Operating Systems and Middleware: Supporting Controlled Interaction
by Max Hailperin

The free re-release was prepared from final page proofs and should be completely identical to the commercially published version. In particular, all the errata listed on the web site still apply. (The author intends to release subsequent versions that incorporate the corrections as well as updates and improvements. Subsequent versions may also be in a more easily modifiable form to encourage participation by other contributors. Please email suggestions to max@gustavus.edu.)

Credits from the commercially published version:

Senior Product Manager: Alyssa Pratt
Managing Editor: Mary Franz
Development Editor: Jill Batistick
Senior Marketing Manager: Karen Seitz
Associate Product Manager: Jennifer Smith
Editorial Assistant: Allison Murphy
Senior Manufacturing Coordinator: Justin Palmeiro
Cover Designer: Deborah VanRooyen
Compositor: Interactive Composition Corporation

CHAPTER

4

Synchronization and Deadlocks

4.1 Introduction

In Chapters 2 and 3, you have seen how an operating system can support concurrent threads of execution. Now the time has come to consider how the system supports controlled interaction between those threads. Because threads running at the same time on the same computer can inherently interact by reading and writing a common set of memory locations, the hard part is providing control. In particular, this chapter will examine control over the relative timing of execution steps that take place in differing threads.

Recall that the scheduler is granted considerable authority to temporarily preempt the execution of a thread and dispatch another thread. The scheduler may do so in response to unpredictable external events, such as how long an I/O request takes to complete. Therefore, the computational steps taken by two (or more) threads will be interleaved in a quite unpredictable manner, unless the programmer has taken explicit measures to control the order of events. Those control measures are known as *synchronization*. Synchronization causes one thread to wait for another.

In Section 4.2, I will provide a more detailed case for why synchronization is needed by describing the problems that can occur when interacting threads are not properly synchronized. The uncontrolled interactions are called races. By examining

some typical races, I will illustrate the need for one particular form of synchronization, mutual exclusion. Mutual exclusion ensures that only one thread at a time can operate on a shared data structure or other resource. Section 4.3 presents two closely related ways mutual exclusion can be obtained. They are known as mutexes and monitors.

After covering mutual exclusion, I will turn to other, more general synchronization challenges and to mechanisms that can address those challenges. To take one example, you may want to ensure that some memory locations are read after they have been filled with useful values, rather than before. I devote Section 4.4 to enumerating several of the most common synchronization patterns other than mutual exclusion. Afterward, I devote Sections 4.5 and 4.6 to two popular mechanisms used to handle these situations. One, condition variables, is an important extension to monitors; the combination of monitors with condition variables allows many situations to be cleanly handled. The other, semaphores, is an old favorite because it provides a single, simple mechanism that in principle suffices for all synchronization problems. However, semaphores can be hard to understand and use correctly.

Synchronization solves the problem of races, but it creates a new problem of its own: deadlock. Recall that synchronization typically involves making threads wait; for example, in mutual exclusion, a thread may need to wait its turn in order to enforce the rule of one at a time. Deadlock results when a cycle of waiting threads forms; for example, thread A waits for thread B, which happens to be waiting for thread A, as shown in Figure 4.1. Because this pathology results from synchronization, I will address it and three of the most practical cures in Section 4.7, after completing the study of synchronization itself.

Synchronization interacts with scheduling (the topic of Chapter 3) in some interesting ways. In particular, unless special precautions are taken, synchronization mechanisms can subvert priority scheduling, allowing a low-priority thread to run while a high-priority thread waits. Therefore, in Section 4.8, I will briefly consider the interactions between synchronization and scheduling, as well as what can be done to tame them.

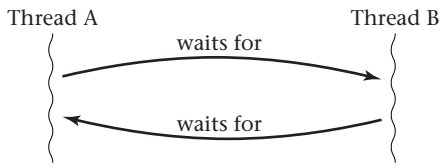

Figure 4.1 Deadlock results when threads wait for one another in a complete cycle. In this simple example, thread A is waiting for thread B, which is waiting for thread A.

Finally, I conclude the chapter in Section 4.9 by looking at security issues related to synchronization. In particular, I show how subtle synchronization bugs, which may nearly never cause a malfunction unless provoked, can be exploited by an attacker in order to circumvent the system's normal security policies. After this concluding section, I provide exercises, programming and exploration projects, and notes.

Despite the wide range of synchronization-related topics I cover in this chapter, there are two I leave for later chapters. Atomic transactions are a particularly sophisticated and important synchronization pattern, commonly encountered in middleware; therefore, I devote Chapter 5 entirely to them. Also, explicitly passing a message between threads (for example, via a network) provides synchronization as well as communication, because the message cannot be received until after it has been transmitted. Despite this synchronization role, I chose to address various forms of message passing in Chapters 9 and 10, the chapters related to communication.

4.2 Races and the Need for Mutual Exclusion

When two or more threads operate on a shared data structure, some very strange malfunctions can occur if the timing of the threads turns out precisely so that they interfere with one another. For example, consider the following code that might appear in a `sellTicket` procedure (for an event without assigned seats):

```
if(seatsRemaining > 0){
  dispenseTicket();
  seatsRemaining = seatsRemaining - 1;
} else
  displaySorrySoldOut();
```

On the surface, this code looks like it should never sell more tickets than seats are available. However, what happens if multiple threads (perhaps controlling different points of sale) are executing the same code? Most of the time, all will be well. Even if two people try to buy tickets at what humans perceive as the same moment, on the time scale of the computer, probably one will happen first and the other second, as shown in Figure 4.2. In that case, all is well. However, once in a blue moon, the timing may be exactly wrong, and the following scenario results, as shown in Figure 4.3.

1. Thread A checks `seatsRemaining > 0`. Because `seatsRemaining` is 1, the test succeeds. Thread A will take the first branch of the `if`.
2. Thread B checks `seatsRemaining > 0`. Because `seatsRemaining` is 1, the test succeeds. Thread B will take the first branch of the `if`.
3. Thread A dispenses a ticket and decreases `seatsRemaining` to 0.
4. Thread B dispenses a ticket and decreases `seatsRemaining` to −1.
5. One customer winds up sitting on the lap of another.

Thread A	Thread B

`if(seatsRemaining > 0)`
`dispenseTicket();`
`seatsRemaining=seatsRemaining-1;`
	`if(seatsRemaining > 0)...else`
	`displaySorrySoldOut();`

Figure 4.2 Even if two humans think they are trying to buy the last ticket at the same time, chances are good that one's thread (thread A in this example) will run before the other's. Thread B will then correctly discover that no seats remain.

Thread A	Thread B
`if(seatsRemaining > 0)`	`if(seatsRemaining > 0)`
`dispenseTicket();`	`dispenseTicket();`
`seatsRemaining=seatsRemaining-1;`	`seatsRemaining=seatsRemaining-1;`

Figure 4.3 If threads A and B are interleaved, both can act as though there were a ticket left to sell, even though only one really exists for the two of them.

Of course, there are plenty of other equally unlikely scenarios that result in misbehavior. In Exercise 4.1, you can come up with a scenario where, starting with `seatsRemaining` being 2, two threads each dispense a ticket, but `seatsRemaining` is left as 1 rather than 0.

These scenarios are examples of *races*. In a race, two threads use the same data structure, without any mechanism to ensure only one thread uses the data structure at a time. If either thread precedes the other, all is well. However, if the two are interleaved, the program malfunctions. Generally, the malfunction can be expressed as some invariant property being violated. In the ticket-sales example, the invariant is that the value of `seatsRemaining` should be nonnegative and when added to the number of tickets dispensed should equal the total number of seats. (This invariant assumes that `seatsRemaining` was initialized to the total number of seats.)

When an invariant involves more than one variable, a race can result even if one of the threads only reads the variables, without modifying them. For example, suppose there are two variables, one recording how many tickets have been sold and the other

recording the amount of cash in the money drawer. There should be an invariant relation between these: the number of tickets sold times the price per ticket, plus the amount of starting cash, should equal the cash on hand. Suppose one thread is in the midst of selling a ticket. It has updated one of the variables, but not yet the other. If at exactly that moment another thread chooses to run an audit function, which inspects the values of the two variables, it will find them in an inconsistent state.

That inconsistency may not sound so terrible, but what if a similar inconsistency occurred in a medical setting, and one variable recorded the drug to administer, while the other recorded the dose? Can you see how dangerous an inconsistency could be? Something very much like that happened in a radiation therapy machine, the Therac-25, with occasionally lethal consequences. (Worse, some patients suffered terrible but not immediately lethal injuries and lingered for some time in excruciating, intractable pain.)

From the ticket-sales example, you can see that having two threads carrying out operations on the same data structure is harmless, as long as there never are two operations under way at the same time. In other words, the interleaving of the threads' execution needs to be at the granularity of complete operations, such as selling a ticket or auditing the cash drawer. The two threads can take turns selling tickets in any arbitrary fashion, not just in strict alternation. However, each sale should be completed without interruption.

The reason why any interleaving of complete operations is safe is because each is designed to both rely on the invariant and preserve it. Provided that you initially construct the data structure in a state where the invariant holds, any sequence whatsoever of invariant-preserving operations will leave the invariant intact.

What is needed, then, is a synchronization mechanism that allows one thread to lock the data structure before it begins work, thereby excluding all other threads from operating on that structure. When the thread that locked the structure is done, it unlocks, allowing another thread to take its turn. Because any thread in the midst of one of the operations temporarily excludes all the others, this arrangement is called *mutual exclusion*. Mutual exclusion establishes the granularity at which threads may be interleaved by the scheduler.

4.3 Mutexes and Monitors

As you saw in Section 4.2, threads that share data structures need to have a mechanism for obtaining exclusive access to those structures. A programmer can arrange for this exclusive access by creating a special lock object associated with each shared data structure. The lock can be held by only one thread at a time. If the threads operate on

(or even examine) the data structure only when holding the corresponding lock, this discipline will prevent races.

To support this form of race prevention, operating systems and middleware generally provide mutual exclusion locks. Because the name *mutual exclusion lock* is rather ungainly, something shorter is generally used. Some programmers simply talk of *locks*, but that can lead to confusion because other synchronization mechanisms are also called locks. (For example, I introduce readers/writers locks in Section 4.4.2.) Therefore, the name *mutex* has become popular as a shortened form of *mutual exclusion lock*. In particular, the POSIX standard refers to mutexes. Therefore, I will use that name in this book as well.

Section 4.3.1 presents the POSIX application programming interface (API) for mutexes. Section 4.3.2 discusses an alternative, more structured interface to mutexes, known as monitors. Finally, Section 4.3.3 shows what lies behind both of those interfaces by explaining the mechanisms typically used to implement mutexes.

4.3.1 The Mutex Application Programming Interface

A mutex can be in either of two states: locked (that is, held by some thread), or unlocked (that is, not held by any thread). Any implementation of mutexes must have some way to create a mutex and initialize its state. Conventionally, mutexes are initialized to the unlocked state. As a minimum, there must be two other operations: one to lock a mutex, and one to unlock it.

The lock and unlock operations are much less symmetrical than they sound. The unlock operation can be applied only when the mutex is locked; this operation does its job and returns, without making the calling thread wait. The lock operation, on the other hand, can be invoked even when the lock is already locked. For this reason, the calling thread may need to wait, as shown in Figure 4.4. When a thread invokes the lock operation on a mutex, and that mutex is already in the locked state, the thread is

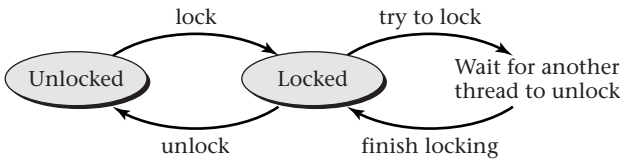

Figure 4.4 Locking an unlocked mutex and unlocking a locked one change the mutex's state. However, a thread can also try to lock an already-locked mutex. In this case, the thread waits and acquires the mutex lock when another thread unlocks it.

made to wait until another thread has unlocked the mutex. At that point, the thread that wanted to lock the mutex can resume execution, find the mutex unlocked, lock it, and proceed.

If more than one thread is trying to lock the same mutex, only one of them will switch the mutex from unlocked to locked; that thread will be allowed to proceed. The others will wait until the mutex is again unlocked. This behavior of the lock operation provides mutual exclusion. For a thread to proceed past the point where it invokes the lock operation, it must be the single thread that succeeds in switching the mutex from unlocked to locked. Until the thread unlocks the mutex, one can say it *holds* the mutex (that is, has exclusive rights) and can safely operate on the associated data structure in a race-free fashion.

This freedom from races exists regardless of which one of the waiting threads is chosen as the one to lock the mutex. However, the question of which thread goes first may matter for other reasons; I return to it in Section 4.8.2.

