MODERN OPERATING SYSTEMS

Fifth Edition

Andrew S. Tanenbaum
Herbert Bos

MODERN OPERATING SYSTEMS

FIFTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

6

DEADLOCKS

Computer systems are full of resources that can be used only by one process at a time. Common examples include printers, cameras, microphones, 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 invariably will 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 scan an object with a 3D scanner and then print the front, top, and side view of the object on a printer. Process *A* requests permission to use the 3D scanner and is granted it. Process *B* is programmed differently and requests the printer first and is also granted it. Now *A* asks for the printer, but the request is suspended until *B* releases it. Unfortunately, instead of releasing the printer, *B* asks for the 3D 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, printers, and (in some data centers) 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), deadlocks of the same kind can occur as described above. More complicated situations can cause deadlocks involving three, four, or more devices and users.

Deadlocks can also occur in a variety of other situations. 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 several kinds of deadlocks, see how they arise, and study some ways of preventing or avoiding them. Although these deadlocks arise in the context of operating systems, they also occur in database systems, big data analytics and many other contexts in computer science, so this material is actually applicable to a wide variety of concurrent systems.

6.1 RESOURCES

A major class of deadlocks involves resources to which some process has been granted exclusive access. These resources include devices, data records, 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 printer) or a piece of information (e.g., a record in a database). A computer will normally have many different resources that a process can acquire. For some resources, several identical instances may be available, such as three printers. 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 must be acquired, used, and released over the course of time.

6.1.1 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 16 GB of user memory, one printer, and two 16-GB 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 the computation, it exceeds its time quantum and is swapped out to SSD or disk.

Process *B* now runs and tries, unsuccessfully as it turns out, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither one 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 potentially causing failure. If a process has begun to scan an object with a 3D scanner, suddenly taking the scanner away from it and

giving it to another process will result in a garbled 3D model of the object. 3D scanners are not preemptable at an arbitrary moment.

Whether a resource is preemptable depends on the context. On a standard PC, memory is preemptable because pages can always be swapped out to SSD or disk to recover it. However, on a low-end device that does not support swapping or paging, deadlocks cannot be avoided by just swapping out a memory hog.

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 abstract sequence of events required to use a resource is given below.

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.

6.1.2 Resource Acquisition

For some kinds of resources, such as records in a database system, it is up to the user processes rather than the system to manage resource usage themselves. One way of allowing this 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, the use of the resource, and finally an up on the resource to release it. These steps are shown in Fig. 6-1(a).

```
typedef int semaphore;
semaphore resource_1;

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

(a)

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

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

(b)

Figure 6-1. 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. 6-1(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. 6-2. In Fig. 6-2(a), both processes ask for the resources in the same order. In Fig. 6-2(b), they ask for them in a different order. This difference may seem minor, but it is not.

In Fig. 6-2(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. 6-2(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 to acquire the other one. Neither process will ever run again. Bad news: 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.

6.1.3 The Dining Philosophers Problem

In 1965, Dijkstra posed and then solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new

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

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

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

(a)

```
    semaphore resource_1;
    semaphore resource_2;

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

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

(b)

Figure 6-2. (a) Deadlock-free code. (b) Code with a potential deadlock.

primitive is by showing how elegantly it solves the dining philosophers problem. We merely use it as a nice illustration of how deadlocks occur and how to avoid them. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 6-3.

The life of a philosopher consists of alternating periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian food to Chinese food, substituting rice for spaghetti and chopsticks for forks.)

Figure 6-4 shows the obvious solution. The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could easily modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This

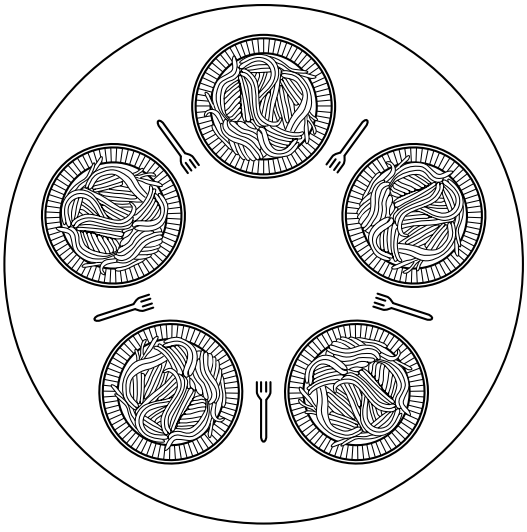

Figure 6-3. Lunch time in the Philosophy Department.

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                           /* philosopher is thinking */
            take_fork(i);                       /* take left fork */
            take_fork((i+1) % N);               /* take right fork; % is modulo operator */
            eat( );                             /* yum-yum, spaghetti */
            put_fork(i);                        /* put left fork back on the table */
            put_fork((i+1) % N);                /* put right fork back on the table */
      }
}
```

Figure 6-4. A nonsolution to the dining philosophers problem.

proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think that if the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small. This

```
#define N               5                       /* number of philosophers */
#define LEFT            (i+N-1)%N               /* number of i's left neighbor */
#define RIGHT           (i+1)%N                 /* number of i's right neighbor */
#define THINKING        0                       /* philosopher is thinking */
#define HUNGRY          1                       /* philosopher is trying to get forks */
#define EATING          2                       /* philosopher is eating */
typedef int semaphore;                          /* semaphores are a special kind of int */
int state[N];                                   /* array to keep track of everyone's state */
semaphore mutex = 1;                            /* mutual exclusion for critical regions */
semaphore s[N];                                 /* one semaphore per philosopher */

void philosopher(int i)                         /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                              /* repeat forever */
        think( );                               /* philosopher is thinking */
        take_forks(i);                          /* acquire two forks or block */
        eat( );                                 /* yum-yum, spaghetti */
        put_forks(i);                           /* put both forks back on table */
    }
}

void take_forks(int i)                          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                               /* enter critical region */
    state[i] = HUNGRY;                          /* record fact that philosopher i is hungry */
    test(i);                                    /* try to acquire 2 forks */
    up(&mutex);                                 /* exit critical region */
    down(&s[i]);                                /* block if forks were not acquired */
}

void put_forks(i)                               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                               /* enter critical region */
    state[i] = THINKING;                        /* philosopher has finished eating */
    test(LEFT);                                 /* see if left neighbor can now eat */
    test(RIGHT);                                /* see if right neighbor can now eat */
    up(&mutex);                                 /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Figure 6-5. A solution to the dining philosophers problem.

observation is true, and in nearly all applications trying again later is not a problem. For example, in the popular Ethernet local area network, if two computers send a packet at the same time, each one waits a random time and tries again; in practice this solution works fine. However, in a few applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. Think about safety control in a nuclear power plant.

