requests and is granted the printer. When $B$ reaches point $t$, it requests the plotter.



**Figure 3-8.** Two process resource trajectories.

The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter, and is equally impossible.

If the system ever enters the box bounded by $I_1$ and $I_2$ on the sides and $I_5$ and $I_6$ top and bottom, it will eventually deadlock when it gets to the intersection of $I_2$ and $I_6$. At this point $A$ is requesting the plotter and $B$ is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered. At point $t$ the only safe thing to do is run process $A$ until it gels to $I_4$. Beyond that, any trajectory to $u$ will do.

The important thing to see here is at point $t$ $B$ is requesting a resource. The system must decide whether to grant it or not. If the grant is made, the system will enter an unsafe region and eventually deadlock. To avoid the deadlock, $B$ should be suspended until $A$ has

requested and released the plotter.

## 3.5.2 Safe and Unsafe States

The deadlock avoidance algorithms that we will study use the information of Fig. 3-6. At any instant of time, there is a current state consisting of $E$, $A$, $C$, and $R$. A state is said to be safe if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately. It is easiest to illustrate this concept by an example using one resource. In Fig. 3-9(a) we have a state in which $A$ has 3 instances of the resource but may need as many as 9 eventually. $B$ currently has 2 and may need 4 altogether, later. Similarly, $C$ also has 2 but may need an additional 5. A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 3

(a)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |

Free: 1

(b)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | — |
| C | 2 | 7 |

Free: 5

(c)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | — |
| C | 7 | 7 |

Free: 0

(d)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | — |
| C | 0 | — |

Free: 7

(e)

**Figure 3-9.** Demonstration that the state in (a) is safe.

The state of Fig. 3-9(a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler could simply run $B$ exclusively, until it asked for and got two more instances of the resource, leading to the state of Fig. 3-9(b). When $B$ completes, we get the state of Fig. 3-9(c). Then the scheduler can run $C$ leading eventually to Fig. 3-9(d). When $C$ completes, we get Fig. 3-9(e). Now $A$ can get the six instances of the resource it needs and also complete. Thus the state of Fig. 3-9(a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. 3-10(a), but this time $A$ requests and gets another resource, giving Fig. 3-10(b). Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run $B$ until it asked for all its resources, as shown in Fig. 3-10(c).

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 3

(a)

| | Has | Max |
|---|---|---|
| A | 4 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 2

(b)

| | Has | Max |
|---|---|---|
| A | 4 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |

Free: 0

(c)

| | Has | Max |
|---|---|---|
| A | 4 | 9 |
| B | — | — |
| C | 2 | 7 |

Free: 4

(d)

**Figure 3-10.** Demonstration that the state in (b) is not safe.

Eventually, *B* completes and we get the situation of Fig. 3-10(d). At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion. Thus the allocation decision that moved the system from Fig. 3-10(a) to Fig. 3-10(b) went from a safe state to an unsafe state. Running *A* or *C* next starting at Fig. 3-10(b) does not work either. In retrospect, *A*'s request should not have been granted.

It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig. 3-10(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that *A* might release a resource before asking for any more, allowing *C* to complete and avoiding deadlock altogether. Thus the difference between a safe state and an unsafe state is that from a safe state the system can *guarantee* that all processes will finish; from an unsafe state, no such guarantee can be given.

## 3.5.3 The Banker's Algorithm for a Single Resource

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965) and is known as the **banker's algorithm** and is an extension of the deadlock detection algorithm given in Sec. 3.4.1. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied. If granting the request leads to a safe state, it is carried out. In Fig. 3-11(a) we see four customers. *A*, *B*, *C* and *D*, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K dollars). The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

| | Has | Max | | | Has | Max | | | Has | Max |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 6 | | A | 1 | 6 | | A | 1 | 6 |
| B | 0 | 5 | | B | 1 | 5 | | B | 2 | 5 |
| C | 0 | 4 | | C | 2 | 4 | | C | 2 | 4 |
| D | 0 | 7 | | D | 4 | 7 | | D | 4 | 7 |

Free: 10          Free: 2          Free: 1

(a)             (b)             (c)

**Figure 3-11.** Three resource allocation states: (a) Safe. (b) Safe (c) Unsafe.

The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in Fig. 3-11(b). This state is safe because with two units left, the banker can delay any requests except *C*'s, thus letting *C* finish and release all four of his resources. With four units in

hand, the banker can let either *D* or *B* have the necessary units, and so on.