Besides the basic operations to initialize a mutex, lock it, and unlock it, there may be other, less essential, operations as well. For example, there may be one to test whether a mutex is immediately lockable without waiting, and then to lock it if it is so. For systems that rely on manual reclamation of memory, there may also be an operation to destroy a mutex when it will no longer be used.

Individual operating systems and middleware systems provide mutex APIs that fit the general pattern I described, with varying details. In order to see one concrete example of an API, I will present the mutex operations included in the POSIX standard. Because this is a standard, many different operating systems provide this API, as well as perhaps other system-specific APIs.

In the POSIX API, you can declare **`my_mutex`** to be a mutex and initialize it with the default attributes as follows:

```
pthread_mutex_t my_mutex;
pthread_mutex_init(&my_mutex, 0);
```

A thread that wants to lock the mutex, operate on the associated data structure, and then unlock the mutex would do the following (perhaps with some error-checking added):

```
pthread_mutex_lock(&my_mutex);
// operate on the protected data structure
pthread_mutex_unlock(&my_mutex);
```

As an example, Figure 4.5 shows the key procedures from the ticket-sales example, written in C using the POSIX API. When all threads are done using the mutex (leaving

```
void sellTicket(){
  pthread_mutex_lock(&my_mutex);
  if(seatsRemaining > 0){
    dispenseTicket();
    seatsRemaining = seatsRemaining - 1;
    cashOnHand = cashOnHand + PRICE;
  } else
    displaySorrySoldOut();
  pthread_mutex_unlock(&my_mutex);
}

void audit(){
  pthread_mutex_lock(&my_mutex);
  int revenue = (TOTAL_SEATS - seatsRemaining) * PRICE;
  if(cashOnHand != revenue + STARTING_CASH){
    printf("Cash fails to match.\n");
    exit(1);
  }
  pthread_mutex_unlock(&my_mutex);
}
```

Figure 4.5 Each of these procedures begins by locking **my_mutex** and ends by unlocking it. Therefore, they will never race, even if called from concurrent threads. Additional code not shown here (perhaps in the main procedure) would first initialize **my_mutex**.

it in the unlocked state), the programmer is expected to destroy it, so that any underlying memory can be reclaimed. This is done by executing the following procedure call:

```
pthread_mutex_destroy(&my_mutex);
```

POSIX also provides a couple of variants on **pthread_mutex_lock** that are useful under more limited circumstances. One, **pthread_mutex_trylock**, differs in that it will never wait to acquire a mutex. Instead, it returns an error code if unable to immediately acquire the lock. The other, **pthread_mutex_timedlock**, allows the programmer to specify a maximum amount of time to wait. If the mutex cannot be acquired within that time, **pthread_mutex_timedlock** returns an error code.

Beyond their wide availability, another reason why POSIX mutexes are worth studying is that the programmer is allowed to choose among several variants, which provide different answers to two questions about exceptional circumstances. Other

mutex APIs might include one specific answer to these questions, rather than exposing the full range of possibilities. The questions at issue are as follows:

- What happens if a thread tries to unlock a mutex that is unlocked, or that was locked by a different thread?
- What happens if a thread tries to lock a mutex that it already holds? (Note that if the thread were to wait for itself to unlock the mutex, this situation would constitute the simplest possible case of a deadlock. The cycle of waiting threads would consist of a single thread, waiting for itself.)

The POSIX standard allows the programmer to select from four different types of mutexes, each of which answers these two questions in a different way:

`PTHREAD_MUTEX_DEFAULT` If a thread tries to lock a mutex it already holds or unlock one it doesn't hold, all bets are off as to what will happen. The programmer has a responsibility never to make either of these attempts. Different POSIX-compliant systems may behave differently.

`PTHREAD_MUTEX_ERROR_CHECK` If a thread tries to lock a mutex that it already holds, or unlock a mutex that it doesn't hold, the operation returns an error code.

`PTHREAD_MUTEX_NORMAL` If a thread tries to lock a mutex that it already holds, it goes into a deadlock situation, waiting for itself to unlock the mutex, just as it would wait for any other thread. If a thread tries to unlock a mutex that it doesn't hold, all bets are off; each POSIX-compliant system is free to respond however it likes.

`PTHREAD_MUTEX_RECURSIVE` If a thread tries to unlock a mutex that it doesn't hold, the operation returns an error code. If a thread tries to lock a mutex that it already holds, the system simply increments a count of how many times the thread has locked the mutex and allows the thread to proceed. When the thread invokes the unlock operation, the counter is decremented, and only when it reaches 0 is the mutex really unlocked.

If you want to provoke a debate among experts on concurrent programming, ask their opinion of *recursive locking*, that is, of the mutex behavior specified by the POSIX option **`PTHREAD_MUTEX_RECURSIVE`**. On the one hand, recursive locking gets rid of one especially silly class of deadlocks, in which a thread waits for a mutex it already holds. On the other hand, a programmer with recursive locking available may not follow as disciplined a development approach. In particular, the programmer may not keep track of exactly which locks are held at each point in the program's execution.

4.3.2 Monitors: A More Structured Interface to Mutexes

Object-oriented programming involves packaging together data structures with the procedures that operate on them. In this context, mutexes can be used in a very rigidly structured way:

- All state within an object should be kept private, accessible only to code associated with that object.
- Every object (that might be shared between threads) should contain a mutex as an additional field, beyond those fields containing the object's state.
- Every method of an object (except private ones used internally) should start by locking that object's mutex and end by unlocking the mutex immediately before returning.

If these three rules are followed, then it will be impossible for two threads to race on the state of an object, because all access to the object's state will be protected by the object's mutex.

Programmers can follow these rules manually, or the programming language can provide automatic support for the rules. Automation ensures that the rules are consistently followed. It also means the source program will not be cluttered with mutex clichés, and hence will be more readable.

An object that automatically follows the mutex rules is called a *monitor*. Monitors are found in some programming languages, such as Concurrent Pascal, that have been used in research settings without becoming commercially popular. In these languages, using monitors can be as simple as using the keyword **`monitor`** at the beginning of a declaration for a class of objects. All public methods will then automatically lock and unlock an automatically supplied mutex. (Monitor languages also support another synchronization feature, condition variables, which I discuss in Section 4.5.)

Although true monitors have not become popular, the Java programming language provides a close approximation. To achieve monitor-style synchronization, the Java programmer needs to exercise some self-discipline, but less than with raw mutexes. More importantly, the resulting Java program is essentially as uncluttered as a true monitor program would be; all that is added is one keyword, **`synchronized`**, at the declaration of each nonprivate method.

Each Java object automatically has a mutex associated with it, of the recursively lockable kind. The programmer can choose to lock any object's mutex for the duration of any block of code by using a **`synchronized`** statement:

```
synchronized(someObject){
  // the code to do while holding someObject's mutex
}
```

Note that in this case, the code need not be operating on the state of `someObject`; nor does this code need to be in a method associated with that object. In other words, the `synchronized` statement is essentially as flexible as using raw mutexes, with the one key advantage that locking and unlocking are automatically paired. This advantage is important, because it eliminates one big class of programming errors. Programmers often forget to unlock mutexes under exceptional circumstances. For example, a procedure may lock a mutex at the beginning and unlock it at the end. However, in between may come an `if` statement that can return from the procedure with the mutex still locked.

Although the `synchronized` statement is flexible, typical Java programs don't use it much. Instead, programmers add the keyword `synchronized` to the declaration of public methods. For example, a `TicketVendor` class might follow the outline in Figure 4.6. Marking a method `synchronized` is equivalent to wrapping the entire body of that method in a `synchronized` statement:

```
synchronized(this){
  // the body
}
```

In other words, a synchronized method on an object will be executed while holding that object's mutex. For example, the `sellTicket` method is synchronized, so if two different threads invoke it, one will be served while the other waits its turn, because the `sellTicket` method is implicitly locking a mutex upon entry and unlocking it upon return, just like was done explicitly in the POSIX version of Figure 4.5. Similarly, a thread executing the `audit` method will need to wait until no ticket sale is in progress, because this method is also marked synchronized, and so acquires the same mutex.

In order to program in a monitor style in Java, you need to be disciplined in your use of the `private` and `public` keywords (including making all state `private`), and you need to mark all the public methods as `synchronized`.

4.3.3 Underlying Mechanisms for Mutexes

In this subsection, I will show how mutexes typically operate behind the scenes. I start with a version that functions correctly, but is inefficient, and then show how to build a more efficient version on top of it, and then a yet more efficient version on top of that. Keep in mind that I will not throw away my first two versions: they play a critical role in the final version. For simplicity, all three versions will be of the `PTHREAD_MUTEX_NORMAL` kind; that is, they won't do anything special if a thread tries to lock a mutex it already holds, nor if it tries to unlock one it doesn't hold. In Exercise 4.3, you can figure out the changes needed for `PTHREAD_MUTEX_RECURSIVE`.

```
public class TicketVendor {
  private int seatsRemaining, cashOnHand;
  private static final int PRICE = 1000;

  public synchronized void sellTicket(){
    if(seatsRemaining > 0){
      dispenseTicket();
      seatsRemaining = seatsRemaining - 1;
      cashOnHand = cashOnHand + PRICE;
    } else
      displaySorrySoldOut();
  }

  public synchronized void audit(){
    // check seatsRemaining, cashOnHand
  }

  private void dispenseTicket(){
    // ...
  }

  private void displaySorrySoldOut(){
    // ...
  }

  public TicketVendor(){
    // ...
  }
}
```

Figure 4.6 Outline of a monitor-style class in Java.

The three versions of mutex are called the basic spinlock, cache-conscious spinlock, and queuing mutex, in increasing order of sophistication. The meaning of these names will become apparent as I explain the functioning of each kind of mutex. I will start with the basic spinlock.

All modern processor architectures have at least one instruction that can be used to both change the contents of a memory location and obtain information about the previous contents of the location. Crucially, these instructions are executed *atomically*, that is, as an indivisible unit that cannot be broken up by the arrival of an interrupt nor interleaved with the execution of an instruction on another processor. The details of these instructions vary; for concreteness, I will use the *exchange* operation,

which atomically swaps the contents of a register with the contents of a memory location.

Suppose I represent a basic spinlock as a memory location that contains 1 if the mutex is unlocked and 0 if the mutex is locked. The unlock operation can be trivial: to unlock a mutex, just store 1 into it. The lock operation is a bit trickier and uses the atomic exchange operation; I can express it in pseudocode, as shown in Figure 4.7. The key idea here is to keep looping until the thread succeeds in changing the mutex from 1 to 0. Until then (so long as some other thread holds the lock), the thread keeps swapping one 0 with another 0, which does no harm. This process is illustrated in Figure 4.8.

To understand the motivation behind the cache-conscious spinlock, you need to know a little about cache coherence protocols in multiprocessor systems. Copies of

```
to lock mutex:
  let temp = 0
  repeat
    atomically exchange temp and mutex
  until temp = 1
```

Figure 4.7 The basic spinlock version of a mutex is a memory location storing 1 for unlocked and 0 for locked. Locking the mutex consists of repeatedly exchanging a register containing 0 with the memory location until the location is changed from 1 to 0.

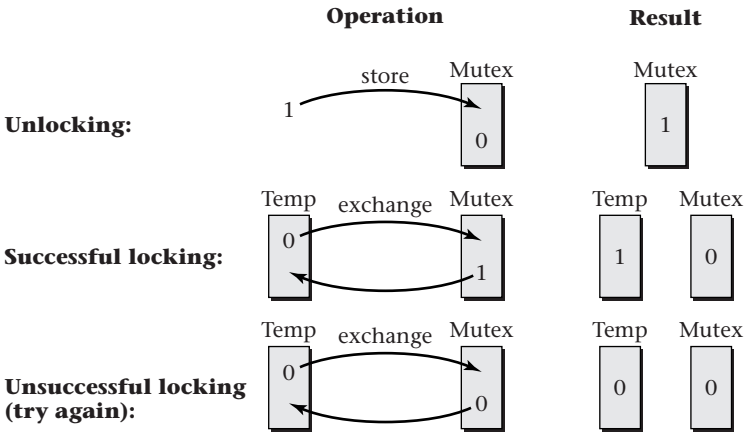

Figure 4.8 Unlocking a basic spinlock consists of storing a 1 into it. Locking it consists of storing a 0 into it using an atomic exchange instruction. The exchange instruction allows the locking thread to verify that the value in memory really was changed from 1 to 0. If not, the thread repeats the attempt.

a given block of memory can reside in several different processors' caches, as long as the processors only read from the memory locations. As soon as one processor wants to write into the cache block, however, some communication between the caches is necessary so that other processors don't read out-of-date values. Most typically, the cache where the writing occurs invalidates all the other caches' copies so that it has exclusive ownership. If one of the other processors now wants to write, the block needs to be flushed out of the first cache and loaded exclusively into the second. If the two processors keep alternately writing into the same block, there will be continual traffic on the memory interconnect as the cache block is transferred back and forth between the two caches.