One improvement to Fig. 6-4 that has no deadlock and no starvation is to protect the five statements following the call to *think* by a binary semaphore. Before starting to acquire forks, a philosopher would do a down on *mutex*. After replacing the forks, she would do an up on *mutex*. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 6-5 is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is currently eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as its main code, but the other procedures, *take_forks*, *put_forks*, and *test*, are ordinary procedures and not separate processes.

While we may consider the dining philosophers a contrived example, mostly popular with university professors and few others, deadlocks are quite real and may occur easily. A lot of research has gone into ways to deal with them. This chapter discusses the deadlock and starvation problems in detail, as well as what can be done about them.

6.2 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 event 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 are single threaded and that no interrupts are possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by 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 (such as a fork) 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. This kind of deadlock is called a **resource deadlock**. It is probably the most common kind, but it is not the only kind. We first study resource deadlocks in detail and then at the end of the chapter return briefly to other kinds of deadlocks.

6.2.1 Conditions for Resource Deadlocks

More than 50 years ago, Coffman et al. (1971) showed that four conditions must hold for there to be a (resource) 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 that were 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 list 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 resource deadlock to occur. If one of them is absent, no resource 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.

6.2.2 Deadlock Modeling

A year later, 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. A directed 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. 6-6(a), resource *R* is currently assigned to process *A*.

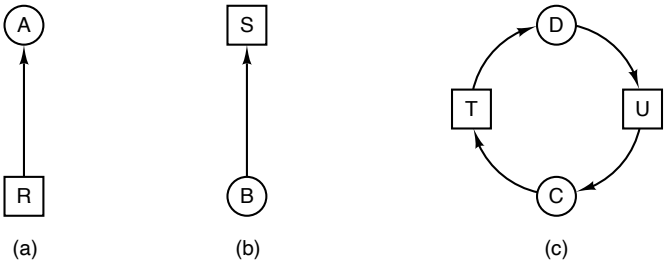

Figure 6-6. Resource -allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

A directed arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 6-6(b), process *B* is waiting for resource *S*. In Fig. 6-6(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 $C - T - D - U - C$.

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. 6-7(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 does 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. 6-7(d). If these six requests are carried out in that order, the six resulting resource graphs are as shown in Fig. 6-7(e)–(j). After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 6-7(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 6-7(j).

However, as we have already mentioned, the operating system is under no obligation to run the processes in any particular order. 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. 6-7, 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. 6-7(k) instead of Fig. 6-7(d). This sequence leads to the resource graphs of Fig. 6-7(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* blocks when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished.

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 lets 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 we 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. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let them occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions.

In the next four sections, we will examine each of these methods in turn.

6.3 THE OSTRICH ALGORITHM

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem.[†] People react to this strategy in different ways. Mathematicians find it 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 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, consider an operating system that blocks the caller when an open system call on a physical device such as a 3D scanner or a

[†] 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, and lions know this.

