Process Synchronization:

Process synchronization involves using tools that control access to shared data to avoid race conditions. These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.

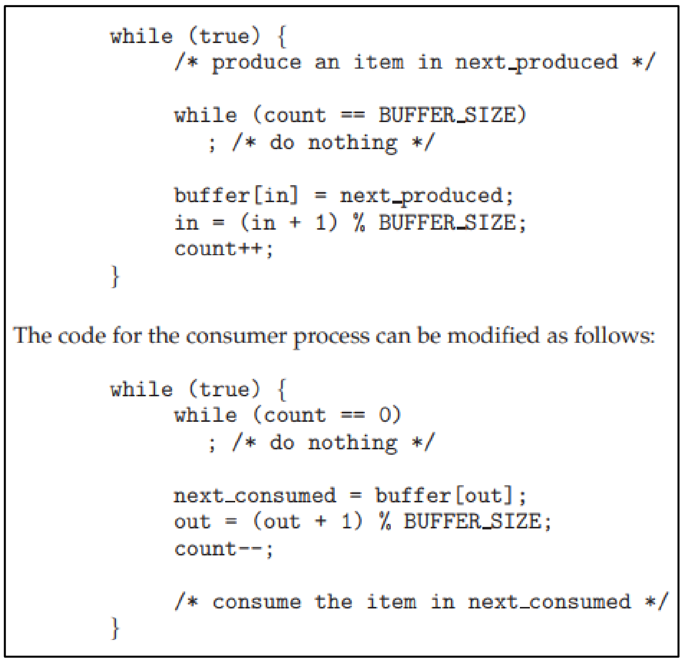
Synchronization Tools:

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing.

Background:

We illustrated this model with the producer–consumer problem, which is a representative paradigm of many operating system functions. Specifically, in Section 3.5, we described how a bounded buffer could be used to enable processes to share memory.

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at BUFFER SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable, count, initialized to 0. count is incremented every time we add a new item to the buffer and is decremented every time, we remove one item from the buffer. The code for the producer process can be modified as follows:



```
while (true) {
      /* produce an item in next_produced */

      while (count == BUFFER_SIZE)
         ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
      while (count == 0)
         ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      count--;

      /* consume the item in next_consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements “count++” and “count--”. Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is count == 5, which is generated correctly if the producer and consumer execute separately.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the prominence of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, we devote a major portion of this chapter to *process synchronization* and *coordination* among cooperating processes.

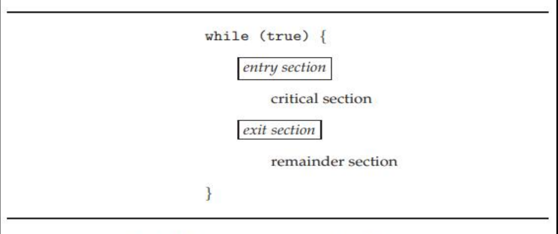
The Critical-Section Problem:

We begin our consideration of process synchronization by discussing the so called *critical-section* problem. Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to synchronize their activity to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is the *entry section*. The critical section may be followed by an *exit section*. The remaining code is the *remainder section.* The general structure of a typical process is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.



```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Figure 6.1 General structure of a typical process.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (kernel code) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Another example is illustrated in Figure 6.2. In this situation, two processes, P0 and P1, are creating child processes using the fork() system call. Recall from Section 3.3.1 that fork() returns the process identifier of the newly created process to the parent process. In this example, there is a race condition on the variable kernel variable next available pid which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions. The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

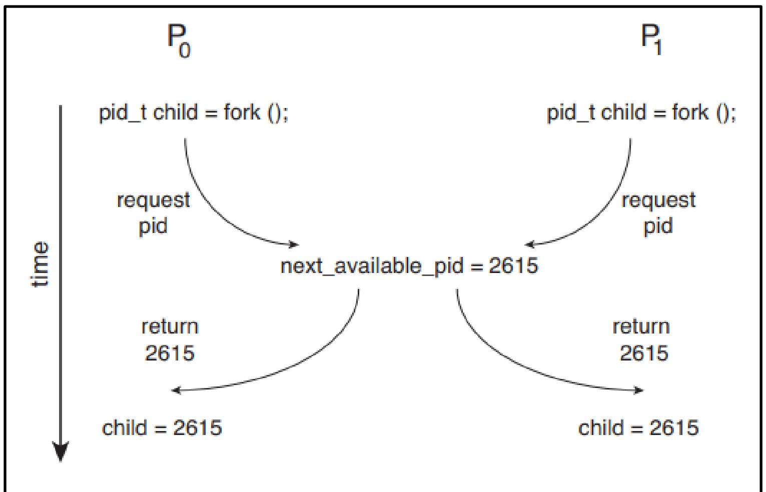


Figure 6.2 Race condition when assigning a pid.

Two general approaches are used to handle critical sections in operating systems: preemptive kernels and nonpreemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

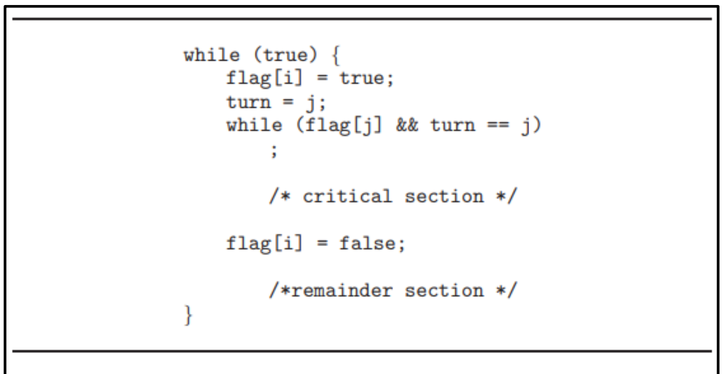
Peterson’s Solution:

Next, we illustrate a classic software-based solution to the critical-section problem known as Peterson’s solution. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson’s solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting. Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1 − i.