This is exactly what will happen with the basic spinlock version of mutex locking if two threads (on two processors) are both waiting for the same lock. The atomic exchange instructions on the two processors will both be writing into the cache block containing the spinlock. Contention for a mutex may not happen often. When it does, however, the performance will be sufficiently terrible to motivate an improvement. Cache-conscious spinlocks will use the same simple approach as basic spinlocks when there is no contention, but will get rid of the cache coherence traffic while waiting for a contended mutex.

In order to allow multiple processors to wait for a lock without generating traffic outside their individual caches, they must be waiting while using only reads of the mutex. When they see the mutex become unlocked, they then need to try grabbing it with an atomic exchange. This approach leads to the pseudocode shown in Figure 4.9. Notice that in the common case where the mutex can be acquired immediately, this version acts just like the original. Only if the attempt to acquire the mutex fails is anything done differently. Even then, the mutex will eventually be acquired the same way as before.

```
to lock mutex:
  let temp = 0
  repeat
    atomically exchange temp and mutex
    if temp = 0 then
      while mutex = 0
        do nothing
  until temp = 1
```

Figure 4.9 Cache-conscious spinlocks are represented the same way as basic spinlocks, using a single memory location. However, the lock operation now uses ordinary read instructions in place of most of the atomic exchanges while waiting for the mutex to be unlocked.

The two versions of mutexes that I have presented thus far share one key property, which explains why both are called spinlocks. They both engage in busy waiting if the mutex is not immediately available. Recall from my discussion of scheduling that busy waiting means waiting by continually executing instructions that check for the awaited event. A mutex that uses busy waiting is called a *spinlock*.

The alternative to busy waiting is to notify the operating system that the thread needs to wait. The operating system can then change the thread's state to waiting and move it to a wait queue, where it is not eligible for time on the processor. Instead, the scheduler will use the processor to run other threads. When the mutex is unlocked, the waiting thread can be made runnable again. Because this form of mutex makes use of a wait queue, it is called a queuing mutex.

Spinlocks are inefficient, for the same reason as any busy waiting is inefficient. The thread does not make any more headway, no matter how many times it spins around its loop. Therefore, using the processor for a different thread would benefit that other thread without harming the waiting one.

However, there is one flaw in this argument. There is some overhead cost for notifying the operating system of the desire to wait, changing the thread's state, and doing a context switch, with the attendant loss of cache locality. Thus, in a situation where the spinlock needs to spin only briefly before finding the mutex unlocked, the thread might actually waste less time busy waiting than it would waste getting out of other threads' ways. The relative efficiency of spinlocks and queuing mutexes depends on how long the thread needs to wait before the mutex becomes available.

For this reason, spinlocks are appropriate to use for mutexes that are held only briefly, and hence should be quickly acquirable. As an example, the Linux kernel uses spinlocks to protect many of its internal data structures during the brief operations on them. For example, I mentioned that the scheduler keeps the runnable threads in two arrays, organized by priority. Whenever the scheduler wants to insert a thread into one of these arrays, or otherwise operate on them, it locks a spinlock, does the brief operation, and then unlocks the spinlock.

Queuing mutexes are still needed for those cases where a thread might hold a mutex a long time—long enough that other contenders shouldn't busy wait. These mutexes will be more complex. Rather than being stored in a single memory location (as with spinlocks), each mutex will have three components:

- A memory location used to record the mutex's state, 1 for unlocked or 0 for locked.
- A list of threads waiting to acquire the mutex. This list is what allows the scheduler to place the threads in a waiting state, in place of busy waiting. Using the terminology of Chapter 3, this list is a wait queue.
- A cache-conscious spinlock, used to protect against races in operations on the mutex itself.

```
to lock mutex:
  lock mutex.spinlock (in cache-conscious fashion)
  if mutex.state = 1 then
    let mutex.state = 0
    unlock mutex.spinlock
  else
    add current thread to mutex.waiters
    remove current thread from runnable threads
    unlock mutex.spinlock
    yield to a runnable thread
```

Figure 4.10 An attempt to lock a queuing mutex that is already in the locked state causes the thread to join the wait queue, **`mutex.waiters`**.

```
to unlock mutex:
  lock mutex.spinlock (in cache-conscious fashion)
  if mutex.waiters is empty then
    let mutex.state = 1
  else
    move one thread from mutex.waiters to runnable
  unlock mutex.spinlock
```

Figure 4.11 If there is any waiting thread, the unlock operation on a queuing mutex causes a thread to become runnable. Note that in this case, the mutex is left in the locked state; effectively, the locked mutex is being passed directly from one thread to another.

In my pseudocode, I will refer to these three components as **`mutex.state`**, **`mutex.waiters`**, and **`mutex.spinlock`**, respectively.

Under these assumptions, the locking and unlocking operations can be performed as shown in the pseudocode of Figures 4.10 and 4.11. Figures 4.12 and 4.13 illustrate the functioning of these operations. One important feature to note in this mutex design concerns what happens when a thread performs the unlock operation on a mutex that has one or more threads in the waiters list. As you can see in Figure 4.11, the mutex's state variable is not changed from the locked state (0) to the unlocked state (1). Instead, the mutex is left locked, and one of the waiting threads is woken up. In other words, the locked mutex is passed directly from one thread to another, without ever really being unlocked. In Section 4.8.2, I will explain how this design is partially responsible for the so-called convoy phenomenon, which I describe there. In that same section, I will also present an alternative design for mutexes that puts the mutex into the unlocked state.

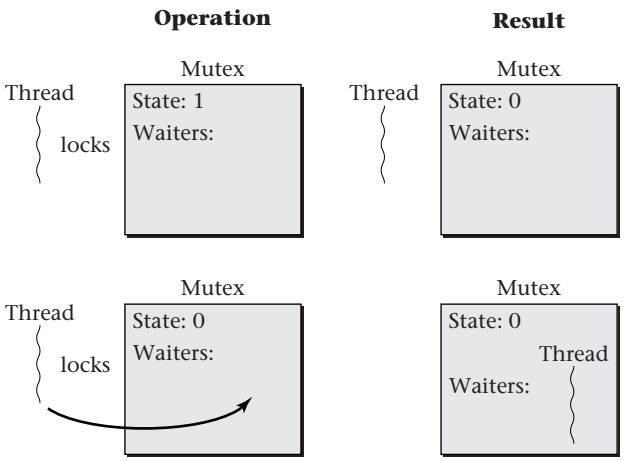

Figure 4.12 Locking a queuing mutex that is unlocked simply changes the mutex's state. Locking an already-locked queuing mutex, on the other hand, puts the thread into the waiters list.

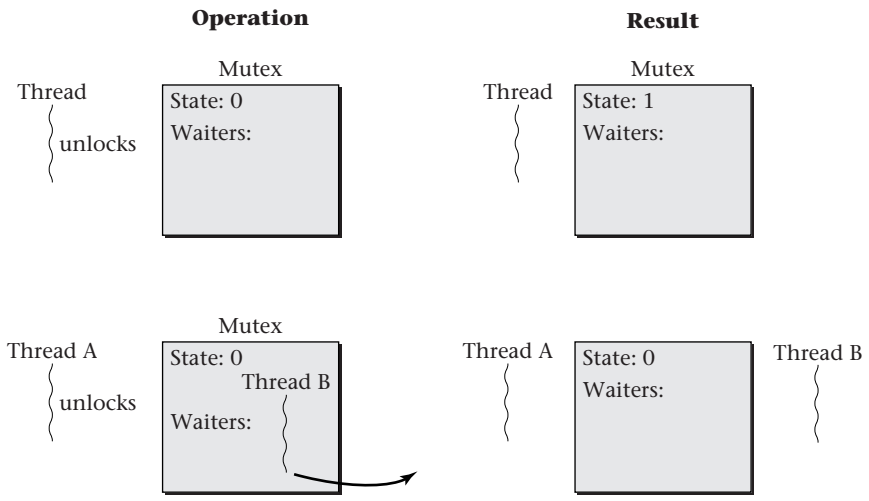

Figure 4.13 Unlocking a queuing mutex with no waiting threads simply changes the mutex's state. Unlocking a queuing mutex with waiting threads, on the other hand, leaves the state set to locked but causes one of the waiting threads to start running again, having acquired the lock.

4.4 Other Synchronization Patterns

Recall that synchronization refers to any form of control over the relative timing of two or more threads. As such, synchronization includes more than just mutual exclusion; a programmer may want to impose some restriction on relative timing other than the rule of one thread at a time. In this section, I present three other patterns of synchronization that crop up over and over again in many applications: bounded buffers, readers/writers locks, and barriers. Sections 4.4.1 through 4.4.3 will describe the desired synchronization; Sections 4.5 and 4.6 show techniques that can be used to achieve the synchronization.

4.4.1 Bounded Buffers

Often, two threads are linked together in a processing *pipeline*. That is, the first thread produces a sequence of values that are consumed by the second thread. For example, the first thread may be extracting all the textual words from a document (by skipping over the formatting codes) and passing those words to a second thread that speaks the words aloud.

One simple way to organize the processing would be by strict alternation between the producing and consuming threads. In the preceding example, the first thread would extract a word, and then wait while the second thread converted it into sound. The second thread would then wait while the first thread extracted the next word. However, this approach doesn't yield any concurrency: only one thread is runnable at a time. This lack of concurrency may result in suboptimal performance if the computer system has two processors, or if one of the threads spends a lot of time waiting for an I/O device.

Instead, consider running the producer and the consumer concurrently. Every time the producer has a new value ready, the producer will store the value into an intermediate storage area, called a *buffer*. Every time the consumer is ready for the next value, it will retrieve the value from the buffer. Under normal circumstances, each can operate at its own pace. However, if the consumer goes to the buffer to retrieve a value and finds the buffer empty, the consumer will need to wait for the producer to catch up. Also, if you want to limit the size of the buffer (that is, to use a *bounded buffer*), you need to make the producer wait if it gets too far ahead of the consumer and fills the buffer. Putting these two synchronization restrictions in place ensures that over the long haul, the rate of the two threads will match up, although over the short term, either may run faster than the other.

You should be familiar with the bounded buffer pattern from businesses in the real world. For example, the cooks at a fast-food restaurant fry burgers concurrently with

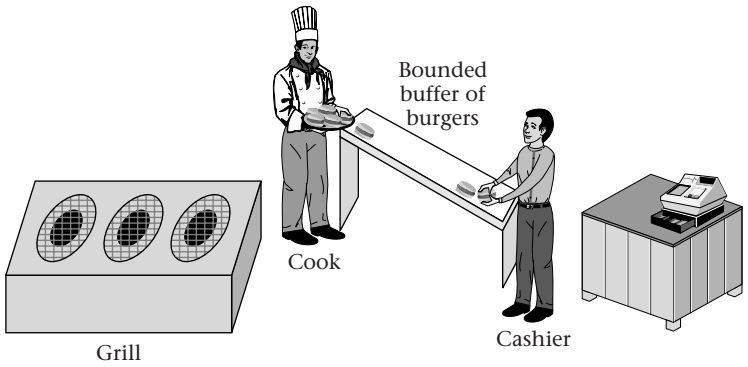

Figure 4.14 A cook fries burgers and places them in a bounded buffer, queued up for later sale. A cashier takes burgers from the buffer to sell. If there are none available, the cashier waits. Similarly, if the buffer area is full, the cook takes a break from frying burgers.

the cashiers selling them. In between the two is a bounded buffer of already-cooked burgers. The exact number of burgers in the buffer will grow or shrink somewhat as one group of workers is temporarily a little faster than the other. Only under extreme circumstances does one group of workers have to wait for the other, as illustrated in Figure 4.14.

One easy place to see bounded buffers at work in computer systems is the *pipe* feature built into UNIX-family operating systems, including Linux and Mac OS X. (Microsoft Windows also now has an analogous feature.) Pipes allow the output produced by one process to serve as input for another. For example, on a Mac OS X system, you could open a terminal window with a shell in it and give the following command:

```
ls | say
```

This runs two programs concurrently. The first, `ls`, lists the files in your current directory. The second one, `say`, converts its textual input into speech and plays it over the computer's speakers. In the shell command, the vertical bar character (`|`) indicates the pipe from the first program to the second. The net result is a spoken listing of your files.

A more mundane version of this example works not only on Mac OS X, but also on other UNIX-family systems such as Linux:

```
ls | tr a-z A-Z
```

Again, this runs two programs concurrently. This time the second one, `tr`, copies characters from its input to its output, with some changes (transliterations) along the way; in this case, replacing lowercase letters `a-z` with the corresponding uppercase letters

`A-Z`. The net result is an uppercase listing of your files. The file listing may get ahead of the transliteration, as long as it doesn't overflow a buffer the operating system provides for the pipe. Once there is a backlog of listed files in the buffer, the transliteration can run as fast as it wants until it exhausts that backlog.