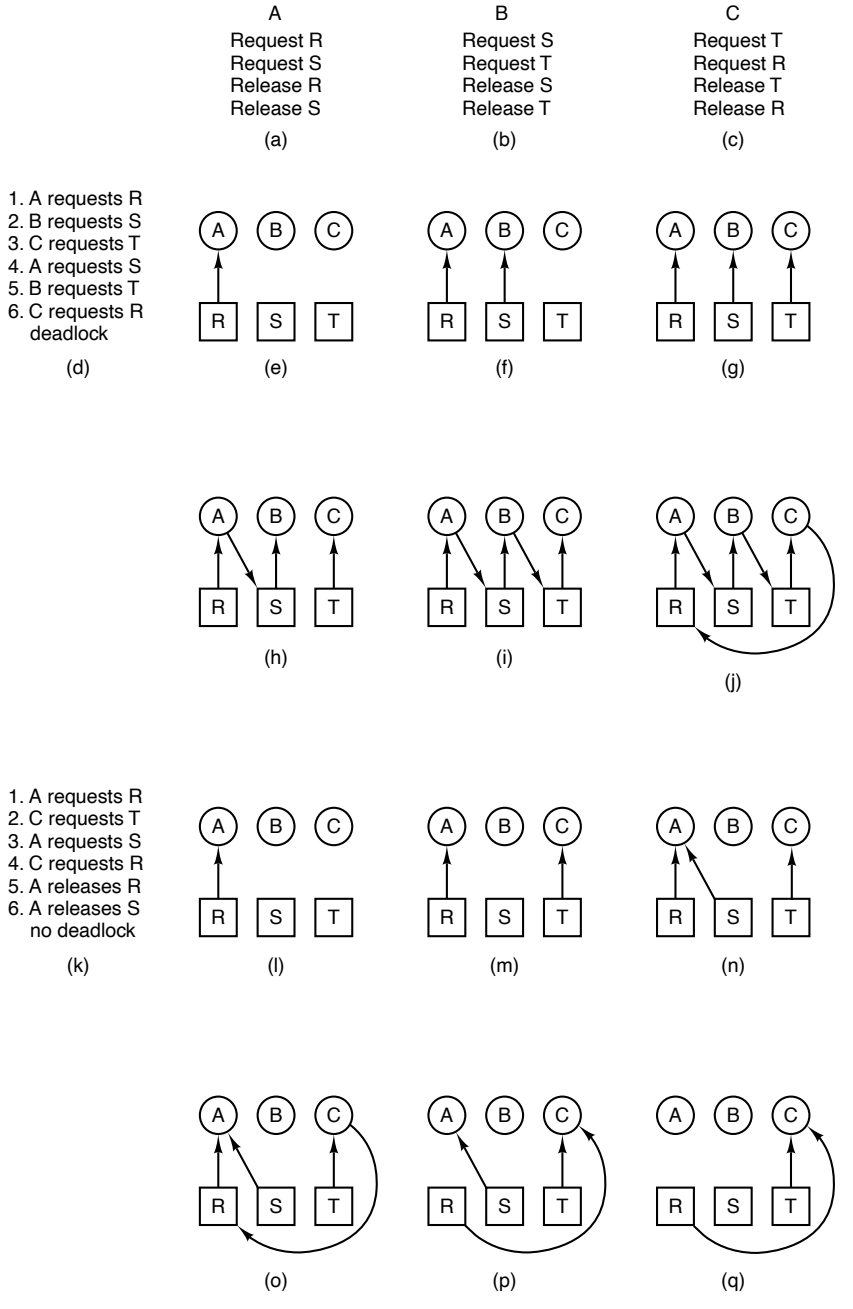

Figure 6-7. An example of how deadlock occurs and how it can be avoided.

printer cannot be carried out because the device is busy. Typically it is up to the device driver to decide what action to take under such circumstances. Blocking or returning an error code are two obvious possibilities. If one process successfully opens the scanner and another successfully opens the printer and then each process tries to open the other one and blocks trying, we have a deadlock. Few current systems will detect this.

6.4 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.

6.4.1 Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: there is only one resource of each type and each device can be acquired by a single process. As an example, consider a system with six resources: a Blu-ray recorder (R), a scanner (S), a tape drive (T), a USB microphone (U), a video camera (V), and a wafer cutter (W)—but no more than one of each class of resource. In other words, we are excluding systems with, say, two scanners for the moment. We will treat them later, using a different method.

The six resources, *R* through *W*, are used by seven processes, *A* though *G*. 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 a resource graph of the sort illustrated in Fig. 6-6. 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 and can continue executing normally.

Drawing a resource graph is straightforward even though our system is now considerably more complex than the simple ones we discussed so far. We show the corresponding resource graph of Fig. 6-8(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 6-8(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. (Note that to make this example more interesting we have allowed processes, namely *D*, to ask for two resources at once.)

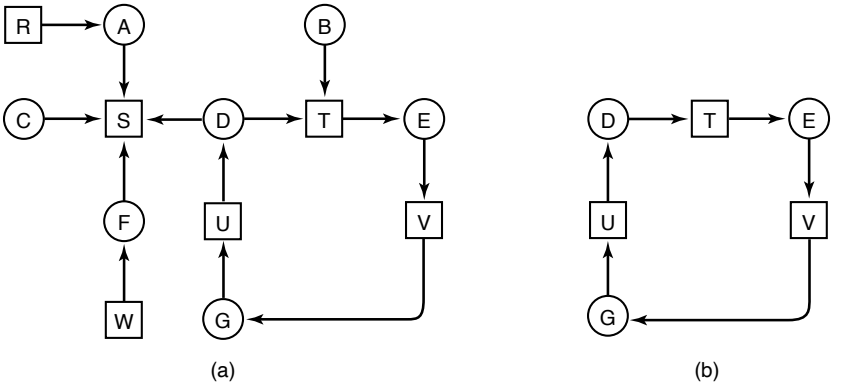

Figure 6-8. (a) A resource graph. (b) A cycle extracted from (a).

Although it is relatively simple to pick out the deadlocked processes by visual inspection 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 exists. It uses one dynamic data structure, *L*, a list of nodes, as well as a list of arcs. During the algorithm, to prevent repeated inspections, arcs will be marked to indicate that they have already been inspected,

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

1. For each node, *N*, in the graph, perform the following five 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. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, 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.

What this algorithm does is take each node, in turn, as the root of what it hopes will be a tree, and do 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. 6-8(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.

6.4.2 Deadlock Detection with Multiple Resources 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 ($1 \le i \le m$). 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 ith 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. 6-9.

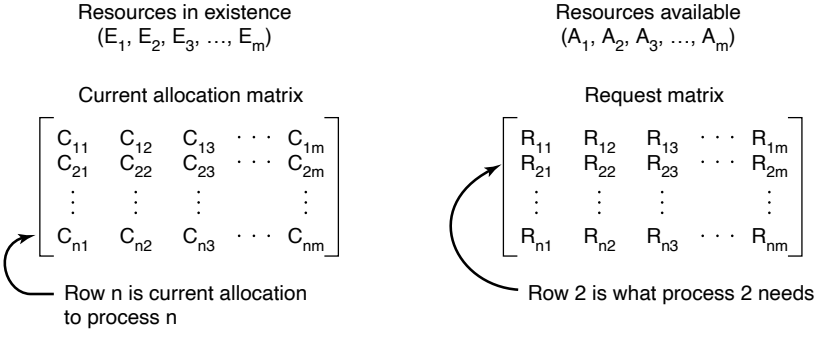

Figure 6-9. The four data structures needed by the deadlock detection algorithm.

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

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j . SP0.6v$$

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 $1 \leq i \leq m$.

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. This algorithm assumes a worst-case scenario: all processes keep all acquired resources until they exit.

The deadlock detection algorithm can now be given as follows:

1. Look for an unmarked process, P_i, for which the ith row of R is less than or equal to A.
2. If such a process is found, add the ith 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 to completion, none of them are deadlocked. If some of them can never finish, 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, see Fig. 6-10. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanners, and cameras. Process 1 has one scanner. Process 2 has two tape drives and a camera. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the R matrix.

