CHAPTER 7
Threading

Threading is the creation and management of multiple units of execution within a single process. Threading is a significant source of programming error, through the introduction of data races and deadlocks. The topic of threading can—and indeed does—fill whole books. Those works tend to focus on the myriad interfaces in a particular threading library. While we will cover the basics of the Linux threading API, the goal of this chapter is to go meta: How does threading fit into the system programmer's overall toolkit? Why use threads—and, more importantly, why not? What design patterns help us conceptualize and build threading applications? And, finally, what are data races and how can we prevent them?

Binaries, Processes, and Threads

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion. *Processes* are the operating system abstraction representing those binaries in action: the loaded binary, virtualized memory, kernel resources such as open files, an associated user, and so on. *Threads* are the unit of execution within a process: a virtualized processor, a stack, and program state. Put another way, processes are running binaries and threads are the smallest unit of execution schedulable by an operating system's process scheduler.

A process *contains* one or more threads. If a process contains but one thread, there is only a single unit of execution in the process and only one thing going on at a time. We call such processes *single threaded*. They are the classic Unix process. If a process contains more than one thread, then there is more than one thing going on at once. We call such processes *multithreaded*.

Modern operating systems provide two fundamental virtualized abstractions to user-space: virtual memory and a virtualized processor. Together, they give the illusion to

each running process that it alone consumes the machine's resources. Virtualized memory affords each process a unique view of memory that seamlessly maps back to physical RAM or on-disk storage (via paging). The system's RAM may in actuality contain the data of 100 different running processes, but each process sees virtual memory all of its own. A virtualized processor lets processes act as if they alone run on the system, with the operating system hiding the fact that multiple processes are multitasking across (perhaps) multiple processors.

Virtualized memory is associated with the process and not the thread. Thus, each process has a unique view of memory, but all of the threads in a given process *share* that memory. Conversely, a virtualized processor is associated with threads and not processes. Each thread is an independently schedulable entity, allowing a single process to "do" more than one thing at a time. Many programmers combine the two illusions of virtualized memory and virtualized processor, but threads require you to separate them. Threads have the illusion, as processes do, of having a processor (or several) all to themselves. Threads, unlike processes, do not have the illusion of having memory all to themselves—all the threads within a process share the entirety of their memory address space.

Multithreading

What is the point of threads? We obviously need processes, since they are the abstraction of a running program. But why decouple the unit of execution and introduce threads? There are six primary benefits to multithreading:

Programming abstraction
: Dividing up work and assigning each division to a unit of execution (a thread) is a natural approach to many problems. Design patterns that utilize this approach include the thread-per-connection and thread pool patterns. Programmers find these patterns useful and intuitive. Some, however, view threads as an anti-pattern. The inimitable Alan Cox summed this up well with the quote, "threads are for people who can't program state machines." That is, there is in theory no programming problem that is solvable with threads that isn't solvable with a state machine.

Parallelism
: In machines with multiple processors, threads provide an efficient way to achieve *true parallelism*. As each thread receives its own virtualized processor and is an independently schedulable entity, multiple threads may run on multiple processors at the same time, improving a system's throughput. To the extent that threads are used to achieve parallelism—that is, there are no more threads than processors—the "threads are for people who can't program state machines" maxim does not apply.

Improving responsiveness
: Even on a uniprocessor machine, multithreading can improve a process's responsiveness. In a single-threaded process, a long-running operation can prevent an application from responding to user input, making it appear as if the application has froze. With multithreading, such operations may be delegated to worker threads, allowing at least one thread to remain responsive to user input and perform UI operations.

Blocking I/O
: This is related to the previous item. Without threads, blocking I/O halts the entire process. This can be detrimental to both throughput and latency. In a multithreaded process, individual threads may block, waiting on I/O, while other threads continue to make forward progress. Asynchronous and nonblocking I/O are alternative solutions to threads for this issue.

Context switching
: The cost of switching from one thread to a different thread within the same process is significantly cheaper than process-to-process context switching.

Memory savings
: Threads provide an efficient way to share memory yet utilize multiple units of execution. In this manner they are an alternative to multiple processes.