4.4.2 Readers/Writers Locks

My next example of a synchronization pattern is actually quite similar to mutual exclusion. Recall that in the ticket-sales example, the audit function needed to acquire the mutex, even though auditing is a read-only operation, in order to make sure that the audit read a consistent combination of state variables. That design achieved correctness, but at the cost of needlessly limiting concurrency: it prevented two audits from being under way at the same time, even though two (or more) read-only operations cannot possibly interfere with each other. My goal now is to rectify that problem.

A *readers/writers lock* is much like a mutex, except that when a thread locks the lock, it specifies whether it is planning to do any writing to the protected data structure or only reading from it. Just as with a mutex, the lock operation may not immediately complete; instead, it waits until such time as the lock can be acquired. The difference is that any number of readers can hold the lock at the same time, as shown in Figure 4.15;

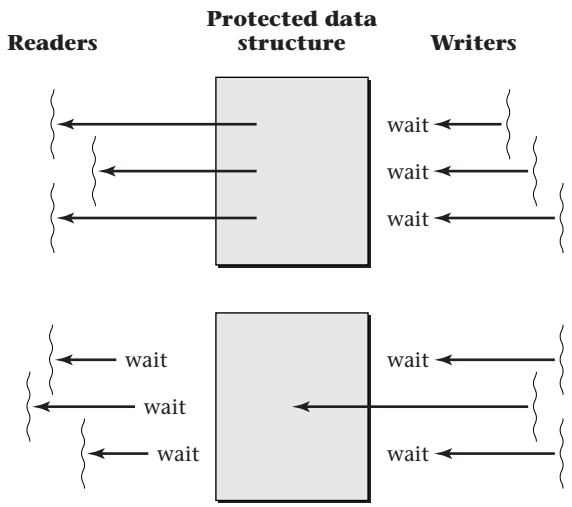

Figure 4.15 A readers/writers lock can be held either by any number of readers or by one writer. When the lock is held by readers, all the reader threads can read the protected data structure concurrently.

they will not wait for each other. A reader will wait, however, if a writer holds the lock. A writer will wait if the lock is held by any other thread, whether by another writer or by one or more readers.

Readers/writers locks are particularly valuable in situations where some of the read-only operations are time-consuming, as when reading a file stored on disk. This is especially true if many readers are expected. The choice between a mutex and a readers/writers lock is a performance trade-off. Because the mutex is simpler, it has lower overhead. However, the readers/writers lock may pay for its overhead by allowing more concurrency.

One interesting design question arises if a readers/writers lock is held by one or more readers and has one or more writers waiting. Suppose a new reader tries to acquire the lock. Should it be allowed to, or should it be forced to wait until after the writers? On the surface, there seems to be no reason for the reader to wait, because it can coexist with the existing readers, thereby achieving greater concurrency. The problem is that an overlapping succession of readers can keep the writers waiting arbitrarily long. The writers could wind up waiting even when the only remaining readers arrived long after the writers did. This is a form of *starvation*, in that a thread is unfairly prevented from running by other threads. To prevent this particular kind of starvation, some versions of readers/writers locks make new readers wait until after the waiting writers.

In Section 4.5, you will learn how you could build readers/writers locks from more primitive synchronization mechanisms. However, because readers/writers locks are so generally useful, they are already provided by many systems, so you may never actually have to build them yourself. The POSIX standard, for example, includes readers/writers locks with procedures such as `pthread_rwlock_init`, `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, and `pthread_rwlock_unlock`. The POSIX standard leaves it up to each individual system how to prioritize new readers versus waiting writers.

The POSIX standard also includes a more specialized form of readers/writers locks specifically associated with files. This reflects my earlier comment that readers/writers locking is especially valuable when reading may be time-consuming, as with a file stored on disk. In the POSIX standard, file locks are available only through the complex `fcntl` procedure. However, most UNIX-family operating systems also provide a simpler interface, `flock`.

4.4.3 Barriers

Barrier synchronization is the last common synchronization pattern I will discuss. Barriers are most commonly used in programs that do large-scale numerical calculations for scientific or engineering applications, such as simulating ocean currents. However,

they may also crop up in other applications, as long as there is a requirement for all threads in a group to finish one phase of the computation before any of them moves on to the next phase. In scientific computations, the threads are often dividing up the processing of a large matrix. For example, ten threads may each process 200 rows of a 2000-row matrix. The requirement for all threads to finish one phase of processing before starting the next comes from the fact that the overall computation is a sequence of matrix operations; parallel processing occurs only within each matrix operation.

When a barrier is created (initialized), the programmer specifies how many threads will be sharing it. Each of the threads completes the first phase of the computation and then invokes the barrier's wait operation. For most of the threads, the wait operation does not immediately return; therefore, the thread calling it cannot immediately proceed. The one exception is whichever thread is the last to call the wait operation. The barrier can tell which thread is the last one, because the programmer specified how many threads there are. When this last thread invokes the wait operation, the wait operation immediately returns. Moreover, all the other waiting threads finally have their wait operations also return, as illustrated in Figure 4.16. Thus, they can now all proceed on to the second phase of the computation. Typically, the same barrier can then be reused between the second and third phases, and so forth. (In other words, the barrier reinitializes its state once it releases all the waiting threads.)

Just as with readers/writers locks, you will see how barriers can be defined in terms of more general synchronization mechanisms. However, once again there is little reason to do so in practice, because barriers are provided as part of POSIX and other widely available APIs.

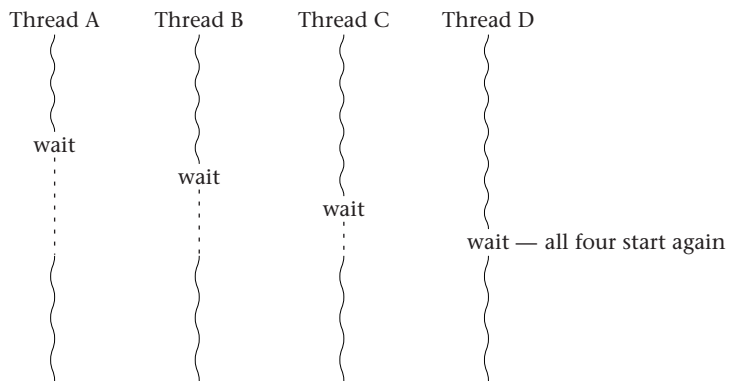

Figure 4.16 A barrier is created for a specific number of threads. In this case, there are four. When the last of those threads invokes the wait operation, all the waiting threads in the group start running again.

4.5 Condition Variables

In order to solve synchronization problems, such as the three described in Section 4.4, you need some mechanism that allows a thread to wait until circumstances are appropriate for it to proceed. A producer may need to wait for buffer space, or a consumer may need to wait for data. A reader may need to wait until a writer has unlocked, or a writer may need to wait for the last reader to unlock. A thread that has reached a barrier may need to wait for all the other threads to do so. Each situation has its own condition for which a thread must wait, and there are many other application-specific conditions besides. (A video playback that has been paused might wait until the user presses the pause button again.)

All these examples can be handled by using *condition variables*, a synchronization mechanism that works in partnership with monitors or with mutexes used in the style of monitors. There are two basic operations on a condition variable: *wait* and *notify*. (Some systems use the name *signal* instead of notify.) A thread that finds circumstances not to its liking executes the wait operation and thereby goes to sleep until such time as another thread invokes the notify operation. For example, in a bounded buffer, the producer might wait on a condition variable if it finds the buffer full. The consumer, upon freeing up some space in the buffer, would invoke the notify operation on that condition variable.

Before delving into all the important details and variants, a concrete example may be helpful. Figure 4.17 shows the Java code for a `BoundedBuffer` class.

Before I explain how this example works, and then return to a more general discussion of condition variables, you should take a moment to consider how you would test such a class. First, it might help to reduce the size of the buffer, so that all qualitatively different situations can be tested more quickly. Second, you need a test program that has multiple threads doing insertions and retrievals, with some way to see the difference between when each operation is started and when it completes. In the case of the retrievals, you will also need to see that the retrieved values are correct. Designing such a test program is surprisingly interesting; you can have this experience in Programming Project 4.5.

In Java, each object has a single condition variable automatically associated with it, just as it has a mutex. The `wait` method waits on the object's condition variable, and the `notifyAll` method wakes up all threads waiting on the object's condition variable. Both of these methods need to be called by a thread that holds the object's mutex. In my `BoundedBuffer` example, I ensured this in a straightforward way by using `wait` and `notifyAll` inside methods that are marked `synchronized`.

Having seen that `wait` and `notifyAll` need to be called with the mutex held, you may spot a problem. If a waiting thread holds the mutex, there will be no way for

```
public class BoundedBuffer {
  private Object[] buffer = new Object[20]; // arbitrary size
  private int numOccupied = 0;
  private int firstOccupied = 0;

  /* invariant: 0 <= numOccupied <= buffer.length
     0 <= firstOccupied < buffer.length
     buffer[(firstOccupied + i) % buffer.length]
     contains the (i+1)th oldest entry,
     for all i such that 0 <= i < numOccupied  */

  public synchronized void insert(Object o)
    throws InterruptedException
  {
    while(numOccupied == buffer.length)
      // wait for space
      wait();
    buffer[(firstOccupied + numOccupied) % buffer.length] = o;
    numOccupied++;
    // in case any retrieves are waiting for data, wake them
    notifyAll();
  }

  public synchronized Object retrieve()
    throws InterruptedException
  {
    while(numOccupied == 0)
      // wait for data
      wait();
    Object retrieved = buffer[firstOccupied];
    buffer[firstOccupied] = null; // may help garbage collector
    firstOccupied = (firstOccupied + 1) % buffer.length;
    numOccupied--;
    // in case any inserts are waiting for space, wake them
    notifyAll();
    return retrieved;
  }
}
```

Figure 4.17 `BoundedBuffer` class using monitors and condition variables.

any other thread to acquire the mutex, and thus be able to call **`notifyAll`**. Until you learn the rest of the story, it seems as though any thread that invokes **`wait`** is doomed to eternal waiting.

The solution to this dilemma is as follows. When a thread invokes the wait operation, it must hold the associated mutex. However, the wait operation releases the mutex before putting the thread into its waiting state. That way, the mutex is available to a potential waker. When the waiting thread is awoken, it reacquires the mutex before the wait operation returns. (In the case of recursive mutexes, as used in Java, the awakening thread reacquires the mutex with the same lock count as before, so that it can still do just as many unlock operations.)

The fact that a waiting thread temporarily releases the mutex helps explain two features of the **`BoundedBuffer`** example. First, the waiting is done at the very beginning of the methods. This ensures that the invariant is still intact when the mutex is released. (More generally, the waiting could happen later, as long as no state variables have been updated, or even as long as they have been put back into an invariant-respecting state.) Second, the waiting is done in a loop; only when the waited-for condition has been verified to hold does the method move on to its real work. The loop is essential because an awoken thread needs to reacquire the mutex, contending with any other threads that are also trying to acquire the mutex. There is no guarantee that the awoken thread will get the mutex first. As such, there is no guarantee what state it will find; it may need to wait again. Another reason the loop is necessary is because on rare occasion the **`wait`** procedure may return without **`notify`** or **`notifyAll`** having been invoked, a circumstance known as a *spurious wakeup*.

When a waiting thread releases the mutex in order to wait on the condition variable, these two actions are done indivisibly. There is no way another thread can acquire the mutex before the first thread has started waiting on the condition variable. This ensures no other thread will do a notify operation until after the thread that wants to wait is actually waiting.

In addition to waiting for appropriate conditions at the top of each method, I have invoked **`notifyAll`** at the end of each method. This position is less crucial, because the **`notifyAll`** method does not release the mutex. The calling thread continues to hold the mutex until it reaches the end of the synchronized method. Because an awoken thread needs to reacquire the mutex, it will not be able to make any headway until the notifying method finishes, regardless of where in that method the notification is done.

One early version of monitors with condition variables (as described by Hoare) used a different approach. The notify operation immediately transferred the mutex to the awoken thread, with no contention from other waiting threads. The thread performing the notify operation then waited until it received the mutex back from

the awoken thread. Today, however, the version I described previously seems to be dominant. In particular, it is used not only in Java, but also in the POSIX API.

The `BoundedBuffer` code in Figure 4.17 takes a very aggressive approach to notifying waiting threads: at the end of any operation all waiting threads are woken using `notifyAll`. This is a very safe approach; if the `BoundedBuffer`'s state was changed in a way of interest to any thread, that thread will be sure to notice. Other threads that don't care can simply go back to waiting. However, the program's efficiency may be improved somewhat by reducing the amount of notification done. Remember, though, that correctness should always come first, with optimization later, if at all. Before optimizing, check whether the simple, correct version actually performs inadequately.