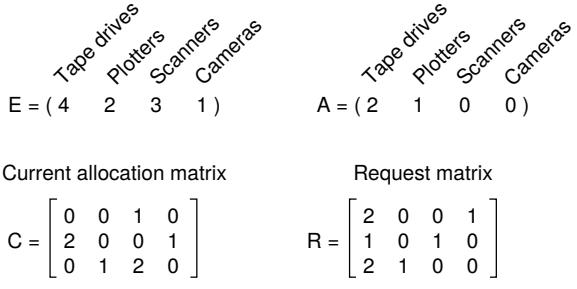

Figure 6-10. An example for the deadlock detection algorithm.

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 camera 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. 6-10. Suppose that process 3 needs a camera as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system will eventually be deadlocked. Even if we give process 3 its two tape drives and one plotter, the system deadlocks when it requests the camera.

Now that we know how to detect deadlocks (at least with static resource requests known in advance), 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.

6.4.3 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, in other words, 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 accumulates.

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 that resource by starting at 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 then 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 partway 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 field of a table in the database, running it once, killing it, and then running it again will add 2 to the field, which is incorrect.

6.5 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. 6-9). 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 make the allocation

only 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.

6.5.1 Resource Trajectories

The main algorithms for deadlock avoidance are based on the concept of safe states. Before describing them, 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. 6-11, 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.

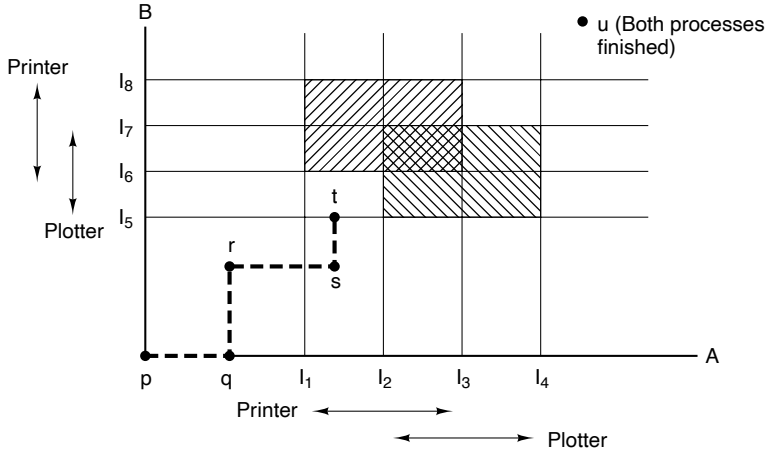

Figure 6-11. Two process resource trajectories.

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 (because processes cannot run backward in time, of course).

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.

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 gets to I_4. Beyond that, any trajectory to u will do.

The important thing to see here is that 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.

6.5.2 Safe and Unsafe States

The deadlock avoidance algorithms that we will study use the information of Fig. 6-9. 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 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. 6-12(a) we have a state in which A has three instances of the resource but may need as many as nine eventually. B currently has two and may need four altogether, later. Similarly, C also has two but may need an additional five. A total of 10 instances of the resource exist, so with seven resources already allocated, three there are still free.

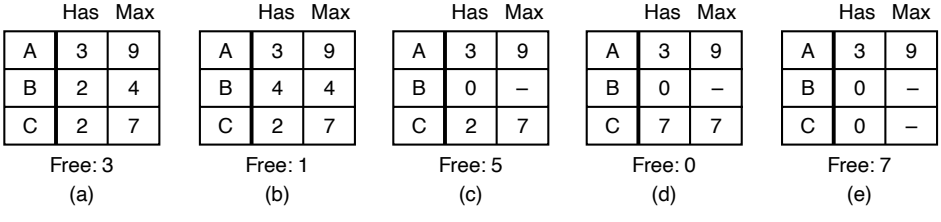

Figure 6-12. Demonstration that the state in (a) is safe.

The state of Fig. 6-12(a) is safe because there exists a sequence of allocations that allows all processes to complete execution. Namely, the scheduler can simply

run *B* exclusively, until it asks for and gets two more instances of the resource, leading to the state of Fig. 6-12(b). When *B* completes, we get the state of Fig. 6-12(c). Then the scheduler can run *C*, leading eventually to Fig. 6-12(d). When *C* completes, we get Fig. 6-12(e). Now *A* can get the six instances of the resource it needs and also complete. Thus, the state of Fig. 6-12(a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. 6-13(a), but this time *A* requests and gets another resource, giving Fig. 6-13(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. 6-13(c).

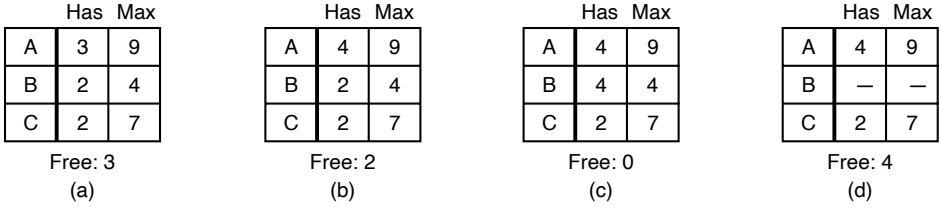

Figure 6-13. Demonstration that the state in (b) is not safe.

Eventually, *B* completes and we get the state of Fig. 6-13(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. 6-13(a) to Fig. 6-13(b) went from a safe to an unsafe state. Running *A* or *C* next starting at Fig. 6-13(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. 6-13(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.

6.5.3 The Banker's Algorithm for a Single Resource

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965); it is known as the **banker's algorithm** and is an extension of the deadlock detection algorithm given in Sec. 6.5. 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. (Years ago, banks did not lend money unless they knew they could be repaid.) What the algorithm does is check to see if granting the request leads to an unsafe state. If so, the request is denied. If granting the request leads to a safe state, it is carried out. In Fig. 6-14(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, printers, and the banker is the operating system.)