Peterson’s solution requires the two processes to share two data items:

int turn;

Boolean flag[2];



```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Figure 6.3 The structure of process P_i in Peterson’s solution.

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

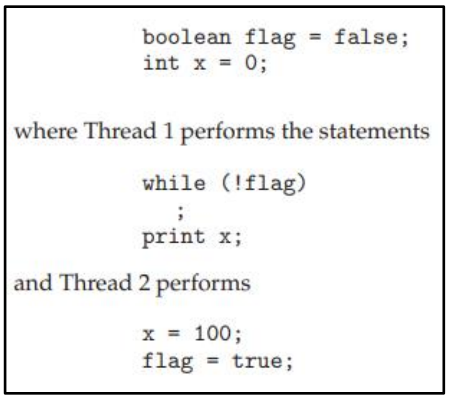
1. Mutual exclusion is preserved.

2. The progress requirement is satisfied.

3. The bounded-waiting requirement is met.

As mentioned at the beginning of this section, Peterson’s solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. For a singlethreaded application, this reordering is immaterial as far as program correctness is concerned, as the final values are consistent with what is expected. (This is like balancing a checkbook— the actual order in which credit and debit operations are performed is unimportant, because the final balance will still be the same.) But for a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

As an example, consider the following data that are shared between two threads:



```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
   ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

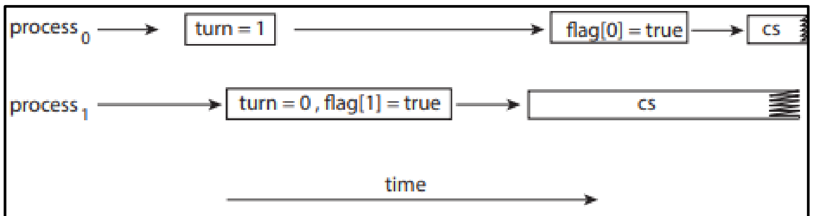


Figure 6.4 The effects of instruction reordering in Peterson's solution.

Hardware Support for Synchronization:

software-based solutions are not guaranteed to work on modern computer architectures. In this section, we present three hardware instructions that provide support for solving the critical-section problem. These primitive operations can be used directly as synchronization tools, or they can be used to form the foundation of more abstract synchronization mechanisms.

1.Memory Barriers:

we saw that a system may reorder instructions, a policy that can lead to unreliable data states. How a computer architecture determines what memory guarantees it will provide to an application program is known as its memory model.

In general, a memory model falls into one of two categories:

1. Strongly ordered, where a memory modification on one processor is immediately visible to all other processors.

2. Weakly ordered, where modifications to memory on one processor may not be immediately visible to other processors.

Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor. To address this issue, computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as memory barriers or memory fences.

Let's return to our most recent example, in which reordering of instructions could have resulted in the wrong output, and use a memory barrier to ensure that we obtain the expected output.

If we add a memory barrier operation to Thread 1

```
while (!flag)
   memory_barrier();
print x;
```

we guarantee that the value of flag is loaded before the value of x.

Similarly, if we place a memory barrier between the assignments performed by Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

we ensure that the assignment to x occurs before the assignment to flag.

2.Hardware Instructions:

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically— that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the test and set() and compare and swap() instructions.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic test_and_set() instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 6.6 Mutual-exclusion implementation with test_and_set().

The test and set() instruction can be defined as shown in Figure 6.5. The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order. If the machine supports the test and set() instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi is shown in Figure 6.6.

The CAS instruction operates on three operands and is defined in Figure 6.7. The operand value is set to new value only if the expression (*value == expected) is true. Regardless, CAS always returns the original value of the variable value. The important characteristic of this instruction is that it is executed atomically. Thus, if two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7 The definition of the atomic `compare_and_swap()` instruction.

Mutual exclusion using CAS can be provided as follows: A global variable (lock) is declared and is initialized to 0. The first process that invokes compare and swap() will set lock to 1. It will then enter its critical section, because the original value of lock was equal to the expected value of 0. Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0. When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section. The structure of process Pi is shown in Figure 6.8.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Figure 6.8 Mutual exclusion with the `compare_and_swap()` instruction.

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement. In Figure 6.9, we present another algorithm using the compare and swap() instruction that satisfies all the critical-section requirements.
The common data structures are

Boolean waiting[n];
int lock;

MAKING COMPARE-AND-SWAP ATOMIC

On Intel x86 architectures, the assembly language statement cmpxchg is used to implement the compare_and_swap() instruction. To enforce atomic execution, the lock prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

      /* remainder section */
}
```

Figure 6.9 Bounded-waiting mutual exclusion with compare_and_swap().

3.Atomic Variables:

Typically, the compare and swap() instruction is not used directly to provide mutual exclusion. Rather, it is used as a basic building block for constructing other tools that solve the critical-section problem. One such tool is an atomic variable, which provides atomic operations on basic data types such as integers and Booleans. We know from Section 6.1 that incrementing or decrementing an integer value may produce a race condition. Atomic variables can be used in to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.

Most systems that support atomic variables provide special atomic data types as well as functions for accessing and manipulating atomic variables. These functions are often implemented using compare and swap() operations.

As an example, the following increments the atomic integer sequence:

increment(&sequence);

where the increment() function is implemented using the CAS instruction:

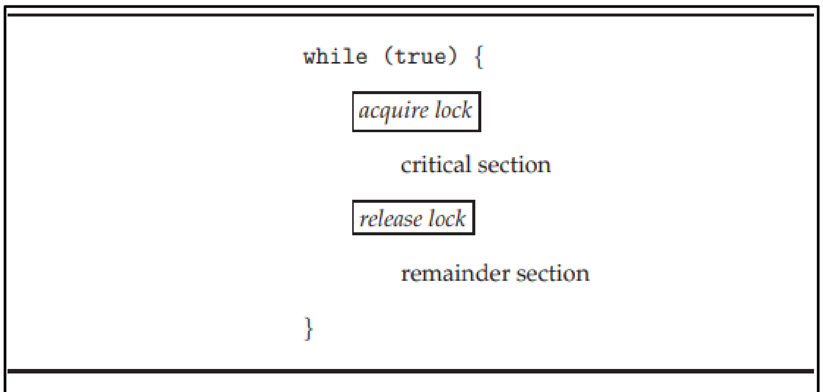
```
void increment(atomic_int *v)
{
   int temp;