There are two approaches to reducing notification. One is to put the `notifyAll` inside an `if` statement, so that it is done only under some circumstances, rather than unconditionally. In particular, producers should be waiting only if the buffer is full, and consumers should be waiting only if the buffer is empty. Therefore, the only times when notification is needed are when inserting into an empty buffer or retrieving from a full buffer. In Programming Project 4.6, you can modify the code to reflect this and test that it still works.

The other approach to reducing notification is to use the `notify` method in place of `notifyAll`. This way, only a single waiting thread is awoken, rather than all waiting threads. Remember that optimization should be considered only if the straightforward version performs inadequately. This cautious attitude is appropriate because programmers find it rather tricky to reason about whether `notify` will suffice. As such, this optimization is quite error-prone. In order to verify that the change from `notifyAll` to `notify` is correct, you need to check two conditions:

1. There is no danger of waking too few threads. Either you have some way to know that only one is waiting, or you know that only one would be able to proceed, with the others looping back to waiting.
2. There is no danger of waking the wrong thread. Either you have some way to know that only one is waiting, or you know that all are equally able to proceed. If there is any thread that could proceed if it got the mutex first, then all threads have that property. For example, if all the waiting threads are executing the identical while loop, this condition will be satisfied.

In Exercise 4.4, you can show that these two conditions do not hold for the `BoundedBuffer` example: replacing `notifyAll` with `notify` would not be safe in this case. This is true even if the notification operation is done unconditionally, rather than inside an `if` statement.

One limitation of Java is that each object has only a single condition variable. In the `BoundedBuffer` example, any thread waits on that one condition variable, whether it is waiting for space in the `insert` method or for data in the `retrieve` method. In a system which allows multiple condition variables to be associated with the same monitor (or mutex), you could use two different condition variables. That would allow you to specifically notify a thread waiting for space (or one waiting for data).

The POSIX API allows multiple condition variables per mutex. In Programming Project 4.7 you can use this feature to rewrite the `BoundedBuffer` example with two separate condition variables, one used to wait for space and the other used to wait for data.

POSIX condition variables are initialized with `pthread_cond_init` independent of any particular mutex; the mutex is instead passed as an argument to `pthread_cond_wait`, along with the condition variable being waited on. This is a somewhat error-prone arrangement, because all concurrent waiters need to pass in the same mutex. The operations corresponding to `notify` and `notifyAll` are called `pthread_cond_signal` and `pthread_cond_broadcast`.

The same technique I illustrated with `BoundedBuffer` can be applied equally well for readers/writers locks or barriers; I leave these as Programming Projects 4.8 and 4.11. More importantly, the same technique will also work for application-specific synchronization needs. For example, a video player might have a state variable that records whether it is currently paused. The playback thread checks that variable before displaying each frame, and if paused, waits on a condition variable. The user-interface thread sets the variable in response to the user pressing the pause button. When the user interface puts the variable into the unpaused state, it does a notify operation on the condition variable. You can develop an application analogous to this in Programming Project 4.3.

4.6 Semaphores

You have seen that monitors with condition variables are quite general and can be used to synthesize other more special-purpose synchronization mechanisms, such as readers/writers locks. Another synchronization mechanism with the same generality is the semaphore. For most purposes, semaphores are less natural, resulting in more error-prone code. In those applications where they are natural (for example, bounded buffers), they result in very succinct, clear code. That is probably not the main reason

for their continued use, however. Instead, they seem to be hanging on largely out of historical inertia, having gotten a seven- to nine-year head start over monitors. (Semaphores date to 1965, as opposed to the early 1970s for monitors.)

A *semaphore* is essentially an unsigned integer variable, that is, a variable that can take on only nonnegative integer values. However, semaphores may not be freely operated on with arbitrary arithmetic. Instead, only three operations are allowed:

- At the time the semaphore is created, it may be initialized to any nonnegative integer of the programmer's choice.
- A semaphore may be increased by 1. The operation to do this is generally called either `up` or `V`. The letter `V` is short for a Dutch word that made sense to Dijkstra, the 1965 originator of semaphores. I will use `up`.
- A semaphore may be decreased by 1. The operation to do this is frequently called either `down` or `P`. Again, `P` is a Dutch abbreviation. I will use `down`. Because the semaphore's value must stay nonnegative, the thread performing a `down` operation waits if the value is 0. Only once another thread has performed an `up` operation to make the value positive does the waiting thread continue with its `down` operation.

One common use for semaphores is as mutexes. If a semaphore is initialized to 1, it can serve as a mutex, with `down` as the locking operation and `up` as the unlocking operation. Assuming that locking and unlocking are properly paired, the semaphore will only ever have the values 0 and 1. When it is locked, the value will be 0, and any further attempt to lock it (using `down`) will be forced to wait. When it is unlocked, the value will be 1, and locking can proceed. Note, however, that semaphores used in this limited way have no advantage over mutexes. Moreover, if a program bug results in an attempt to unlock an already unlocked mutex, a special-purpose mutex could signal the error, whereas a general-purpose semaphore will simply increase to 2, likely causing nasty behavior later when two threads are both allowed to execute `down`.

A better use for semaphores is for keeping track of the available quantity of some resource, such as free spaces or data values in a bounded buffer. Whenever a thread creates a unit of the resource, it increases the semaphore. Whenever a thread wishes to consume a unit of the resource, it first does a `down` operation on the semaphore. This both forces the thread to wait until at least one unit of the resource is available and stakes the thread's claim to that unit.

Following this pattern, the `BoundedBuffer` class can be rewritten to use semaphores, as shown in Figure 4.18. This assumes the availability of a class of semaphores.

```
public class BoundedBuffer {
  private java.util.List buffer =
    java.util.Collections.synchronizedList
    (new java.util.LinkedList());

  private static final int SIZE = 20; // arbitrary

  private Semaphore occupiedSem = new Semaphore(0);
  private Semaphore freeSem = new Semaphore(SIZE);

  /* invariant: occupiedSem + freeSem = SIZE
     buffer.size() = occupiedSem
     buffer contains entries from oldest to youngest */

  public void insert(Object o) throws InterruptedException{
    freeSem.down();
    buffer.add(o);
    occupiedSem.up();
  }

  public Object retrieve() throws InterruptedException{
    occupiedSem.down();
    Object retrieved = buffer.remove(0);
    freeSem.up();
    return retrieved;
  }
}
```

Figure 4.18 Alternative `BoundedBuffer` class, using semaphores.

In Programming Project 4.12, you can write a `Semaphore` class using Java's built-in mutexes and condition variables.

In order to show semaphores in the best possible light, I also moved away from using an array to store the buffer. Instead, I used a `List`, provided by the Java API. If, in Programming Project 4.13, you try rewriting this example to use an array (as in Figure 4.17), you will discover two blemishes. First, you will need the `numOccupied` integer variable, as in Figure 4.17. This duplicates the information contained in `occupiedSem`, simply in a different form. Second, you will need to introduce explicit mutex synchronization with `synchronized` statements around the code that updates the nonsemaphore state variables. With those complications, semaphores lose some of their charm. However, by using a `List`, I hid the extra complexity.

4.7 Deadlock

In Chapter 2, I introduced concurrency as a way to solve problems of responsiveness and throughput. Unfortunately, concurrency created its own problem—races. Therefore, I introduced synchronization to solve the problem of races. The obvious question is, what new problems arise from synchronization? One easy answer is that synchronization has reintroduced the original responsiveness and throughput problems to some lesser degree, because synchronization reduces concurrency. However, as you will see in this section, synchronization also creates an entirely new problem, and one that is potentially more serious. Section 4.7.1 explains this problem, known as deadlock, whereby threads can wind up permanently waiting. Sections 4.7.2 through 4.7.4 explain three different solutions to the problem.

4.7.1 The Deadlock Problem

To illustrate what a deadlock is, and how one can arise, consider a highly simplified system for keeping bank accounts. Suppose each account is an object with two components: a mutex and a current balance. A procedure for transferring money from one account to another might look as follows, in pseudocode:

```
to transfer amount from sourceAccount to destinationAccount:
  lock sourceAccount.mutex
  lock destinationAccount.mutex
  sourceAccount.balance = sourceAccount.balance - amount
  destinationAccount.balance = destinationAccount.balance + amount
  unlock sourceAccount.mutex
  unlock destinationAccount.mutex
```

Suppose I am feeling generous and transfer $100 from **myAccount** to **yourAccount**. Suppose you are feeling even more generous and transfer $250 from **yourAccount** to **myAccount**. With any luck, at the end I should be $150 richer and you should be $150 poorer. If either transfer request is completed before the other starts, this is exactly what happens. However, what if the two execute concurrently?

The mutexes prevent any race condition, so you can be sure that the accounts are not left in an inconsistent state. Note that we have locked both accounts for the entire duration of the transfer, rather than locking each only long enough to update its balance. That way, an auditor can't see an alarming situation where money has disappeared from one account but not yet appeared in the other account.

However, even though there is no race, not even with an auditor, all is not well. Consider the following sequence of events:

1. I lock the source account of my transfer to you. That is, I lock `myAccount.mutex`.
2. You lock the source account of your transfer to me. That is, you lock `yourAccount.mutex`.
3. I try to lock the destination account of my transfer to you. That is, I try to lock `yourAccount.mutex`. Because you already hold this mutex, I am forced to wait.
4. You try to lock the destination account of your transfer to me. That is, you try to lock `myAccount.mutex`. Because I already hold this mutex, you are forced to wait.

At this point, each of us is waiting for the other: we have deadlocked.

More generally, a *deadlock* exists whenever there is a cycle of threads, each waiting for some resource held by the next. In the example, there were two threads and the resources involved were two mutexes. Although deadlocks can involve other resources as well (consider readers/writers locks, for example), I will focus on mutexes for simplicity.

As an example of a deadlock involving more than two threads, consider generalizing the preceding scenario of transferring money between bank accounts. Suppose, for example, that there are five bank accounts, numbered 0 through 4. There are also five threads. Each thread is trying to transfer money from one account to another, as shown in Figure 4.19. As before, each transfer involves locking the source and destination accounts. Once again, the threads can deadlock if each one locks the source account first, and then tries to lock the destination account. This situation is

Thread	Source Account	Destination Account
0	0	1
1	1	2
2	2	3
3	3	4
4	4	0

Figure 4.19 Each of five threads tries to transfer money from a source account to a destination account. If each thread locks its source account, none will be able to proceed by locking its destination account.

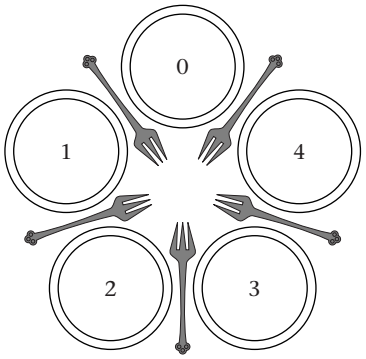

Figure 4.20 Five philosophers, numbered 0 through 4, have places around a circular dining table. There is a fork between each pair of adjacent places. When each philosopher tries to pick up two forks, one at a time, deadlock can result.

much more famous when dressed up as the dining philosophers problem, which I describe next.

In 1972, Dijkstra wrote about a group of five philosophers, each of whom had a place at a round dining table, where they ate a particularly difficult kind of spaghetti that required two forks. There were five forks at the table, one between each pair of adjacent plates, as shown in Figure 4.20. Apparently Dijkstra was not concerned with communicable diseases such as mononucleosis, because he thought it was OK for the philosophers seated to the left and right of a particular fork to share it. Instead, he was concerned with the possibility of deadlock. If all five philosophers start by picking up their respective left-hand forks and then wait for their right-hand forks to become available, they wind up deadlocked. In Exploration Project 4.2, you can try out a computer simulation of the dining philosophers. In that same Exploration Project, you can also apply the deadlock prevention approach described in Section 4.7.2 to the dining philosophers problem.

Deadlocks are usually quite rare even if no special attempt is made to prevent them, because most locks are not held very long. Thus, the window of opportunity for deadlocking is quite narrow, and, like races, the timing must be exactly wrong. For a very noncritical system, one might choose to ignore the possibility of deadlocks. Even if the system needs the occasional reboot due to deadlocking, other malfunctions will probably be more common. Nonetheless, you should learn some options for dealing with deadlocks, both because some systems are critical and because ignoring a known problem is unprofessional. In Sections 4.7.2 through 4.7.4, I explain three of the most practical ways to address the threat of deadlocks.