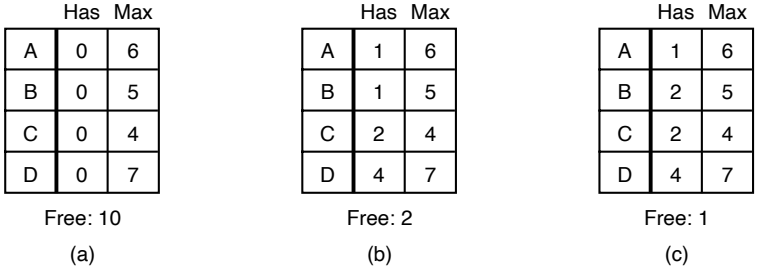

Figure 6-14. Three resource allocation states: (a) safe. (b) safe. (c) unsafe.

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. 6-14(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. 6-14(b). We would have situation Fig. 6-14(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, seeing whether 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.

6.5.4 The Banker's Algorithm for Multiple Resources

The banker's algorithm can be generalized to handle multiple resources. Figure 6-15 shows how it works.

In Fig. 6-15, 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. 6-9. 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.

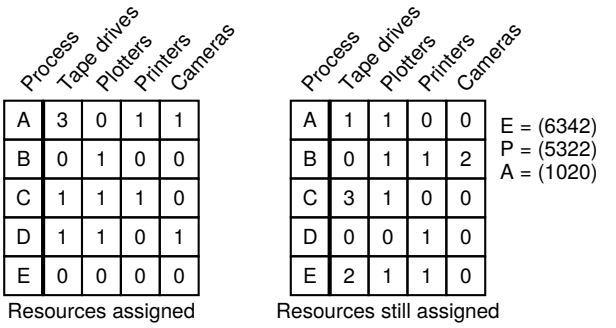

Figure 6-15. The banker's algorithm with multiple resources.

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 cameras. Of these, five tape drives, three plotters, two printers, and two cameras are currently assigned. This fact can be seen by adding up the entries in the four resource columns in the left-hand matrix. The available resource vector is just 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 (assuming processes keep all resources until they exit).
2. Assume the process of the chosen row requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all of 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 no process is left whose resource needs can be met (in which case the system was not safe).

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. 6-15. The current state is safe. Suppose that process *B* now makes a request for the 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, so *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. Some systems, however, use heuristics similar to those of the banker's algorithm to prevent deadlock. For instance, networks may throttle traffic when buffer utilization reaches higher than, say, 70%—estimating that the remaining 30% will be sufficient for current users to complete their service and return their resources.

6.6 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).

6.6.1 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. For data, the simplest method is to make data read only, so that processes can use the data concurrently. 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.

If the daemon is programmed to begin printing even before all the output is spooled, the printer might lie idle if an output process decides 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. However, this decision itself could lead to deadlock. What would happen if two processes each filled up one half of the available spooling space with output and neither was finished producing its full output? In this case, we would have two processes that had each finished part, but not all, of their output, and could not continue. Neither process will ever finish, so we would have a deadlock on the disk.

Nevertheless, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource unless absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

6.6.2 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 will 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 preallocates all resources immediately and does not release them until they are no longer needed by the job (or in the simplest case, 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.

6.6.3 Attacking the No-Preemption Condition

Attacking the third condition (no preemption) is also a possibility. 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. However, some resources can be virtualized to avoid this situation. Spooling printer output to the SSD or disk and allowing only the printer daemon access to the real printer eliminates deadlocks involving the printer, although it creates a potential for deadlock over disk space. With large SSDs/disks though, running out of storage space is unlikely.

However, not all resources can be virtualized like this. For example, records in databases or tables inside the operating system must be locked to be used and therein lies the potential for deadlock.

6.6.4 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. 6-16(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.

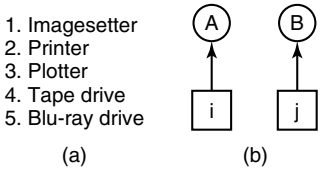

Figure 6-16. (a) Numerically ordered resources. (b) A resource graph.

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. 6-16(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 more than two 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.

Various approaches to deadlock prevention are summarized in Fig. 6-17.

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

Figure 6-17. Summary of approaches to deadlock prevention.

6.7 OTHER ISSUES

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

6.7.1 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, waits a bit, 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 locked record 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 has read or written messages to the network, updated files, or anything else that cannot be safely repeated. 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.

6.7.2 Communication 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. Sometimes the resources are hardware or software objects, such as cameras or database records, but sometimes they are more abstract. Resource deadlock is a problem of **competition synchronization**. Independent processes would complete service if their execution were not interleaved with competing processes. A process locks resources in order to prevent inconsistent resource states caused by interleaved access to resources. Interleaved access to locked resources, however, enables resource deadlock. In Fig. 6-5, we saw a resource deadlock where the resources were semaphores. A semaphore is a bit more abstract than a camera, but in this example, each process successfully acquired a resource (one of the semaphores) and deadlocked trying to acquire another one (the other semaphore). This situation is a classical resource deadlock.

However, as we mentioned at the start of the chapter, while resource deadlocks are the most common kind, they are not the only kind. Another kind of deadlock can occur in communication systems (e.g., networks), in which two or more processes communicate by sending messages. A common arrangement is that process *A* sends a request message to process *B*, and then blocks until *B* sends back a reply message. Suppose that the request message gets lost. *A* is blocked waiting for the reply. *B* is blocked waiting for a request asking it to do something. We have a deadlock.