   do {
      temp = *v;
   }
   while (temp != compare_and_swap(v, temp, temp+1));
}
```

Mutex Locks:

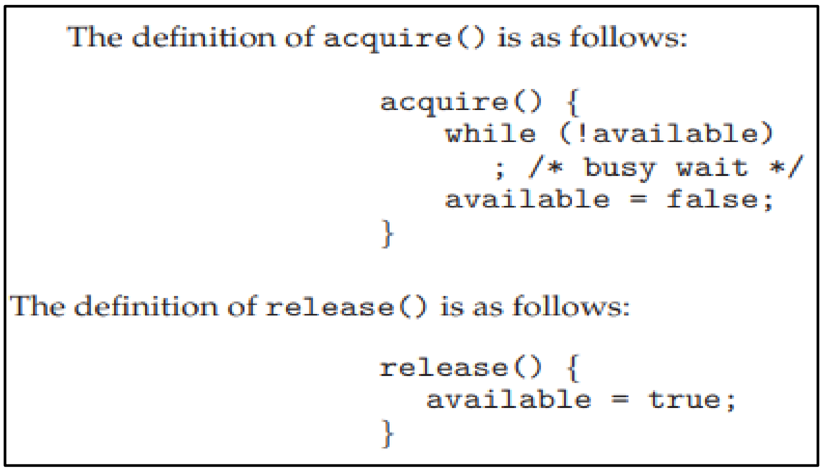
The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated as well as generally inaccessible to application programmers. Instead, operating-system designers build higher-level software tools to solve the critical-section problem. The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.) We use the mutex lock to protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire () function acquires the lock, and the release () function releases the lock, as illustrated in Figure 6.10.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire () succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.



```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

Figure 6.10 Solution to the critical-section problem using mutex locks.



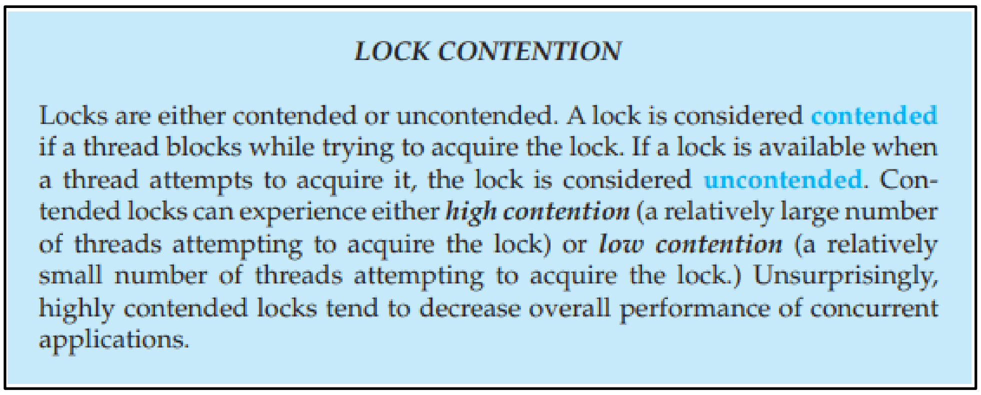
The definition of acquire() is as follows:

```
acquire() {
   while (!available)
     ; /* busy wait */
   available = false;
}
```

The definition of release() is as follows:

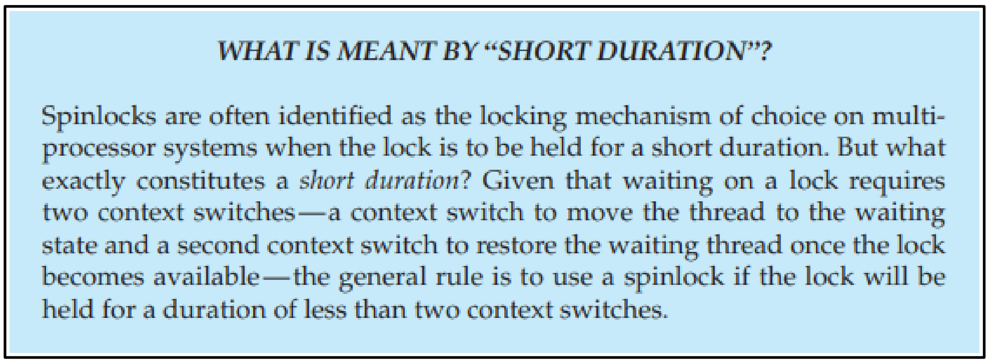
```
release() {
  available = true;
}
```

Calls to either acquire() or release() must be performed atomically. Thus, mutex locks can be implemented using the CAS operation described in Section 6.4, and we leave the description of this technique as an exercise.



LOCK CONTENTION

Locks are either contended or uncontended. A lock is considered **contended** if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Contended locks can experience either ***high contention*** (a relatively large number of threads attempting to acquire the lock) or ***low contention*** (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.



WHAT IS MEANT BY "SHORT DURATION"?

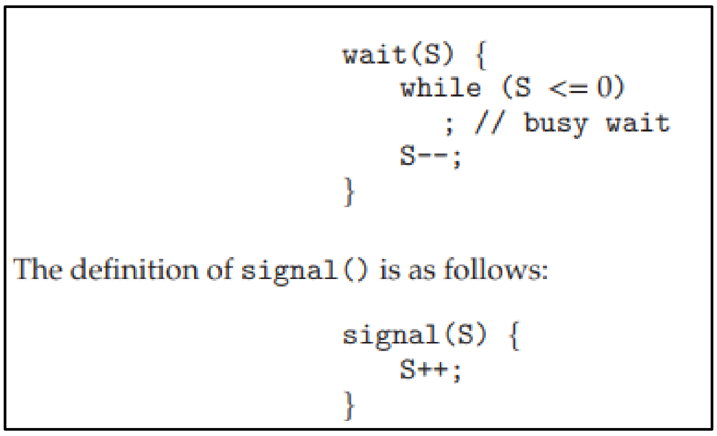
Spinlocks are often identified as the locking mechanism of choice on multiprocessor systems when the lock is to be held for a short duration. But what exactly constitutes a *short duration*? Given that waiting on a lock requires two context switches—a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available—the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

The main disadvantage of the implementation given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().

The type of mutex lock we have been describing is also called a spinlock because the process "spins" while waiting for the lock to become available. (We see the same issue with the code examples illustrating the compare and swap() instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.