Consider what would happen if a request from *B* for one more unit were granted in Fig. 3-11(b). We would have situation Fig. 3-11(c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not *have* to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

The banker's algorithm considers each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

## 3.5.4 The Banker's Algorithm for Multiple Resources

The banker's algorithm can be generalized to handle multiple resources. Figure 3-12 shows how it works.



| | Process | Tape drives | Plotters | Scanners | CD ROMs |
|---|---|---|---|---|---|
| A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 1 | 0 |
| D | 1 | 1 | 0 | 1 |
| E | 0 | 0 | 0 | 0 |

Resources assigned

| | Process | Tape drives | Plotters | Scanners | CD ROMs |
|---|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 3 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |
| E | 2 | 1 | 1 | 0 |

Resources still needed

E = (6342)
P = (5322)
A = (1020)

**Figure 3-12.** The banker's algorithm with multiple resources.

In Fig. 3-12 we see two matrices. The one on the left shows how many of each resource are currently assigned to each of the five processes. The matrix on the right shows how many resources each process still needs in order to complete. These matrices are just *C* and *R* from Fig. 3-6. As in the single resource case, processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each instant.

The three vectors at the right of the figure show the existing resources, *E*, the possessed resources, *P*, and the available resources, *A*, respectively. From *E* we see that the system has six tape drives, three plotters, four printers, and two CD-ROM drives. Of these, five tape drives, three plotters, two printers, and two CD-ROM drives are currently assigned.

This fact can be seen by adding up the four resource columns in the left-hand matrix. The available resource vector is simply the difference between what the system has and what is currently in use.

The algorithm for checking to see if a state is safe can now be stated.

1.  Look for a row, $R$, whose unmet resource needs are all smaller than or equal to $A$. If no such row exists, the system will eventually deadlock since no process can run to completion.

2.  Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the $A$ vector.

3.  Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.

Now let us get back to the example of Fig. 3-12. The current state is safe. Suppose that process $B$ now requests a printer. This request can be granted because the resulting state is still safe (process $D$ can finish, and then processes $A$ or $E$, followed by the rest).

Now imagine that after giving $B$ one of the two remaining printers, $E$ wants the last printer. Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to deadlock. Clearly $E$'s request must be deferred for a while.

The banker's algorithm was first published by Dijkstra in 1965. Since that time, nearly every book on operating systems has described it in detail. Innumerable papers have been written about various aspects of it. Unfortunately, few authors have had the audacity to point out that although in theory the algorithm is wonderful, in practice it is essentially useless because processes rarely know in advance what their maximum resource needs will be. In addition, the number of processes is not fixed, but dynamically varying as new users log in and out. Furthermore, resources that were thought to be available can suddenly vanish (tape drives can break). Thus in practice, few, if any, existing systems use the banker's algorithm for avoiding deadlocks.

# 3.6 DEADLOCK PREVENTION

Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock? The answer is to go back to the four conditions stated by Coffman et al. (1971)

to see if they can provide a clue. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible (Havender, 1968).

## 3.6.1 Attacking the Mutual Exclusion Condition

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

Unfortunately, not all devices can be spooled (the process table does not lend itself well to being spooled). Furthermore, competition for disk space for spooling can itself lead to deadlock. What would happen if two processes each filled up half of the available spooling space with output and neither was finished producing output? If the daemon was programmed to begin printing even before all the output was spooled, the printer might lie idle if an output process decided to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. In this case we have two processes that have each finished part, but not all, of their output, and cannot continue. Neither process will ever finish, so we have a deadlock on the disk.

Nevertheless, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource when that is not absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

## 3.6.2 Attacking the Hold and Wait Condition

The second of the conditions stated by Coffman et al. looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and

the plotter for an hour.