This, though, is not the classical resource deadlock. *A* does not have possession of some resource *B* wants, and vice versa. In fact, there are no resources at all in sight. But it is a deadlock according to our formal definition since we have a set of (two) processes, each blocked waiting for an event only the other one can cause. This situation is called a **communication deadlock** to contrast it with the more common resource deadlock. Communication deadlock is an anomaly of *cooperation synchronization*. The processes in this type of deadlock could not complete service if executed independently.

Communication deadlocks cannot be prevented by ordering the resources (since there are no resources) or avoided by careful scheduling (since there are no moments when a request could be postponed). Luckily, there is another technique that can usually be employed to break communication deadlocks: timeouts. In most network communication systems, whenever a message is sent to which a reply is expected, a timer is started. If the timer goes off before the reply arrives, the sender of the message assumes that the message has been lost and sends it again (and again and again if needed). In this way, the deadlock is broken. Phrased differently, the timeout serves as a heuristic to detect deadlocks and enables

recovery. This heuristic is applicable to resource deadlocks also and is relied upon by users with buggy device drivers that can deadlock and freeze the system.

Of course, if the original message was not lost but the reply was simply delayed, the intended recipient may get the message two or more times, possibly with undesirable consequences. Think about an electronic banking system in which the message contains instructions to make a payment. Clearly, that should not be repeated (and executed) multiple times just because the network is slow or the timeout too short. Designing the communication rules, called the **protocol**, to get everything right is a complex subject, but one far beyond the scope of this book. Readers interested in network protocols might be interested in another book by one of the authors, *Computer Networks* (Tanenbaum et al., 2020).

Not all deadlocks occurring in communication systems or networks are communication deadlocks. Resource deadlocks can also occur there. Consider, for example, the network of Fig. 6-18. It is a simplified view of the Internet. Very simplified. The Internet consists of two kinds of computers: hosts and routers. A **host** is a user computer, either someone's tablet or PC at home, a PC at a company, or a corporate server. Hosts do work for people. A **router** is a specialized communications computer that moves packets of data from the source to the destination. Each host is connected to one or more routers, either by a digital subscriber line, cable TV connection, LAN, dial-up line, wireless network, optical fiber, or something else.

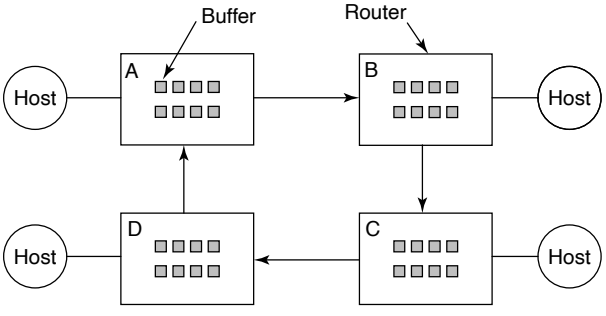

Figure 6-18. A resource deadlock in a network.

When a packet comes into a router from one of its hosts, it is put into a buffer for subsequent transmission to another router and then to another until it gets to the destination. These buffers are resources and there are a finite number of them. In Fig. 6-19, each router has only eight buffers (in practice they have millions, but that does not change the nature of the potential deadlock, just its frequency). Suppose that all the packets at router *A* need to go to *B* and all the packets at *B* need to go to *C* and all the packets at *C* need to go to *D* and all the packets at *D* need to go to *A*. No packet can move because there is no buffer at the other end and we have a classical resource deadlock, albeit in the middle of a communications system.

6.7.3 Livelock

In some situations, a process tries to be polite by giving up the locks it already acquired whenever it notices that it cannot obtain the next lock it needs. Then it waits a millisecond, say, and tries again. In principle, this is good and should help to detect and avoid deadlock. However, if the other process does the same thing at exactly the same time, they will be in the situation of two people trying to pass each other on the street when both of them politely step aside, and yet no progress is possible, because they keep stepping the same way at the same time.

Consider an atomic primitive *try_lock* in which the calling process tests a mutex and either grabs it or returns failure. In other words, it never blocks. Programmers can use it together with *acquire_lock* which also tries to grab the lock, but blocks if the lock is not currently available. Now imagine a pair of processes running in parallel (perhaps on different cores) that use two resources, as shown in Fig. 6-19. Each one needs two resources and uses the *try_lock* primitive to try to acquire the necessary locks. If the attempt fails, the process gives up the lock it holds and tries again. In Fig. 6-19, process *A* runs and acquires resource 1, while process 2 runs and acquires resource 2. Next, they try to acquire the other lock and fail. To be polite, they give up the lock they are currently holding and try again. This procedure repeats until a bored user (or some other entity) puts one of these processes out of its misery. Clearly, no process is blocked and we could even say that things are happening, so this is not a deadlock. Still, no progress is possible, so we do have something equivalent: a **livelock**.

Livelock and deadlock can occur in surprising ways. In some systems, the total number of processes allowed 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 children. 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 livelock. 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? Probably not a good idea.

Most operating systems, including UNIX and Windows, basically just ignore the problem on the assumption that most users would prefer an occasional livelock (or even deadlock) to a rule restricting all users to one process, one open file, and

```
void process_A(void) {
      acquire_lock(&resource_1);
      while (try_lock(&resource_2) == FAIL) {
            release_lock(&resource_1);
            wait_fixed_time();
            acquire_lock(&resource_1);
      }
      use_both_resources( );
      release_lock(&resource_2);
      release_lock(&resource_1);
}

void process_B(void) {
      acquire_lock(&resource_2);
      while (try_lock(&resource_1) == FAIL) {
            release_lock(&resource_2);
            wait_fixed_time();
            acquire_lock(&resource_2);
      }
      use_both_resources( );
      release_lock(&resource_1);
      release_lock(&resource_2);
}
```

Figure 6-19. Polite processes that may cause livelock.

one of everything. If these problems 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. 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.

6.7.4 Starvation

We already saw that a problem closely related to deadlock and livelock 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 algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Who 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-served 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.