Semaphores:

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the wait() operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "to increment"). The definition of wait() is as follows:



```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically.

Semaphore Usage:

Operating systems often distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0. In process P1, we insert the statements

```
S1;
signal(synch);
```

In process P2, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation:

Recall that the implementation of mutex locks discussed in Section 6.5 suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same problem. To overcome this problem, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is suspended, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

Now, the wait() semaphore operation can be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

and the signal() semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

The sleep() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a suspended process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

As mentioned, it is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a criticalsection problem, and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

Monitors:

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place, and these sequences do not always occur.

We have seen an example of such errors in the use of a count in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then, the count value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that mutex locks and semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when either mutex locks or semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a binary semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we list several difficulties that may result. Note that these difficulties will arise even if a single process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

• Suppose that a program interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    ...
   critical section
    ...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

• Suppose that a program replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
    ...
   critical section
    ...
wait(mutex);
```

In this case, the process will permanently block on the second call to wait(), as the semaphore is now unavailable.

• Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated, or the process will permanently block.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem. One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs. In this section, we describe one fundamental high-level synchronization construct— the monitor type.

Monitor Usage:

An abstract data type—or ADT—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A monitor type is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown in Figure 6.11.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

          .
          .
          .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 6.11 Pseudocode syntax of a monitor.

A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). The operation

x.wait();

means that the process invoking this operation is suspended until another process invokes

x.signal();

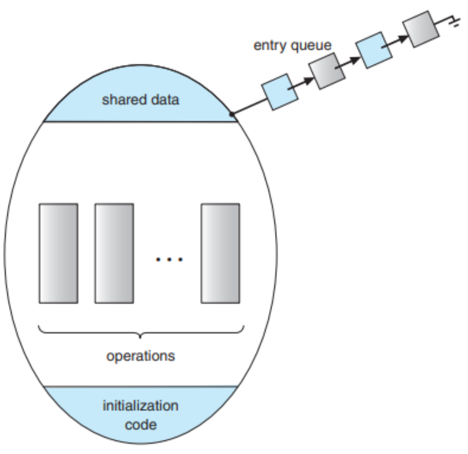


Figure 6.12 Schematic view of a monitor.

Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. Signal and wait. P either waits until Q leaves the monitor or waits for another condition.
2. Signal and continue. Q either waits until P leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since P was already executing in the monitor, the signal-andcontinue method seems more reasonable. On the other, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold. A compromise between these two choices exists as well: when thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.

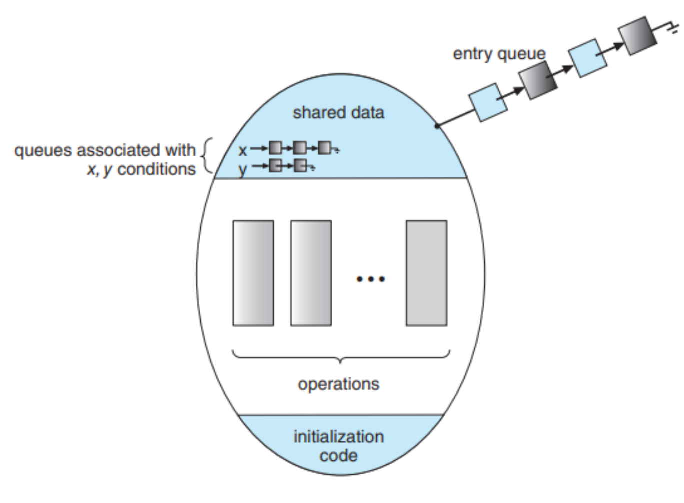


Figure 6.13 Monitor with condition variables.

Implementing a Monitor Using Semaphores:

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a binary semaphore mutex (initialized to 1) is provided to ensure mutual exclusion. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

We will use the signal-and-wait scheme in our implementation. Since a signaling process must wait until the resumed process either leaves or waits, an additional binary semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
     ...
   body of F
     ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

Mutual exclusion within a monitor is ensured. We can now describe how condition variables are implemented as well. For each condition x, we introduce a binary semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen (see the bibliographical notes at the end of the chapter). In some cases, however, the generality of the implementation is unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.27.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

Figure 6.14 A monitor to allocate a single resource.

Resuming Processes within a Monitor:

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. In these circumstances, the conditional wait construct can be used.

This construct has the form

```
x.wait(c);
```

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a priority number, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the Resource Allocator monitor shown in Figure 6.14, which controls the allocation of a single resource among competing processes.

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
   ...
    access the resource;
   ...
R.release();
```

where R is an instance of type ResourceAllocator.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

Classic Problems of Synchronization:

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

The Bounded-Buffer Problem:

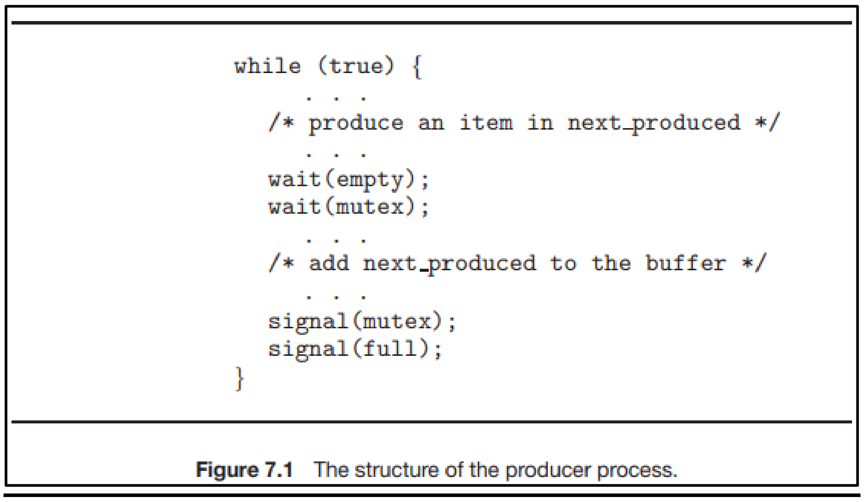
The bounded-buffer problem was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

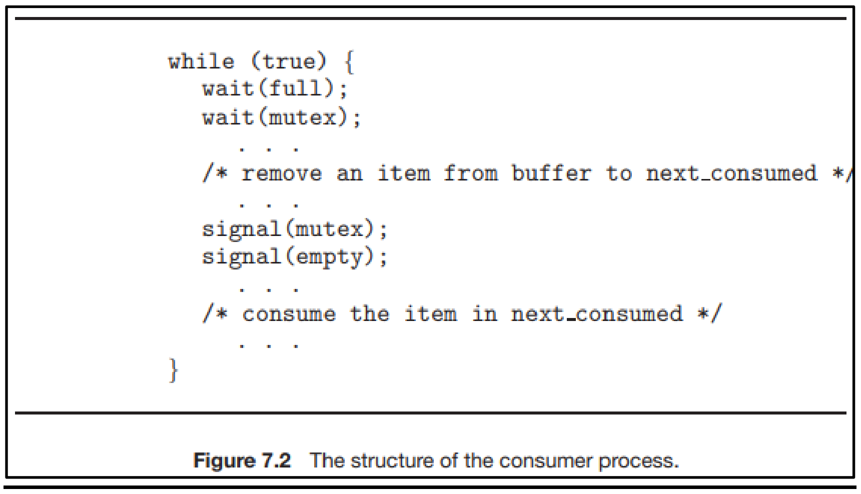
We assume that the pool consists of n buffers, each capable of holding one item. The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 7.1, and the code for the consumer process is shown in Figure 7.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.