Nevertheless, some mainframe batch systems require the user to list all the resources on the first line of each job. The system then acquires all resources immediately and keeps them until the job finishes. While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks.

A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.
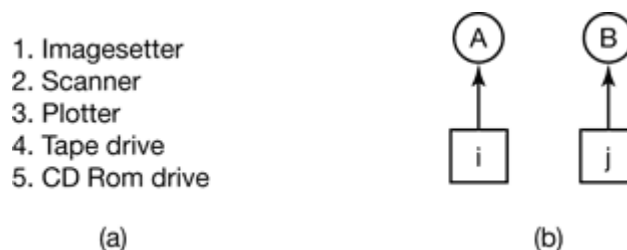
## 3.6.3 Attacking the No Preemption Condition

Attacking the third condition (no preemption) is even less promising than attacking the second one. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst.

## 3.6.4 Attacking the Circular Wait Condition

Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 3-13(a). Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

```
1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

        (a)
```

         (A)        (B)
          ↑          ↑
         [ i ]      [ j ]

         (b)

**Figure 3-13.** (a) Numerically ordered resources. (b) A resource graph.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 3-13(b). We can get a deadlock only if $A$ requests resource $j$ and $B$ requests resource $i$. Assuming $i$ and $j$ are distinct resources, they will have different numbers. If $i > j$, then $A$ is not allowed to request $j$ because that is lower

than what it already has. If $i < j$, then $B$ is not allowed to request $i$ because that is lower than what it already has. Either way, deadlock is impossible.

With multiple processes, the same logic holds. At every instant, one of the assigned resources will be highest. The process holding that resource will never ask for a resource already assigned. It will either finish, or at worst, request even higher numbered resources, all of which are available. Eventually, it will finish and free its resources. At this point, some other process will hold the highest resource and can also finish. In short, there exists a scenario in which all processes finish, so no deadlock is present.

A minor variation of this algorithm is to drop the requirement that resources be acquired in strictly increasing sequence and merely insist that no process request a resource lower than what it is already holding. If a process initially requests 9 and 10, and then releases both of them, it is effectively starting all over so there is no reason to prohibit it from now requesting resource 1.

Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone. When the resources include process table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

The various approaches to deadlock prevention are summarized in Fig. 3-14.

| Condition | Approach |
|---|---|
| Mutual exclusion | Spool everything |
| Hold and wait | Request all resources initially |
| No preemption | Take resources away |
| Circular wait | Order resources numerically |

**Figure 3-14.** Summary of approaches, to deadlock prevention.

# 3.7 OTHER ISSUES

In this section we will discuss a few miscellaneous issues related to deadlocks. These include two-phase locking, nonresource deadlocks, and starvation.

## 3.7.1 Two-Phase Locking

Although both avoidance and prevention are not terribly promising in the general case, for specific applications, many excellent special-purpose algorithms are known. As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called **two-phase locking**. In the first phase the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a lock is encountered during the first phase. In these versions, deadlock can occur.

However, this strategy is not applicable in general. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again. Neither is it acceptable to start over if the process hits read or written messages to the network, updated files, or anything else that cannot be safely repealed. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Many applications cannot be structured this way.

## 3.7.2 Nonresource Deadlocks

All of our work so far has concentrated on resource deadlocks. One process wants something that another process has and must wait until the first one gives it up. Deadlocks can also occur in other situations, however, including those not involving resources at all.

For example, it can happen that two processes deadlock each waiting for the other one to do something. This often happens with semaphores. In Chap. 2 we saw examples in which a process had to do a down on two semaphores, typically *mutex* and another one. If these are done in the wrong order, deadlock can result.

## 3.7.3 Starvation

A problem closely related to deadlock is **starvation**. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some kind of algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Which one should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print

(assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-serve, resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

# 3.8 RESEARCH ON DEADLOCKS

If ever there was a subject that was investigated mercilessly during the early days of operating systems, it was deadlocks. The reason for this is that deadlock detection is a nice little graph theory problem that one mathematically-inclined graduate student can get his jaws around and chew on for 3 or 4 years. All kinds of algorithms were devised, each one more exotic and less practical than the previous one. Essentially, all this research has died out, with only a very occasional new paper appearing (e.g., Karacali et al., 2000). When an operating system wants to do deadlock detection or prevention, which few of them do, they use one of the methods discussed in this chapter.