4.7.2 Deadlock Prevention Through Resource Ordering

The ideal way to cope with deadlocks is to prevent them from happening. One very practical technique for deadlock prevention can be illustrated through the example of transferring money between two bank accounts. Each of the two accounts is stored somewhere in the computer's memory, which can be specified through a numerical address. I will use the notation **`min(account1, account2)`** to mean whichever of the two account objects occurs at the lower address (earlier in memory). Similarly, I will use **`max(account1, account2)`** to mean whichever occurs at the higher address. I can use this ordering on the accounts (or any other ordering, such as by account number) to make a deadlock-free transfer procedure:

```
to transfer amount from sourceAccount to destinationAccount:
  lock min(sourceAccount, destinationAccount).mutex
  lock max(sourceAccount, destinationAccount).mutex
  sourceAccount.balance = sourceAccount.balance - amount
  destinationAccount.balance = destinationAccount.balance + amount
  unlock sourceAccount.mutex
  unlock destinationAccount.mutex
```

Now if I try transferring money to you, and you try transferring money to me, we will both lock the two accounts' mutexes in the same order. No deadlock is possible; one transfer will run to completion, and then the other.

The same technique can be used whenever all the mutexes (or other resources) to be acquired are known in advance. Each thread should acquire the resources it needs in an agreed-upon order, such as by increasing memory address. No matter how many threads and resources are involved, no deadlock can occur.

As one further example of this technique, you can look at some code from the Linux kernel. Recall from Chapter 3 that the scheduler keeps the run queue, which holds runnable threads, in a pair of arrays organized by priority. In the kernel source code, this structure is known as a runqueue. Each processor in a multiprocessor system has its own runqueue. When the scheduler moves a thread from one processor's runqueue to another's, it needs to lock both runqueues. Figure 4.21 shows the code to do this. Note that this procedure uses the deadlock prevention technique with one refinement: it also tests for the special case that the two runqueues are in fact one and the same.

Deadlock prevention is not always possible. In particular, the ordering technique I showed cannot be used if the mutexes that need locking only become apparent one by one as the computation proceeds, such as when following a linked list or other pointer-based data structure. Thus, you need to consider coping with deadlocks, rather than only preventing them.

```
static inline
void double_rq_lock(runqueue_t *rq1, runqueue_t *rq2)
{
        if (rq1 == rq2)
                spin_lock(&rq1->lock);
        else {
                if (rq1 < rq2) {
                        spin_lock(&rq1->lock);
                        spin_lock(&rq2->lock);
                } else {
                        spin_lock(&rq2->lock);
                        spin_lock(&rq1->lock);
                }
        }
}
```

Figure 4.21 The Linux scheduler uses deadlock prevention when locking two runqueues.

4.7.3 Ex Post Facto Deadlock Detection

In order to diagnose deadlocks, you need some information about who is waiting for whom. Suppose that each mutex records not just whether it is locked or unlocked, but also which thread it is held by, if any. (This information may be useful for unrelated purposes as well, such as implementing recursive or error-checking mutexes.) Additionally, when a thread is unable to immediately acquire a mutex and is put into a waiting state, you can record which mutex it is waiting for. With this information, you can construct a *resource allocation graph*. Figure 4.22 shows an example graph for Section 4.7.1's sample deadlock between bank account transfers. Squares are threads and circles are mutexes. The arrows show which mutex each thread is waiting to acquire and which thread each mutex is currently held by. Because the graph has a cycle, it shows that the system is deadlocked.

A system can test for deadlocks periodically or when a thread has waited an unreasonably long time for a lock. In order to test for a deadlock, the system uses a standard graph algorithm to check whether the resource allocation graph contains a cycle. With the sort of mutexes described in this book, each mutex can be held by at most one thread and each thread is waiting for at most one mutex, so no vertex in the graph has an out-degree greater than 1. This allows a somewhat simpler graph search than in a fully-general directed graph.

Once a deadlock is detected, a painful action is needed in order to recover: one of the deadlocked threads must be forcibly terminated, or at least rolled back to an earlier state, so as to free up the mutexes it holds. In a general computing environment, where

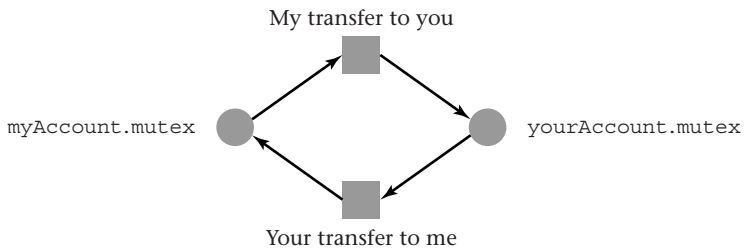

Figure 4.22 The cycle in this resource allocation graph indicates a deadlock. Each square represents a thread and each circle a mutex. An arrow from a square to a circle shows a thread waiting for a mutex, whereas an arrow from a circle to a square shows a mutex being held by a thread.

threads have no clean way to be rolled back, this is a bit akin to freeing yourself from a bear trap by cutting off your leg. For this reason, ex post facto deadlock detection is not common in general-purpose operating systems.

One environment in which ex post facto deadlock detection and recovery works cleanly is database systems, with their support for atomic transactions. I will explain atomic transactions in Chapter 5; for now, you need only understand that a transaction can cleanly be rolled back, such that all the updates it made to the database are undone. Because this infrastructure is available, database systems commonly include deadlock detection. When a deadlock is detected, one of the transactions fails and can be rolled back, undoing all its effects and releasing all its locks. This breaks the deadlock and allows the remaining transactions to complete. The rolled-back transaction can then be restarted.

Figure 4.23 shows an example scenario of deadlock detection taken from the Oracle database system. This transcript shows the time interleaving of two different sessions connected to the same database. One session is shown at the left margin, while the other session is shown indented four spaces. Command lines start with the system's prompt, `SQL>`, and then contain a command typed by the user. Each command line is broken on to a second line, to fit the width of this book's pages. Explanatory comments start with `--`. All other lines are output. In Chapter 5, I will show the recovery from this particular deadlock as part of my explanation of transactions.

4.7.4 Immediate Deadlock Detection

The two approaches to deadlocks presented thus far are aimed at the times before and after the moment when deadlock occurs. One arranges that the prerequisite circumstances leading to deadlock do not occur, while the other notices that deadlock already

```
SQL> update accounts set balance = balance - 100
                     where account_number = 1;

1 row updated.

    SQL> update accounts set balance = balance - 250
                         where account_number = 2;

    1 row updated.

SQL> update accounts set balance = balance + 100
                     where account_number = 2;
-- note no response, for now this SQL session is hanging

    SQL> update accounts set balance = balance + 250
                         where account_number = 1;
    -- this session hangs, but in the other SQL session we get
    -- the following error message:

update accounts set balance = balance + 100
                where account_number = 2
       *
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
```

Figure 4.23 The Oracle database system detects a deadlock between two sessions connected to the same database. One session, shown at the left margin, is transferring $100 from account 1 to account 2. The other session, shown indented, is transferring $250 from account 2 to account 1. Each update statement locks the account being updated. Therefore, each session hangs when it tries locking the account that the other session has previously locked.

has occurred, so that the mess can be cleaned up. Now I will turn to a third alternative: intervening at the very moment when the system would otherwise deadlock. Because this intervention requires techniques similar to those discussed in Section 4.7.3, this technique is conventionally known as a form of deadlock detection rather than deadlock prevention, even though from a literal perspective the deadlock is prevented from happening.

As long as no deadlock is ever allowed to occur, the resource allocation graph will remain acyclic, that is, free of cycles. Each time a thread tries to lock a mutex, the system can act as follows:

- If the mutex is unlocked, lock it and add an edge from the mutex to the thread, so as to indicate which thread now holds the lock.

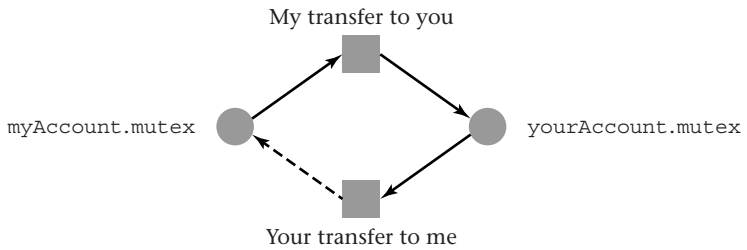

Figure 4.24 In this resource graph, the solid arrows indicate that my transfer holds **`myAccount.mutex`**, your transfer holds **`yourAccount.mutex`**, and my transfer is waiting for **`yourAccount.mutex`**. The dashed arrow indicates a request currently being made by your transfer to lock **`myAccount.mutex`**. If this dashed arrow is added, a cycle is completed, indicating a deadlock. Therefore, the request will fail rather than enter a state of waiting.

- If the mutex is locked, follow the chain of edges from it until that chain dead-ends. (It must, because the graph is acyclic.) Is the end of the chain the same as the thread trying to lock the mutex?
  - If not, add an edge showing that the thread is waiting for the mutex, and put the thread into a waiting state.
  - If the end of the chain is the same thread, adding the extra edge would complete a cycle, as shown in Figure 4.24. Therefore, don't add the edge, and don't put the thread into a waiting state. Instead, return an error code from the lock request (or throw an exception), indicating that the mutex could not be locked because a deadlock would have resulted.

Notice that the graph search here is somewhat simpler than in ex post facto deadlock detection, because the graph is kept acyclic. Nonetheless, the basic idea is the same as deadlock detection, just done proactively rather than after the fact. As with any deadlock detection, some form of roll-back is needed; the application program that tried to lock the mutex must respond to the news that its request could not be granted. The application program must not simply try again to acquire the same mutex, because it will repeatedly get the same error code. Instead, the program must release the locks it currently holds and then restart from the beginning. The chance of needing to repeat this response can be reduced by sleeping briefly after releasing the locks and before restarting.

Designing an application program to correctly handle immediate deadlock detection can be challenging. The difficulty is that before the program releases its existing locks, it should restore the objects those locks were protecting to a consistent state. One case in which immediate deadlock detection can be used reasonably easily is in a program that acquires all its locks before it modifies any objects.

One example of immediate deadlock detection is in Linux and Mac OS X, for the readers/writers locks placed on files using `fcntl`. If a lock request would complete a cycle, the `fcntl` procedure returns the error code `EDEADLK`. However, this deadlock detection is not a mandatory part of the POSIX specification for `fcntl`.

4.8 The Interaction of Synchronization with Scheduling

Recall that the scheduler controls which runnable thread runs, and synchronization actions performed by the running thread control which threads are runnable. Therefore, synchronization and scheduling interact with one another. Two forms of interaction, known as priority inversion and the convoy phenomenon, are particularly interesting. Said another way, they can cause lots of grief. Each can subvert the prioritization of threads, and the convoy phenomenon can also greatly increase the context-switching rate and hence decrease system throughput.

4.8.1 Priority Inversion

When a priority-based scheduler is used, a high-priority thread should not have to wait while a low-priority thread runs. If threads of different priority levels share mutexes or other synchronization primitives, some minor violations of priority ordering are inevitable. For example, consider the following sequence of events involving two threads (high-priority and low-priority) that share a single mutex:

1. The high-priority thread goes into the waiting state, waiting for an I/O request to complete.
2. The low-priority thread runs and acquires the mutex.
3. The I/O request completes, making the high-priority thread runnable again. It preempts the low-priority thread and starts running.
4. The high-priority thread tries to acquire the mutex. Because the mutex is locked, the high-priority thread is forced to wait.
5. The low-priority thread resumes running.

At this point, a high-priority thread is waiting while a low-priority thread runs. However, this temporary violation of priority ordering is not a big deal, because programmers generally ensure that no thread holds a mutex for very long. As such, the low-priority thread will soon release the mutex and allow the high-priority thread to run.

However, another, more insidious problem can lead to longer-term violation of priority order (that is, *priority inversion*). Suppose there are three threads, of low, medium, and high priority. Consider this sequence of events:

1. The high- and medium-priority threads both go into the waiting state, each waiting for an I/O request to complete.
2. The low-priority thread runs and acquires the mutex.
3. The two I/O requests complete, making the high- and medium-priority threads runnable. The high-priority thread preempts the low-priority thread and starts running.
4. The high-priority thread tries to acquire the mutex. Because the mutex is locked, the high-priority thread is forced to wait.
5. At this point, the medium-priority thread has the highest priority of those that are runnable. Therefore, it runs.

In this situation, the medium-priority thread is running and indirectly keeping the high-priority thread from running. (The medium-priority thread is blocking the low-priority thread by virtue of their relative priorities. The low-priority thread is blocking the high-priority thread by holding the mutex.) The medium-priority thread could run a long time. In fact, a whole succession of medium-priority threads with overlapping lifetimes could come and go, and the high-priority thread would wait the whole time despite its higher priority. Thus, the priority inversion could continue for an arbitrarily long time.