```
while (true) {
     . . .
   /* produce an item in next_produced */
     . . .
   wait(empty);
   wait(mutex);
     . . .
   /* add next_produced to the buffer */
     . . .
   signal(mutex);
   signal(full);
}
```

Figure 7.1 The structure of the producer process.



```
while (true) {
   wait(full);
   wait(mutex);
     . . .
   /* remove an item from buffer to next_consumed */
     . . .
   signal(mutex);
   signal(empty);
     . . .
   /* consume the item in next_consumed */
     . . .
}
```

Figure 7.2 The structure of the consumer process.

The Readers –Writers Problem:

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue. To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers–writers problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The binary semaphores mutex and rw mutex are initialized to 1; read count is a counting semaphore initialized to 0. The semaphore rw mutex is common to both reader and writer processes.

The code for a writer process is shown in Figure 7.3; the code for a reader process is shown in Figure 7.4. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on rw mutex, and n − 1 readers are queued on mutex. Also observe that, when a writer executes signal(rw mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions have been generalized to provide reader–writer locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access. When a process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

• In applications where it is easy to identify which processes only read shared data and which processes only write shared data.

• In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

The Dining-Philosophers Problem:

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbour's). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The dining-philosophers problem is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

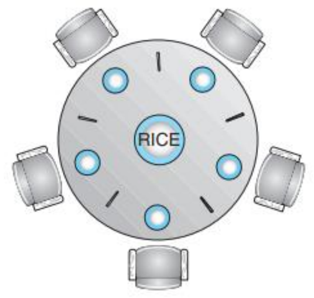
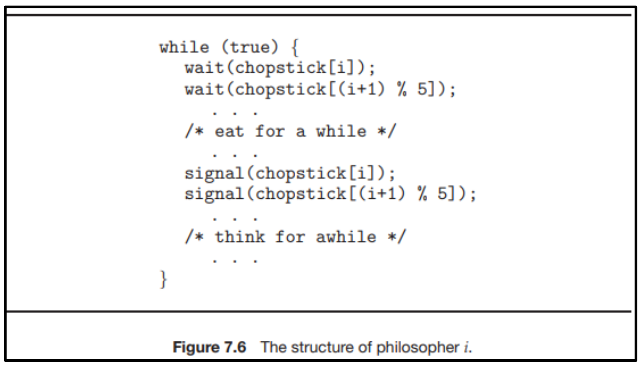


Figure 7.5 The situation of the dining philosophers.



```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Figure 7.6 The structure of philosopher *i*.

Semaphore Solution:

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure 7.6.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are the following:

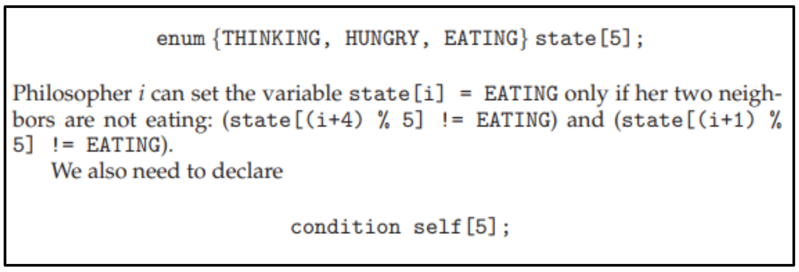
• Allow at most four philosophers to be sitting simultaneously at the table.

• Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

• Use an asymmetric solution— that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an evennumbered philosopher picks up her right chopstick and then her left chopstick.

Monitor Solution:

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:



```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable `state[i] = EATING` only if her two neighbors are not eating: (`state[(i+4) % 5] != EATING`) and (`state[(i+1) % 5] != EATING`).

We also need to declare