There is still a little research on distributed deadlock detection, however. We will not treat that here because (1) it is outside the scope of this book, and (2) none of it is even remotely practical in real systems. Its main function seems to be keeping otherwise unemployed graph theorists off the streets.

# 3.9 SUMMARY

Deadlock is a potential problem in any operating system. It occurs when a group of processes each have been granted exclusive access to some resources, and each one wants yet another resource that belongs to another process in the group. All of them are blocked and none will ever run again.

Deadlock can be avoided by keeping track of which states are safe and which are unsafe. A safe state is one in which there exists a sequence of events that guarantee that all processes can finish. An unsafe state has no such guarantee. The banker's algorithm avoids deadlock by not granting a request if that request will put the system in an unsafe state.

Deadlock can be structurally prevented by building the system in such a way that it can never occur by design. For example, by allowing a process to hold only one resource at any instant the circular wait condition required for deadlock is broken. Deadlock can also be prevented by numbering all the resources, and making processes request them in strictly increasing order. Starvation can be avoided by a first-come, first-served allocation policy.

# PROBLEMS

1. Give an example of a deadlock taken from politics.

2. Students working at individual PCs in a computer laboratory send their files to be printed by a server which spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may the deadlock be avoided?

3. In the preceding question which resources are preemptable and which are nonpreemptable?

4. In Fig. 3-1 the resources are returned in the reverse order of their acquisition. Would giving them back in the other order be just as good?

5. Fig. 3-3 shows the concept of a resource graph. Do illegal graphs exist, that is graphs that structurally violate the model we have used of resource image? If so, give an example of one.

6. The discussion of the ostrich algorithm mentions the possibility of process table slots or other system tables filling up. Can you suggest a way to enable a system administrator to recover from such a situation?

7. Consider Fig. 3-4. Suppose that in step (o) $C$ requested $S$ instead of requesting $R$. Would this lead to deadlock? Suppose that it requested both $S$ and $R$?

8. At a crossroads with STOP signs on all four approaches, the rule is that each driver yields the right of way to the driver on his right. This rule is not adequate when four vehicles arrive simultaneously. Fortunately, humans are sometimes capable of acting more intelligently than computers and the problem is usually resolved when one driver signals the driver to his left to go ahead. Can you draw an analogy between this behavior and any of the ways of recovering from deadlock described in Sec. 3.4.3? Why is a problem with such a simple solution in the human world so difficult to apply to a computer system?

9. Suppose that in Fig, 3-6 $C_{ij} + R_{ij} > E_j$ for some $i$. What implications does this have for all the processes finishing without deadlock?

10. All the trajectories in Fig. 3-8 are horizontal or vertical. Can you envision any circumstances in which diagonal trajectories were also possible?

11. Can the resource trajectory scheme of Fig. 3-8 also be used to illustrate the problem of deadlocks with three processes and three resources? If so, how can this be done? If not, why not?

12. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Suggest a practical problem with actually doing this.

13. Take a careful look at Fig. 3-11(b). If $D$ asks for one more unit, does this lead to a safe state or an unsafe one? What if the request came from $C$ instead of $D$?

14. Can a system be in a state that is neither deadlocked nor safe? If so, give an example. If not, prove that all states are either deadlocked or safe.

15. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

16. Consider the previous problem again, but how with $p$ processes each needing a maximum of $m$ resources and a total of $r$ resources available. What condition must hold to make the system deadlock free?

17. Suppose that process $A$ in Fig. 3-12 requests the last tape drive. Does this action lead to a deadlock?

18. A computer has six tape drives, with $n$ processes competing for them. Each process may need two drives. For which values of $n$ is the system deadlock free?

19. The banker's algorithm is being run in a system with $m$ resource classes and $n$ processes. In the limit of large $m$ and $n$, the number of operations that must be performed to check a state for safety is proportional to $m^a n^b$. What are the values of $a$ and $b$?

20. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

|  | Allocated | Maximum | Available |
|---|---|---|---|
| Process A | 1 0 2 1 1 | 1 1 2 1 3 | 0 0 x 1 1 |
| Process B | 2 0 1 1 0 | 2 2 2 1 0 | |
| Process C | 1 1 0 1 0 | 2 1 3 1 0 | |
| Process D | 1 1 1 1 0 | 1 1 2 2 1 | |

What is the smallest value of $x$ for which this is a safe state?

21. A distributed system using mailboxes has two IPC primitives, send and receive. The latter primitive specifies a process to receive from and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Discuss.

22. Two processes, $A$ and $B$, each need three records, 1, 2. and 3, in a database. If $A$ asks for them in the order 1, 2, 3, and $B$ asks for them in the same order, deadlock is not possible. However, if $B$ asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! or 6 possible combinations each process can request the resources. What fraction of all the combinations are guaranteed to be deadlock free?

23. Now reconsider the above problem, but using two-phase locking. Will that eliminate the potential for deadlock? Does it have any other undesirable characteristics, however? If so, which ones?

24. In an electronic funds transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying an amount of money, the account

to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes running in parallel, there is a very real danger that having locked account $x$ it will be unable to lock $y$ because $y$ has been locked by a process now waiting for $x$. Devise a scheme that avoids deadlocks. Do not release an account record until you have completed the transactions. (In other words, solutions that lock one account and then release it immediately if the other is locked are not allowed.)

25. One way to prevent deadlocks is to eliminate the hold-and-wait condition. In the text it was proposed that before asking for a new resource, a process must first release whatever resources it already holds (assuming that is possible). However, doing so introduces the danger that it may get the new resource but lose some of the existing ones to competing processes. Propose an improvement to this scheme.

26. A computer science student assigned to work on deadlocks thinks of the following brilliant way to eliminate deadlocks. When a process requests a resource, it specifies a time limit. If the process blocks because the resource is not available, a timer is started. If the time limit is exceeded, the process is released and allowed to run again. If you were the professor, what grade would you give this proposal and why.

27. Cinderella and the Prince are getting divorced. To divide their property, they have agreed on the following algorithm. Every morning, each one may send a letter to the other's lawyer requesting one item of property. Since it takes a day for letters to be delivered, they have agreed that if both discover that they have requested the same item on the same day, the next day they will send a letter canceling the request. Among their property is their dog. Woofer. Woofer's doghouse, their canary. Tweeter, and Tweeter's cage. The animals love their houses, so it has been agreed that any division of property separating an animal from its house is invalid, requiring the whole division to start over from scratch. Both Cinderella and the Prince desperately want Woofer. So they can go on (separate) vacations, each spouse has programmed a personal computer to handle the negotiation. When they come back from vacation, the computers are still negotiating. Why? Is deadlock possible? Is starvation possible? Discuss.

28. A student majoring in anthropology and minoring in computer science has embarked on a research project to see if African baboons can be taught about deadlocks. He locates a deep canyon and fastens a rope across it, so the baboons can cross hand-over-hand. Several baboons can cross at the same time, provided that they are all going in the same direction. If eastward moving and westward moving baboons ever get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle) because it is impossible for one baboon to climb over another one while suspended over the canyon. If a baboon wants to cross the canyon, he must check to see that no other baboon is currently crossing in the opposite direction. Write a program using semaphores that avoids deadlock. Do not worry about a series of eastward moving baboons holding up the westward moving baboons indefinitely.

29. Repeat the previous problem, but now avoid starvation. When a baboon that wants to cross to the east arrives at the rope and finds baboons crossing to the west, he waits until the rope is empty, but no more westward moving baboons are allowed to start until at least one baboon has crossed the other way.

30. Program a simulation of the banker's algorithm. Your program should cycle through each of the bank clients asking for a request and evaluating whether it is safe or unsafe. Output

a log of requests and decisions to a file.

---

Actually, this bit of folklore is nonsense. Ostriches can run at 60 km/hour and their kick is powerful enough to kill any lion with visions of a big chicken dinner.