It is worth mentioning that some people do not make a distinction between starvation and deadlock because in both cases there is no forward progress. Others feel that they are fundamentally different because a process could easily be programmed to try to do something *n* times and, if all of them failed, try something else. A blocked process does not have that choice.

6.8 RESEARCH ON DEADLOCKS

If ever there was a subject that was investigated mercilessly during the early days of operating systems (1960s and 1970s), it was deadlocks. The reason is that deadlock detection is a nice little graph-theory problem that one mathematically inclined graduate student could get his jaws around and chew on for 4 years. Many algorithms were devised, each one more exotic and less practical than the previous one. Most of that work has died out. Still, a few papers are still being published on deadlocks.

Recent work on deadlocks includes new approaches to diagnose concurrency problems such as deadlocks. The challenge here is to reproduce the schedules, or "thread interleavings," that lead to the deadlock, livelock, and similar conditions. Unfortunately, recording in detail the scheduling decisions in production systems is too expensive. Instead, researchers are looking for ways to record the interleavings at a coarser granularity while guaranteeing reproducibility of the deadlock or livelock problem (Kasikci et al., 2017).

Marino et al. (2013), on the other hand, use concurrency control to make sure that deadlocks cannot occur in the first place. In contrast, Duo et al. (2020) use formal modeling to derive constraints on the scheduling to ensure deadlocks will not happen.

Solutions for deadlock detection are not limited to a single system. For instance, Hu et al. (2017) give a method that prevents deadlocks due to the use of Remote DMA (RDMA) in data centers. RDMA is just like regular DMA as discussed in Chap. 5, except that the DMA transfer is now initiated from a remote machine across the network. In a specific mode, known as priority flow control, the RDMA transfer requires the (exclusive) reservation of RDMA buffers in the intermediate nodes in the network. It does so to make sure there can be no packet drops due to buffers overflowing. However, these buffers are just the sort of limited

resources that we discussed in this chapter. Suppose two hosts both want to perform an RDMA transfer and both have to use buffers on intermediate nodes N_1 and N_2. If one of the hosts reserves the last buffer on N_1 and the other the last buffer on N_2, neither can make progress. By ensuring that the packets from different flows end up in different buffers, Hu et al. (2017) show that deadlocks are not possible.

Another research direction is to try and detect deadlocks. For instance, Pyla and Varadarajan (2012) present a deadlock detection system that associates memory updates with one or more locks guarding the updates and does not make the updates globally visible until all locks that protect the updates are released. All memory updates in a critical region are then performed atomically. By checking at the acquisition of the locks, it is possible to detect deadlocks early and start the recovery procedure . Recovery from a deadlock consists simply of picking one of the locks and discarding all pending memory updates associated with it. The work by Cai and Chan (2012) presents a new dynamic deadlock detection scheme that iteratively prunes lock dependencies that have no incoming or outgoing edges.

Finally, there is a huge amount of theoretical work on distributed deadlock detection. However, we will not consider it 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.

6.9 SUMMARY

Deadlock is a potential problem in any operating system. It occurs when all the members of a set of processes are blocked waiting for an event that only other members of the same set can cause. This situation causes all the processes to wait forever. Commonly the event that the processes are waiting for is the release of some resource held by another member of the set. Another situation in which deadlock is possible is when a set of communicating processes are all waiting for a message and the communication channel is empty and no timeouts are pending.

Resource 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.

Resource 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. Resource deadlock can also be prevented by numbering all the resources and making processes request them in strictly increasing order.

Resource deadlock is not the only kind of deadlock. Communication deadlock is also a potential problem in some systems although it can often be handled by setting appropriate timeouts.

Livelock is similar to deadlock in that it can stop all forward progress, but it is technically different since it involves processes that are not actually blocked. Starvation can be avoided by a first-come, first-served allocation policy.

PROBLEMS

1. Give an example of a deadlock taken from politics.
2. In the dining philosophers problem, let the following protocol be used: An even-numbered philosopher always picks up his left fork before picking up his right fork; an odd-numbered philosopher always picks up his right fork before picking up his left fork. Will this protocol guarantee deadlock-free operation?
3. In the solution to the dining philosophers problem (Fig. 6-5), why is the state variable set to *HUNGRY* in the procedure *take_forks*?
4. Consider the procedure *put_forks* in Fig. 6-5. Suppose that the variable *state*[*i*] was set to *THINKING after* the two calls to *test*, rather than *before*. How would this change affect the solution?
5. Students working at individual PCs in a computer laboratory send their files to be printed by a server that 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?
6. In the preceding question, which resources are preemptable and which are nonpreemptable?
7. The four conditions (mutual exclusion, hold and wait, no preemption and circular wait) are necessary for a resource deadlock to occur. Give an example to show that these conditions are not sufficient for a resource deadlock to occur. When are these conditions sufficient for a resource deadlock to occur?
8. City streets are vulnerable to a circular blocking condition called gridlock, in which intersections are blocked by cars that then block cars behind them that then block the cars that are trying to enter the previous intersection, etc. All intersections around a city block are filled with vehicles that block the oncoming traffic in a circular manner. Gridlock is a resource deadlock and a problem in competition synchronization. New York City's prevention algorithm, called "don't block the box," prohibits cars from entering an intersection unless the space following the intersection is also available. Which prevention algorithm is this? Can you provide any other prevention algorithms for gridlock?
9. Suppose four cars each approach an intersection from four different directions simultaneously. Each corner of the intersection has a stop sign. Assume that traffic regulations require that when two cars approach adjacent stop signs at the same time, the car on the left must yield to the car on the right. Thus, as four cars each drive up to their individual stop signs, each waits (indefinitely) for the car on the left to proceed. Is this anomaly a communication deadlock? Is it a resource deadlock?

10. Is it possible that a resource deadlock involves multiple units of one type and a single unit of another? If so, give an example.

11. Figure 6-6 shows the concept of a resource graph. Do illegal graphs exist, that is, graphs that structurally violate the model we have used of resource usage? If so, give an example of one.