```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining philosopher's problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers, whose definition is shown in Figure 7.7. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
                 ...
                 eat
                 ...
DiningPhilosophers.putdown(i);
```

Deadlocks:

In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resource. if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state because the resources it has requested are held by other waiting threads. This situation is called a deadlock.

System Model:

A system consists of a finite number of resources to be distributed among a number of competing threads. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as network interfaces and DVD drives) are examples of resource types. If a system has four CPUs, then the resource type CPU has four instances. Similarly, the resource type network may have two instances. If a thread requests an instance of a resource type, the allocation of any instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

The various synchronization tools discussed in Chapter 6, such as mutex locks and semaphores, are also system resources; and on contemporary computer systems, they are the most common sources of deadlock. However, definition is not a problem here. A lock is typically associated with a specific data structure— that is, one lock may be used to protect access to a queue, another to protect access to a linked list, and so forth. For that reason, each instance of a lock is typically assigned its own resource class.

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

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

1. Request. The thread requests the resource. If the request cannot be granted immediately (for example, if a mutex lock is currently held by another thread), then the requesting thread must wait until it can acquire the resource.

2. Use. The thread can operate on the resource (for example, if the resource is a mutex lock, the thread can access its critical section).

3. Release. The thread releases the resource.

To illustrate a deadlocked state, we refer back to the dining-philosophers problem from Section 7.1.3. In this situation, resources are represented by chopsticks. If all the philosophers get hungry at the same time, and each philosopher grabs the chopstick on her left, there are no longer any available chopsticks. Each philosopher is then blocked waiting for her right chopstick to become available.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 6 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released.

Deadlock Characterization:

In the previous section we illustrated how deadlock could occur in multithreaded programming using mutex locks. We now look more closely at conditions that characterize deadlock.

Necessary Conditions

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

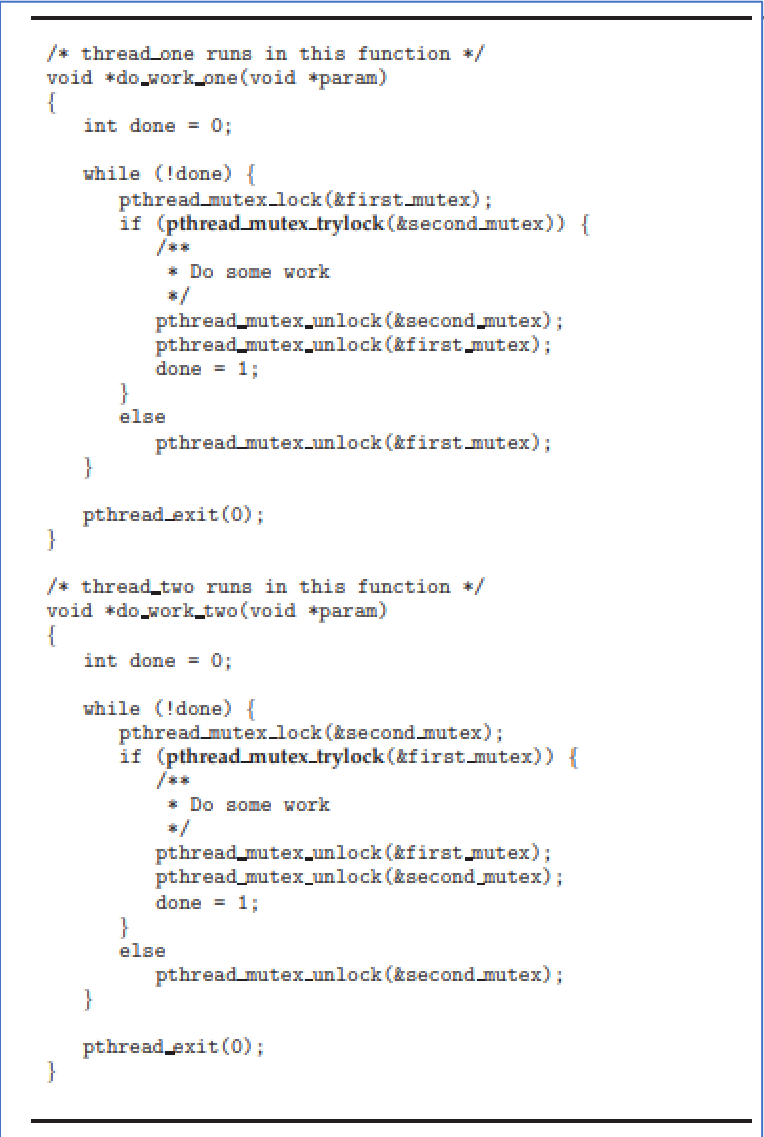
1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.

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

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

4. Circular wait. A set {T0, T1, ..., Tn} of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 8.5, however, that it is useful to consider each condition separately.



```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&first_mutex);
      if (pthread_mutex_trylock(&second_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&second_mutex);
         pthread_mutex_unlock(&first_mutex);
         done = 1;
      }
      else
         pthread_mutex_unlock(&first_mutex);
   }

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&second_mutex);
      if (pthread_mutex_trylock(&first_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&first_mutex);
         pthread_mutex_unlock(&second_mutex);
         done = 1;
      }
      else
         pthread_mutex_unlock(&second_mutex);
   }

   pthread_exit(0);
}
```

Figure 8.2 Livelock example.

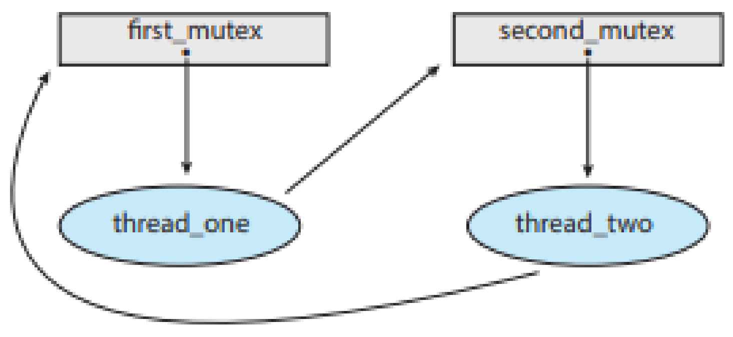


Figure 8.3 Resource-allocation graph for program in Figure 8.1.

Methods for Handling Deadlocks:

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

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

The first solution is the one used by most operating systems, including Linux and Windows. It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the second solution. Some systems—such as databases—adopt the third solution, allowing deadlocks to occur and then managing the recovery.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions (Section 8.3.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 8.5.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether the thread should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

Deadlock Prevention:

As we noted in Section 8.3.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

Mutual Exclusion:

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

Hold and Wait:

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources. One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is, of course, impractical for most applications due to the dynamic nature of requesting resources.

An alternative protocol allows a thread to request resources only when it has none. A thread may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

No Preemption:

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a thread requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. If the resources are neither available nor held by a waiting thread, the requesting thread must wait. While it is waiting, some of its resources may be preempted, but only if another thread requests them. A thread can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions. It cannot generally be applied to such resources as mutex locks and semaphores, precisely the type of resources where deadlock occurs most commonly.

Circular Wait:

The three options presented thus far for deadlock prevention are generally impractical in most situations. However, the fourth and final condition for deadlocks — the circular-wait condition — presents an opportunity for a practical solution by invalidating one of the necessary conditions. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.

To illustrate, we let R = {R1, R2, ..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R → N, where N is the set of natural numbers. We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. For example, the lock ordering in the Pthread program shown in Figure 8.1 could be

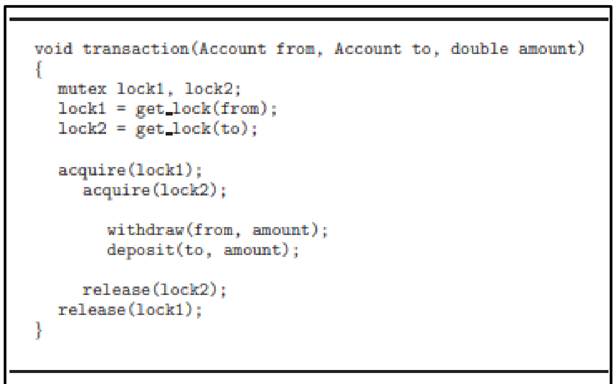
F(first mutex)=1

F(second mutex)=5

It is also important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a function that transfers funds between two accounts. To prevent a race condition, each account has an associated mutex lock that is obtained from a get lock() function such as that shown in Figure 8.7. Deadlock is possible if two threads simultaneously invoke the transaction() function, transposing different accounts. That is, one thread might invoke

transaction(checking account, savings account, 25.0) and another might

invoke transaction(savings account, checking account, 50.0)