One "solution" to the priority inversion problem is to avoid fixed-priority scheduling. Over time, a decay usage scheduler will naturally lower the priority of the medium-priority thread that is running. Eventually, it will drop below the low-priority thread, which will then run and free the mutex, allowing the high-priority thread to run. However, a succession of medium-priority threads, none of which runs for very long, could still hold up the high-priority thread arbitrarily long. Therefore, Microsoft Windows responds to priority inversion by periodically boosting the priority of waiting low-priority processes.

This first "solution" has two shortcomings. First, it may be sluggish in responding to a priority inversion. Second, fixed-priority scheduling is desirable in some applications, such as real-time systems. Therefore, a genuine solution to the priority inversion problem is needed—one that makes the problem go away, rather than just limiting the duration of its effect. The genuine solution is *priority inheritance*.

Priority inheritance is a simple idea: any thread that is waiting for a mutex temporarily "lends" its priority to the thread that holds the mutex. A thread that holds mutexes runs with the highest priority among its own priority and those priorities it

has been lent by threads waiting for the mutexes. In the example with three threads, priority inheritance will allow the low-priority thread that holds the mutex to run as though it were high-priority until it unlocks the mutex. Thus, the truly high-priority thread will get to run as soon as possible, and the medium-priority thread will have to wait.

Notice that the high-priority thread has a very selfish motive for letting the low-priority thread use its priority: it wants to get the low-priority thread out of its way. The same principle can be applied with other forms of scheduling than priority scheduling. By analogy with priority inheritance, one can have *deadline inheritance* (for Earliest Deadline First scheduling) or even a lending of processor allocation shares (for proportional-share scheduling).

4.8.2 The Convoy Phenomenon

I have remarked repeatedly that well-designed programs do not normally hold any mutex for very long; thus, attempts to lock a mutex do not normally encounter contention. This is important because locking a mutex with contention is much more expensive. In particular, the big cost of a request to lock an already-locked mutex is context switching, with the attendant loss of cache performance. Unfortunately, one particularly nasty interaction between scheduling and synchronization, known as the *convoy phenomenon*, can sometimes cause a heavily used mutex to be perpetually contended, causing a large performance loss. Moreover, the convoy phenomenon can subvert scheduling policies, such as the assignment of priorities. In this subsection, I will explain the convoy phenomenon and examine some solutions.

Suppose a system has some very central data structure, protected by a mutex, which each thread operates on fairly frequently. Each time a thread operates on the structure, the thread locks the mutex before and unlocks it after. Each operation is kept as short as possible. Because they are frequent, however, the mutex spends some appreciable fraction of the time locked, perhaps 5 percent.

The scheduler may at any point preempt a thread. For example, the thread may have consumed its allocated time slice. In the example situation where the mutex is locked 5 percent of the time, it would not be very surprising if after a while, a thread were preempted while it held the mutex. When this happens, the programmer who wrote that thread loses all control over how long it holds the mutex locked. Even if the thread was going to unlock the mutex in its very next instruction, it may not get the opportunity to execute that next instruction for some time to come. If the processor is dividing its time among N runnable threads of the same priority level, the thread holding the mutex will presumably not run again for at least N times the context-switching time, even if the other threads all immediately block.

In this situation, a popular mutex is held for a long time. Meanwhile, other threads are running. Because the mutex is a popular one, the chances are good those other threads will try to acquire it. Because the mutex is locked, all the threads that try to acquire the mutex will be queued on its wait queue. This queue of threads is the *convoy*, named by analogy with the unintentional convoy of vehicles that develops behind one slow vehicle on a road with no passing lane. As you will see, this convoy spells trouble.

Eventually, the scheduler will give a new time slice to the thread that holds the mutex. Because of that thread's design, it will quickly unlock the mutex. When that happens, ownership of the mutex is passed to the first thread in the wait queue, and that thread is made runnable. The thread that unlocked the mutex continues to run, however. Because it was just recently given a new time slice, one might expect it to run a long time. However, it probably won't, because before too terribly long, it will try to reacquire the popular mutex and find it locked. ("Darn," it might say, "I shouldn't have given that mutex away to the first of the waiters. Here I am needing it again myself.") Thus, the thread takes its place at the back of the convoy, queued up for the mutex.

At this point, the new holder of the mutex gets to run, but it too gives away the mutex, and hence is unlikely to run a full time slice before it has to queue back up. This continues, with each thread in turn moving from the front of the mutex queue through a brief period of execution and back to the rear of the queue. There may be slight changes in the makeup of the convoy—a thread may stop waiting on the popular mutex, or a new thread may join—but seen in the aggregate, the convoy can persist for a very long time.

This situation causes two problems. First, the context-switching rate goes way up; instead of one context switch per time slice, there is now one context switch per attempt to acquire the popular mutex. The overhead of all those context switches will drive down the system throughput. Second, the scheduler's policy for choosing which thread to run is subverted. For example, in a priority scheduler, the priorities will not govern how the threads run. The reason for this is simple: the scheduler can choose only among the runnable threads, but with the convoy phenomenon there will be only one runnable thread; all the others will be queued up for the mutex.

When I described mutexes, I said that each mutex contains a wait queue—a list of waiting threads. I implied that this list is maintained in a first-in first-out (FIFO) basis, that is, as a true queue. If so, then the convoy threads will essentially be scheduled in a FIFO round-robin, independent of the scheduler policy (for example, priorities), because the threads are dispatched from the mutex queue rather than the scheduler's run queue.

This loss of prioritization can be avoided by handling the mutex's wait queue in priority order the same way as the run queue, rather than FIFO. When a mutex is

unlocked with several threads waiting, ownership of the mutex could be passed not to the thread that waited the longest, but rather to the one with the highest priority.

Changing which one thread is moved from the mutex's waiters list to become runnable does not solve the throughput problem, however. The running thread is still going to have the experience I anthropomorphized as "Darn, I shouldn't have given that mutex away." The context-switching rate will still be one switch per lock acquisition. The convoy may reorder itself, but it will not dissipate.

Therefore, stronger medicine is needed for popular mutexes. Instead of the mutexes I showed in Figures 4.10 and 4.11 on page 87, you can use the version shown in Figure 4.25.

When a popular mutex is unlocked, *all* waiting threads are made runnable and moved from the waiters list to the runnable threads list. However, ownership of the mutex is not transferred to any of them. Instead, the mutex is left in the unlocked state, with **`mutex.state`** equal to 1. That way, the running thread will not have to say "Darn." It can simply relock the mutex; over the course of its time slice, it may lock and unlock the mutex repeatedly, all without context switching.

```
to lock mutex:
  repeat
    lock mutex.spinlock (in cache-conscious fashion)
    if mutex.state = 1 then
      let mutex.state = 0
      unlock mutex.spinlock
      let successful = true
    else
      add current thread to mutex.waiters
      remove current thread from runnable threads
      unlock mutex.spinlock
      yield to a runnable thread
      let successful = false
  until successful

to unlock mutex:
  lock mutex.spinlock (in cache-conscious fashion)
  let mutex.state = 1
  move all threads from mutex.waiters to runnable
  unlock mutex.spinlock
```

Figure 4.25 To protect against convoys, the unlock operation sets the mutex's state to unlocked and makes all waiting threads runnable. Each awoken thread loops back to trying to lock the mutex. This contrasts with the prior version of mutexes, in which one thread was awoken with the mutex left in its locked state.

Because the mutex is held only 5 percent of the time, the mutex will probably not be held when the thread eventually blocks for some other reason (such as a time slice expiration). At that point, the scheduler will select one of the woken threads to run. Note that this will naturally follow the normal scheduling policy, such as priority order.

The woken thread selected to run next did not have the mutex ownership directly transferred to it. Therefore, it will need to loop back to the beginning of the mutex acquisition code, as will each thread in turn when it is scheduled. However, most of the time the threads will find the mutex unlocked, so this won't be expensive. Also, because each thread will be able to run for a normal period without context-switching overhead per lock request, the convoy will dissipate.

The POSIX standard API for mutexes requires that one or the other of the two prioritization-preserving approaches be taken. At a minimum, if ownership of a mutex is directly transferred to a waiting thread, that waiting thread must be selected based on the normal scheduling policy rather than FIFO. Alternatively, a POSIX-compliant mutex implementation can simply dump all the waiting threads back into the scheduler and let it sort them out, as in Figure 4.25.

4.9 Security and Synchronization

A system can be insecure for two reasons: either because its security policies are not well designed, or because some bug in the code enforcing those policies allows the enforcement to be bypassed. For example, you saw in Chapter 3 that a denial of service attack can be mounted by setting some other user's thread to a very low priority. I remarked that as a result, operating systems only allow a thread's priority to be changed by its owner. Had this issue been overlooked, the system would be insecure due to an inadequate policy. However, the system may still be insecure if clever programmers can find a way to bypass this restriction using some low-level bug in the operating system code.

Many security-critical bugs involve synchronization, or more accurately, the lack of synchronization—the bugs are generally race conditions resulting from inadequate synchronization. Four factors make race conditions worth investigation by crackers:

- Any programmer of a complicated concurrent system is likely to introduce race bugs, because concurrency and synchronization are hard to reason about.
- Normal testing of the system is unlikely to have eliminated these bugs, because the system will still work correctly the vast majority of the time.
- Although the race might almost never occur in a normal operation, the cracker may be able to trigger the race by understanding it and carefully staging the necessary sequence of events. Even if the odds can't be improved beyond one in ten thousand (for example), the cracker can easily program a computer to loop through the attempt tens of thousands of times until the lucky timing happens.

- Races allow seemingly impossible situations, defeating the system designer's careful security reasoning.

As a hypothetical example, assume that an operating system had a feature for changing a thread's priority when given a pointer to a block of memory containing two values: an identifier for the thread to be changed and the new priority. Let's call these **`request.thread`** and **`request.priority`**. Suppose that the code looked like this:

```
if request.thread is owned by the current user then
  set request.thread's priority to request.priority
else
  return error code for invalid request
```

Can you see the race? All a cracker needs to do is start out with **`request.thread`** being a worthless thread he or she owns, and then modify **`request.thread`** to be the victim thread after the ownership check but before the priority is set. If the timing doesn't work out, no great harm is done, and the cracker can try again.

This particular example is not entirely realistic in a number of regards, but it does illustrate a particular class of races often contributing to security vulnerabilities: so-called *TOCTTOU* races, an acronym for *Time Of Check To Time Of Use*. An operating system designer would normally guard against this particular TOCTTOU bug by copying the whole request structure into protected memory before doing any checking. However, other TOCTTOU bugs arise with some regularity. Often, they are not in the operating system kernel itself, but rather in a privileged program.

For example, suppose an email delivery program is granted the privilege of writing into any file, independent of file ownership or normal protections, so that it can deliver each user's mail into that user's mail file. Before delivering mail into a mail file, it will check that the mail file is a normal file that is really in the expected location, not an indirect reference (symbolic link) to a file located elsewhere. (I will explain symbolic links in Chapter 8, when I cover file systems. The details are not important here.) That way, you cannot trick the mail delivery program into writing into some sensitive file. Or can you? Perhaps by changing from a genuine mail file to a symbolic link at just the right moment, you can exploit a TOCTTOU vulnerability. Sun Microsystems had this particular problem with their mail software in the early 1990s.

Exercises

4.1 As an example of a race condition, I showed how two threads could each dispense the last remaining ticket by each checking **`seatsRemaining`** before either decrements it. Show a different sequence of events for that same code, whereby

starting with `seatsRemaining` being 2, two threads each dispense a ticket, but `seatsRemaining` is left as 1 rather than 0.

4.2 In the mutex-locking pseudocode of Figure 4.10 on page 87, there are two consecutive steps that remove the current thread from the runnable threads and then unlock the spinlock. Because spinlocks should be held as briefly as possible, we ought to consider whether these steps could be reversed. Explain why reversing them would be a bad idea by giving an example sequence of events where the reversed version malfunctions.

4.3 Show how to change queuing mutexes to correspond with POSIX's mutex-type `PTHREAD_MUTEX_RECURSIVE`. You may add additional components to each mutex beyond the state, waiters, and spinlock.

4.4 Explain why replacing `notifyAll` with `notify` is not safe in the `BoundedBuffer` class of Figure 4.17 on page 95. Give a concrete sequence of events under which the modified version would misbehave.

4.5 A semaphore can be used as a mutex. Does it correspond with the kind POSIX calls `PTHREAD_MUTEX_ERROR_CHECK`, `PTHREAD_MUTEX_NORMAL`, or `PTHREAD_MUTEX_RECURSIVE`? Justify your answer.

4.6 State licensing rules require a child-care center to have no more than three infants present for each adult. You could enforce this rule using a semaphore to track the remaining capacity, that is, the number of additional infants that may be accepted. Each time an infant is about to enter, a `down` operation is done first, with an `up` when the infant leaves. Each time an adult enters, you do three `up` operations, with three `down` operations before the adult may leave.