12. Suppose that there is a resource deadlock in a system. Give an example to show that the set of processes deadlocked can include processes that are not in the circular chain in the corresponding resource allocation graph.

13. In order to control traffic, a network router, *A* periodically sends a message to its neighbor, *B*, telling it to increase or decrease the number of packets that it can handle. At some point in time, Router *A* is flooded with traffic and sends *B* a message telling it to cease sending traffic. It does this by specifying that the number of bytes *B* may send (*A*'s window size) is 0. As traffic surges decrease, *A* sends a new message, telling *B* to restart transmission. It does this by increasing the window size from 0 to a positive number. That message is lost. As described, neither side will ever transmit. What type of deadlock is this?

14. 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?

15. Consider the following state of a system with four processes, *P1*, *P2*, *P3*, and *P4*, and five types of resources, *RS1*, *RS2*, *RS3*, *RS4*, and *RS5*:

C =

0	1	1	1	2
0	1	0	1	0
0	0	0	0	1
2	1	0	0	0

R =

1	1	0	2	1
0	1	0	2	1
0	2	0	3	1
0	2	1	1	0

E = (24144)

A = (01021)

Using the deadlock detection algorithm described in Section 6.4.2, show that there is a deadlock in the system. Identify the processes that are deadlocked.

16. Explain how the system can recover from the deadlock in previous problem using

(a) recovery through preemption.
(b) recovery through rollback.
(c) recovery through killing processes.

17. Suppose that in Fig. 6-9 $C_{ij} + R_{ij} > E_j$ for some i. What implications does this have for the system?

18. What is the key difference between the model shown in Fig. 6-8 and the safe and unsafe states described in Sec. 6.5.2. What is the consequence of this difference?

19. All the trajectories in Fig. 6-11 are horizontal or vertical. Can you envision any circumstances in which diagonal trajectories are also possible?

20. Can the resource trajectory scheme of Fig. 6-11 also be used to illustrate the problem of deadlocks with three processes and three resources? How or why not?

21. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Is there a practical way of actually doing this?

22. Consider a system that uses the banker's algorithm to avoid deadlocks. At some time a process P requests a resource R but is denied even though R is currently available. Does it mean that if the system allocated R to P, the system would deadlock?

23. A key limitation of the banker's algorithm is that it requires knowledge of maximum resource needs of all processes. Is it possible to design a deadlock avoidance algorithm that does not require this information? Explain your answer.

24. Take a careful look at Fig. 6-14(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?

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

26. Consider the previous problem again, but now 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?

27. Suppose that process A in Fig. 6-15 requests the last tape drive. Does this action lead to a deadlock?

28. 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?

29. 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?

30. 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?

31. One way to eliminate circular wait is to have rule saying that a process is entitled only to a single resource at any moment. Give an example to show that this restriction is unacceptable in many cases.

32. 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 six possible combinations in which each process can request them. What fraction of the combinations is sure to be deadlock free?

33. 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.

34. 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 a process having locked account x 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.)

35. 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.

36. 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?

37. Main memory units are preempted in swapping and virtual memory systems. The processor is preempted in time-sharing environments. Do you think that these preemption methods were developed to handle resource deadlock or for other purposes? How high is their overhead?

38. Explain the differences between deadlock, livelock, and starvation.

39. Assume two processes are issuing a seek command to reposition the mechanism to access the disk and enable a read command. Each process is interrupted before executing its read, and discovers that the other has moved the disk arm. Each then reissues the seek command, but is again interrupted by the other. This sequence continually repeats. Is this a resource deadlock or a livelock? What methods would you recommend to handle the anomaly?

40. Local Area Networks utilize a media access method called CSMA/CD, in which stations sharing a bus can sense the medium and detect transmissions as well as collisions. In the Ethernet protocol, stations requesting the shared channel do not transmit frames if they sense the medium is busy. When such transmission has terminated, waiting stations each transmit their frames. Two frames that are transmitted at the same time will collide. If stations immediately and repeatedly retransmit after collision detection, they will continue to collide indefinitely.

(a) Is this a resource deadlock or a livelock?
(b) Can you suggest a solution to this anomaly?
(c) Can starvation occur with this scenario?

41. A program contains an error in the order of cooperation and competition mechanisms, resulting in a consumer process locking a mutex (mutual exclusion semaphore) before it blocks on an empty buffer. The producer process blocks on the mutex before it can place a value in the empty buffer and awaken the consumer. Thus, both processes are blocked forever, the producer waiting for the mutex to be unlocked and the consumer waiting for a signal from the producer. Is this a resource deadlock or a communication deadlock? Suggest methods for its control.

42. Cinderella and the Prince are getting divorced. To divide up 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 that 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.

43. 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.

44. 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.

45. 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.

46. Write a program to implement the deadlock detection algorithm with multiple resources of each type. Your program should read from a file the following inputs: the number of processes, the number of resource types, the number of resources of each type in existence (vector E), the current allocation matrix C (first row, followed by the second row, and so on), the request matrix R (first row, followed by the second row, and so on). The output of your program should indicate whether there is a deadlock in the system. In case there is, the program should print out the identities of all processes that are deadlocked.

47. Write a program that detects if there is a deadlock in the system by using a resource allocation graph. Your program should read from a file the following inputs: the number of processes and the number of resources. For each process it should read four numbers: the number of resources it is currently holding, the IDs of resources it is holding, the number of resources it is currently requesting, and the IDs of resources it is requesting. The output of program should indicate if there is a deadlock in the system. In case there is, the program should print out the identities of all processes that are deadlocked.

48. In certain countries, when two people meet they bow to each other. The protocol is that one of them bows first and stays down until the other one bows. If they bow at the same time, they will not know who should stand up straight first, thus creating a deadlock. Write a program that does not deadlock.

49. In Chap. 2, we have studied monitors. Solve the dining philosophers problem using monitors instead of semaphores.