```
void transaction(Account from, Account to, double amount)
{
  mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);

  acquire(lock1);
    acquire(lock2);

      withdraw(from, amount);
      deposit(to, amount);

    release(lock2);
  release(lock1);
}
```

Figure 8.7 Deadlock example with lock ordering.

Deadlock Avoidance:

Deadlock-prevention algorithms, as discussed in Section 8.5, prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with resources R1 and R2, the system might need to know that thread P will request first R1 and then R2 before releasing both resources, whereas thread Q will request R2 and then R1. With this knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each thread declare the maximum number of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the threads. In the following sections, we explore two deadlock-avoidance algorithms.

Safe State:

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

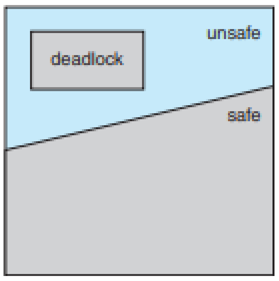


Figure 8.8 Safe, unsafe, and deadlocked state spaces.

Resource-Allocation-Graph Algorithm:

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined in Section 8.3.2 for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge. A claim edge Ti → Rj indicates that thread Ti may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When thread Ti requests resource Rj , the claim edge Ti → Rj is converted to a request edge. Similarly, when a resource Rj is released by Ti , the assignment edge Rj → Ti is reconverted to a claim edge Ti → Rj .

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

Now suppose that thread Ti requests resource Rj . The request can be granted only if converting the request edge Ti → Rj to an assignment edge Rj → Ti does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of threads in the system.

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

To illustrate this algorithm, we consider the resource-allocation graph of Figure 8.9. Suppose that T2 requests R2. Although R2 is currently free, we cannot allocate it to T2, since this action will create a cycle in the graph (Figure 8.10). A cycle, as mentioned, indicates that the system is in an unsafe state. If T1 requests R2, and T2 requests R1, then a deadlock will occur.

Banker's Algorithm:

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

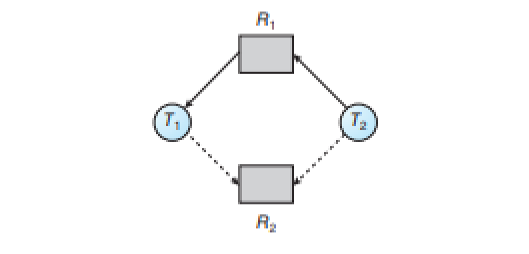


Figure 8.9 Resource-allocation graph for deadlock avoidance.

deadlock-avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the thread must wait until some other thread releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of threads in the system and m is the number of resource types:

• **Available.** A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.

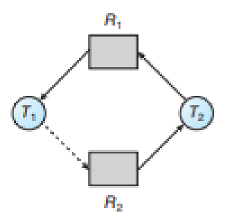


Figure 8.10 An unsafe state in a resource-allocation graph.

• **Max**. An n × m matrix defines the maximum demand of each thread. If Max[i][j] equals k, then thread Ti may request at most k instances of resource type Rj .

• **Allocation.** An n × m matrix defines the number of resources of each type currently allocated to each thread. If Allocation[i][j] equals k, then thread Ti is currently allocated k instances of resource type Rj .

• **Need.** An n × m matrix indicates the remaining resource need of each thread. If Need[i][j] equals k, then thread Ti may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

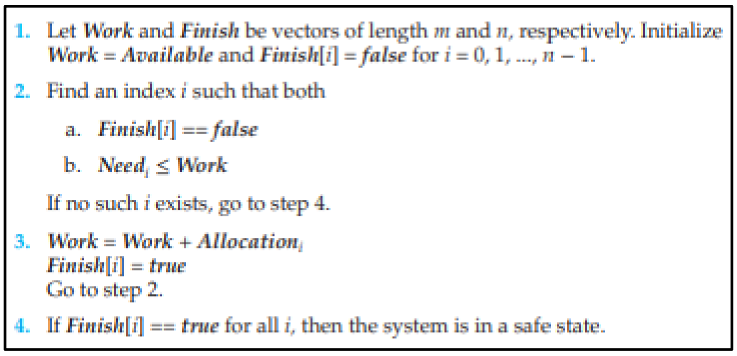
These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that X ≤ Y if and only if X[i] ≤ Y[i] for all i = 1, 2, ..., n. For example, if X = (1,7,3,2) and Y = (0,3,2,1), then Y ≤ X. In addition, Y < X if Y ≤ X and Y ≠ X.

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

Safety Algorithm:

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

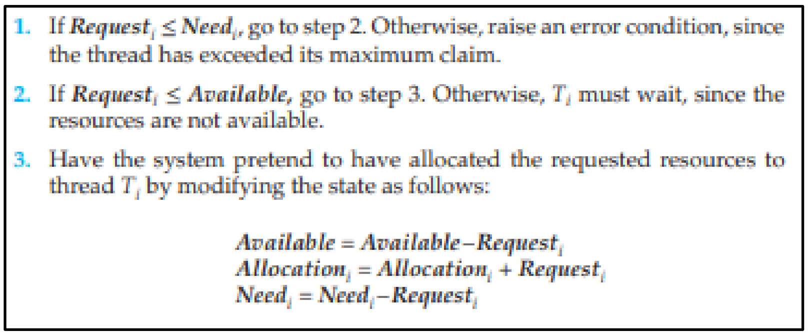