(a) Although this system will enforce the state rules, it can create a problem when two adults try to leave. Explain what can go wrong, with a concrete scenario illustrating the problem.

(b) The difficulty you identified in the preceding subproblem can be remedied by using a mutex as well as the semaphore. Show how.

(c) Alternatively, you could abandon semaphores entirely and use a monitor with one or more condition variables. Show how.

4.7 I illustrated deadlock detection using a transcript taken from an Oracle database (Figure 4.23, page 107). From that transcript you can tell that the locks are at the granularity of one per row, rather than one per table.

(a) What is the evidence for this assertion?

(b) Suppose the locking were done per table instead. Explain why no deadlock would have ensued.

(c) Even if locking were done per table, deadlock could still happen under other circumstances. Give an example.

4.8 Suppose you have two kinds of objects: threads and mutexes. Each locked mutex contains a reference to the thread that holds it as `mutex.owner`; if the mutex is unlocked, `mutex.owner` is `null`. Similarly, each thread that is blocked waiting for a mutex contains a reference to the mutex it is waiting for as `thread.blocker`; if the thread is not waiting for any mutex, `thread.blocker` is `null`. Suppose threads also contain a field, `thread.mark`, which is available for your use and is initialized to 0. Further, suppose you have an array of all the threads in the system as `threads[0]`, `threads[1]`, and so forth, up to `threads[threads.length-1]`. Write a pseudocode algorithm to test whether the system is deadlocked.

4.9 The main topic of this chapter (synchronization) is so closely related to the topics of Chapters 2 and 3 (threads and scheduling) that an author can hardly describe one without also describing the other two. For each of the following pairs of topics, give a brief explanation of why understanding the first topic in the pair is useful for gaining a full understanding of the second:
(a) threads, scheduling
(b) threads, synchronization
(c) scheduling, synchronization
(d) scheduling, threads
(e) synchronization, scheduling
(f) synchronization, threads

Programming Projects

4.1 Flesh out the `TicketVendor` class from Figure 4.6 on page 83 using Figure 4.5 on page 79 for guidance. Add a simple test program that uses a `TicketVendor` from multiple threads. Temporarily remove the `synchronized` keywords and demonstrate race conditions by inserting calls to the `Thread.sleep` method at appropriate points, so that incredibly lucky timing is not necessary. You should set up one demonstration for each race previously considered: two threads selling the last seat, two threads selling seats but the count only going down by 1, and an audit midtransaction. Now reinsert the `synchronized` keyword and show that the race bugs have been resolved, even with the `sleep`s in place.

4.2 Demonstrate races and mutual exclusion as in the previous project, but using a C program with POSIX threads and mutexes. Alternatively, use some other programming language of your choice, with its support for concurrency and mutual exclusion.

4.3 Choose some simplified version of a real-world process that evolves over time, such as a bouncing ball, an investment with compound interest, or populations

of predator and prey. Write a program with two threads. One thread should simulate the process you chose as time passes, possibly with some suitable scaling such as 1 second of simulator time per year of simulated time. The other thread should provide a user interface through which the user can modify the parameters of the ongoing simulation and can also pause and resume the simulation. Be sure to properly synchronize the two threads. Java would be an appropriate language for this project, but you could also use some other language with support for concurrency, synchronization, and user interfaces.

4.4 This project is identical to the previous one, except that instead of building a simulator for a real-world process, you should build a game of the kind where action continues whether or not the user makes a move.

4.5 Write a test program in Java for the `BoundedBuffer` class of Figure 4.17 on page 95.

4.6 Modify the `BoundedBuffer` class of Figure 4.17 (page 95) to call `notifyAll` only when inserting into an empty buffer or retrieving from a full buffer. Test that it still works.

4.7 Rewrite the `BoundedBuffer` class of Figure 4.17 (page 95) in C or C++ using the POSIX API. Use two condition variables, one for availability of space and one for availability of data.

4.8 Define a Java class for readers/writers locks, analogous to the `BoundedBuffer` class of Figure 4.17 (page 95). Allow additional readers to acquire a reader-held lock even if writers are waiting. As an alternative to Java, you may use another programming language with support for mutexes and condition variables.

4.9 Modify your readers/writers locks from the prior project so no additional readers may acquire a reader-held lock if writers are waiting.

4.10 Modify your readers/writers locks from either of the prior two projects to support an additional operation that a reader can use to upgrade its status to writer. (This is similar to dropping the read lock and acquiring a write lock, except that it is atomic: no other writer can sneak in and acquire the lock before the upgrading reader does.) What happens if two threads both hold the lock as readers, and each tries upgrading to become a writer? What do you think a good response would be to that situation?

4.11 Define a Java class for barriers, analogous to the `BoundedBuffer` class of Figure 4.17 (page 95). Alternatively, use another programming language, with support for mutexes and condition variables.

4.12 Define a Java class for semaphores to meet the needs of Figure 4.18 on page 100.

4.13 Rewrite the semaphore-based bounded buffer of Figure 4.18 (page 100) so that instead of using a `List`, it uses an array and a couple integer variables, just like the earlier version (Figure 4.17, page 95). Be sure to provide mutual exclusion for the portion of each method that operates on the array and the integer variables.

4.14 Translate the semaphore-based bounded buffer of Figure 4.18 (page 100) into C or C++ using the POSIX API's semaphores.

4.15 Translate the dining philosophers program of Exploration Project 4.2 into another language. For example, you could use C or C++ with POSIX threads and mutexes.

Exploration Projects

4.1 I illustrated pipes (as a form of bounded buffer) by piping the output from the `ls` command into the `tr` command. One disadvantage of this example is that there is no way to see that the two are run concurrently. For all you can tell, `ls` may be run to completion, with its output going into a temporary file, and then `tr` run afterward, with its input coming from that temporary file. Come up with an alternative demonstration of a pipeline, where it is apparent that the two commands are run concurrently because the first command does not immediately run to termination.

4.2 The Java program in Figure 4.26 simulates the dining philosophers problem, with one thread per philosopher. Each thread uses two nested `synchronized` statements to lock the two objects representing the forks to a philosopher's left and right. Each philosopher dines many times in rapid succession. In order to show whether the threads are still running, each thread prints out a message every 100000 times its philosopher dines.

(a) Try the program out. Depending on how fast your system is, you may need to change the number 100000. The program should initially print out messages, at a rate that is not overwhelmingly fast, but that keeps you aware the program is running. With any luck, after a while, the messages should stop entirely. This is your sign that the threads have deadlocked. What is your experience? Does the program deadlock on your system? Does it do so consistently if you run the program repeatedly? Document what you observed (including its variability) and the circumstances under which you observed it. If you have more than one system available that runs Java, you might want to compare them.

(b) You can guarantee the program won't deadlock by making one of the threads (such as number 0) acquire its right fork before its left fork. Explain why this

```
public class Philosopher extends Thread{

  private Object leftFork, rightFork;
  private int myNumber;

  public Philosopher(Object left, Object right, int number){
    leftFork = left;
    rightFork = right;
    myNumber = number;
  }

  public void run(){
    int timesDined = 0;
    while(true){
      synchronized(leftFork){
        synchronized(rightFork){
          timesDined++;
        }
      }
      if(timesDined % 100000 == 0)
        System.err.println("Thread " + myNumber + " is running.");
    }
  }

  public static void main(String[] args){
    final int PHILOSOPHERS = 5;
    Object[] forks = new Object[PHILOSOPHERS];
    for(int i = 0; i < PHILOSOPHERS; i++){
      forks[i] = new Object();
    }
    for(int i = 0; i < PHILOSOPHERS; i++){
      int next = (i+1) % PHILOSOPHERS;
      Philosopher p = new Philosopher(forks[i], forks[next], i);
      p.start();
    }
  }
}
```

Figure 4.26 Java program to simulate the dining philosophers.

prevents deadlock, and try it out. Does the program now continue printing messages as long as you let it run?

4.3 Search on the Internet for reported security vulnerabilities involving race conditions. How many can you find? How recent is the most recent report? Do you find any cases particularly similar to earlier ones?

Notes

The Therac-25's safety problems were summarized by Leveson and Turner [87]. Those problems went beyond the race bug at issue here, to also include sloppy software development methodology, a total reliance on software to the exclusion of hardware interlocks, and an inadequate mechanism for dealing with problem reports from the field.

When describing races, I spoke of threads' execution as being interleaved. In fact, unsynchronized programs may execute in even more bizarre ways than just interleavings. For example, one thread may see results from another thread out of order. For the Java programming language, considerable effort has gone into specifying exactly what reorderings of the threads' execution steps are legal. However, the bottom line for programmers is still that synchronization should be used to avoid races in the first place; trying to understand the race behavior is a losing battle.

Recall that my brief descriptions of the POSIX and Java APIs are no replacement for the official documentation on the web at *http://www.opengroup.org* and *http://java.sun.com*, respectively. In particular, I claimed that each Java mutex could only be associated with a single condition variable, unlike in the POSIX API. Actually, version 1.5 of the Java API gained a second form of mutexes and condition variables, contained in the `java.util.concurrent` package. These new mechanisms are not as well integrated with the Java programming language as the ones I described, but do have the feature of allowing multiple condition variables per mutex.

My spinlocks depend on an atomic exchange instruction. I mentioned that one could also use some other atomic read-and-update instruction, such as atomic increment. In fact, in 1965 Dijkstra [45] showed that mutual exclusion is also possible using only ordinary load and store instructions. However, this approach is complex and not practical; by 1972, Dijkstra [48] was calling it "only of historical interest."

Semaphores were proposed by Dijkstra in a privately circulated 1965 manuscript [46]; he formally published the work in 1968 [47]. Note, however, that Dijkstra credits Scholten with having shown the usefulness of semaphores that go beyond 0 and 1. Presumably this includes the semaphore solution to the bounded buffer problem, which Dijkstra presents.

The idea of using a consistent ordering to prevent deadlocks was published by Havender, also in 1968 [65]. Note that his title refers to "avoiding deadlock." This is potentially confusing, as today *deadlock avoidance* means something different than *deadlock prevention*. Havender describes what is today called deadlock prevention. Deadlock avoidance is a less practical approach, dating at least to Dijkstra's work in 1965 and fleshed out by Habermann in 1971 [62]. (Remarkably, Habermann's title speaks of "prevention" of deadlocks—so terminology has completely flip-flopped since the seminal papers.) I do not present deadlock avoidance in this textbook. Havender also described other approaches to preventing deadlock; ordering is simply his "Approach 1." The

best of his other three approaches is "Approach 2," which calls for obtaining all necessary resources at the same time, rather than one by one. Coffman, Elphick, and Shoshani [31] published a survey of deadlock issues in 1971, which made the contemporary distinction between deadlock prevention and deadlock avoidance.

In 1971, Courtois, Heymans, and Parnas [35] described both variants of the readers/writers locks that the programming projects call for. (In one, readers take precedence over waiting writers, whereas in the other waiting writers take precedence.) They also point out that neither of these two versions prevents starvation: the only question is which class of threads can starve the other.

Resource allocation graphs were introduced by Holt in the early 1970s; the most accessible publication is number [71]. Holt also considered more sophisticated cases than I presented, such as resources for which multiple units are available, and resources that are produced and consumed rather than merely being acquired and released.

Monitors and condition variables apparently were in the air in the early 1970s. Although the clearest exposition is by Hoare in 1974 [69], similar ideas were also proposed by Brinch Hansen [23] and by Dijkstra [48], both in 1972. Brinch Hansen also designed the monitor-based programming language Concurrent Pascal, for which he later wrote a history [24].

My example of deadlock prevention in the Linux kernel was extracted from the file `kernel/sched.c` in version 2.6.0-test9.

The use of priority inheritance to limit priority inversion was explained by Sha, Rajkumar, and Lehoczky [116]. They also presented an alternative solution to the priority inversion problem, known as the priority ceiling protocol. The priority ceiling protocol sometimes forces a thread to wait before acquiring a mutex, even though the mutex is available. In return for that extra waiting, it guarantees that a high-priority thread will have to loan its priority to at most one lower-priority thread to free up a needed mutex. This allows the designer of a real-time system to calculate a tighter bound on each task's worst-case execution time. Also, the priority ceiling protocol provides a form of deadlock avoidance.

The convoy phenomenon, and its solution, were described by Blasgen et al. [21].

Dijkstra introduced the dining philosophers problem in reference [48]. He presented a more sophisticated solution that not only prevented deadlock but also ensured that each hungry philosopher got a turn to eat, without the neighboring philosophers taking multiple turns first.

The TOCTTOU race vulnerability in Sun's mail delivery software was reported in 1992 by a group known as [8lgm]. Their web site, *http://www.8lgm.org*, may or may not still be around when you read this, but you should be able to find a copy of the advisory somewhere on the web by searching for [8lgm]-Advisory-5.UNIX.mail.24-Jan-1992.