```
1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize
   $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.
2. Find an index $i$ such that both
     a. $Finish[i] == false$
     b. $Need_i \leq Work$
   If no such $i$ exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == true$ for all $i$, then the system is in a safe state.
```

This algorithm may require an order of m × n2 operations to determine whether a state is safe.

Resource-Request Algorithm:

Next, we describe the algorithm for determining whether requests can be safely granted. Let Requesti be the request vector for thread Ti . If Requesti [j] == k, then thread Ti wants k instances of resource type Rj . When a request for resources is made by thread Ti , the following actions are taken:



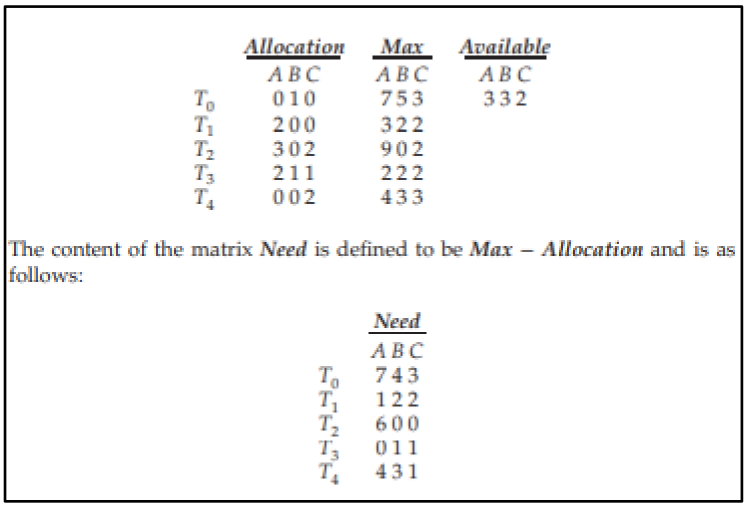
1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

$$\begin{aligned} Available &= Available - Request_i \\ Allocation_i &= Allocation_i + Request_i \\ Need_i &= Need_i - Request_i \end{aligned}$$

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

An Illustrative Example:

To illustrate the use of the banker's algorithm, consider a system with five threads T0 through T4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that the following snapshot represents the current state of the system:

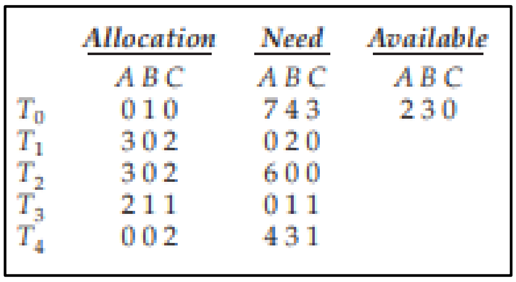


	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max* − *Allocation* and is as follows:

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence satisfies the safety criteria. Suppose now that thread T1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2). To decide whether this request can be immediately granted, we first check that Request1 ≤ Available— that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:



	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence satisfies the safety requirement. Hence, we can immediately grant the request of thread T1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by T4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by T0 cannot be granted, even though the resources are available since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

Deadlock Detection:

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

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

Next, we discuss these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

Single Instance of Each Resource Type

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

More precisely, an edge from Ti to Tj in a wait-for graph implies that thread Ti is waiting for thread Tj to release a resource that Ti needs. An edge Ti → Tj exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges Ti → Rq and Rq → Tj for some resource Rq. In Figure 8.11, we present a resource-allocation graph and the corresponding wait-for graph.

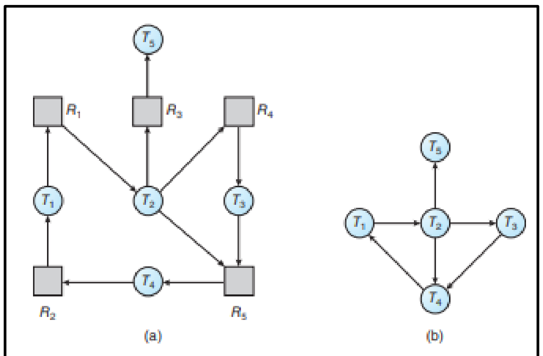


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Several Instances of a Resource Type:

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 8.6.3):

- **Available**. A vector of length m indicates the number of available resources of each type
- **Allocation.** An n × m matrix defines the number of resources of each type currently allocated to each thread.

• **Request.** An n × m matrix indicates the current request of each thread.

If Request[i][j] equals k, then thread Ti is requesting k more instances of resource type Rj . The ≤ relation between two vectors is defined as in Section 8.6.3. To sim plify notation, we again treat the rows in the matrices Allocation and Request as vectors; we refer to them as Allocationi and Requesti . The detection algo rithm described here simply investigates every possible allocation sequence for the threads that remain to be competed. Compare this algorithm with the banker's algorithm of Section 8.6.3

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available*. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then *Finish*[i] = *false*. Otherwise, *Finish*[i] = *true*.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[i] = *true*
   Go to step 2.
4. If *Finish*[i] == *false* for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if *Finish*[i] == *false*, then thread T_i is deadlocked.

This algorithm requires an order of m × n2 operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five threads T0 through T4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. The following snapshot represents the current state of the system:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence results in Finish[i] == true for all i.

Suppose now that thread T2 makes one additional request for an instance of type C. The Request matrix is modified as follows

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by thread T0, the number of available resources is not sufficient to fulfill the requests of the other threads. Thus, a deadlock exists, consisting of threads T1, T2, T3, and T4. 8.7.3

Detection-Algorithm

Usage When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?

2. How many threads will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked threads will be idle until the deadlock can be broken. In addition, the number of threads involved in the deadlock cycle may grow.

Deadlocks occur only when some thread makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting threads.

Recovery from Deadlock:

When a detection algorithm determines that a deadlock exists, several alter natives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more threads to break the circular wait. The other is to preempt some resources from one or more of the deadlocked threads.

Process and Thread Termination:

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

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

• Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed and how much longer the process will compute before completing its designated task

3. How many and what types of resources the process has used (for exam ple, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete

5. How many processes will need to be terminated

Resource Preemption:

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

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