For these reasons, threading is a relatively common feature of operating systems and their applications. On some systems, such as Android, nearly every process on the system sports multiple threads. A decade or two ago, "threads are for people who can't program state machines" was quite often true, as most of the benefits of threads were realizable through other means, such as nonblocking I/O and, yes, state machines. Today, the number of processors in even the smallest of machines—even mobile devices have multiple processors—and technologies such as multicore and simultaneous multithreading (SMT) necessitate threads as a tool to maximize throughput in system programming. Now it is unimaginable to find a high-performance web service *not* running with many threads on many cores.

Context Switching: Processes Versus Threads

One of the performance wins from threading comes from the inexpensive cost of context switching from thread-to-thread within the same process (*intraprocess switching*). On any system, the cost of intraprocess switching is less than the cost of interprocess switching; the former is always a subset of the latter. This cost gap is particularly large on systems other than Linux, where processes are costly abstractions. Hence many systems call threads "lightweight processes."

On Linux, the cost of interprocess switching is not high, but intraprocess switching is near zero: approximately the cost of entering and exiting the kernel. Processes aren't expensive, but threads are cheaper still.

Machine architectures impose costs to process switching that threads do not bear, as process switching involves swapping out one virtual address space for another. On x86, for example, the translation lookaside buffer (TLB), which is a cache mapping virtual to physical memory addresses, must be flushed when swapping out the virtual address space. In certain workloads, TLB misses are incredibly detrimental to system performance. As an extreme example, on some ARM machines, the entirety of the CPU cache must be flushed! Threads do not bear these costs, as thread-to-thread switching does not swap out the virtual address space.

Costs of Multithreading

Despite these benefits, multithreading is not without cost. Indeed, some of the scariest, most nefarious bugs in the history of programming have been caused by threading. Designing, writing, understanding, and—most treacherous of all—debugging multithreading programs is significantly more difficult than with a single-threaded process.

The source of consternation with threads is also their raison d'être: multiple virtualized processors but only one instance of virtualized memory. Put another way, a multithreaded process has multiple things going on at once (*concurrency*) yet all those things share the same memory. Inevitably, the threads in a process are going to share resources—say, the need to read from or write to the same data. Understanding how your program works thus changes from understanding the simple sequential execution of instructions to conceptualizing multiple threads, executing independently, with timing and ordering that can be unpredictable yet absolutely essential to correct operation. Failing to *synchronize* threads can lead to corrupt output, incorrect execution, and program crash. Because understanding and debugging multithreaded programs is so difficult, it is imperative that your threading model and synchronization strategy be part of your system's design from day one.

Alternatives to Multithreading

Depending on your goals with multithreading, there are alternatives. For example, the latency and I/O benefits to threading are attainable via a combination of multiplexed I/O (see "Multiplexed I/O" on page 51), nonblocking I/O ("Nonblocking Reads" on page 35), and asynchronous I/O ("Asynchronous I/O" on page 123). These techniques allow processes to issue I/O operations that do not block the process. If true parallelism is your goal, *N* processes can achieve the same processor utilization as *N* threads, albeit at some cost of increased resource consumption and context switching overhead. Conversely, if memory savings is your goal, Linux provides tools to share memory in a more limited manner than threads.

Contemporary system programmers tend to not find these alternatives compelling. Asynchronous I/O, for example, is often infuriating. And even if you can mitigate the cost of multiple processes via shared memory and other shared resources, the context switching overhead is not going anywhere. Thus, threads are common not just in system programming, but across the stack: from the kernel up through to GUI applications. With the increasing prevalence of multiple cores, the use of threads will only increase.

Threading Models

There are several approaches to implementing threads on a system, with varying degrees of functionality provided by the kernel and user space. The simplest model is realized when the kernel provides native support for threads, and each of those kernel threads translates directly to the user-space concept of a thread. Such a model is called *1:1 threading*, as there is a one-to-one relationship between what the kernel provides and what the user consumes. This model is also known as *kernel-level threading*, as the kernel is the core of the system's threading model.

Threading in Linux, which we will discuss in "Linux Threading Implementations" on page 226, is *1:1*. The Linux kernel implements threads simply as processes that share resources. The threading library creates a new thread via the `clone()` system call and the returned "process" is directly managed as the user-space concept of a thread. That is, on Linux, what user space calls a thread is pretty much what the kernel calls a thread, too.

User-Level Threading

The complete opposite model is *N:1 threading*, also called *user-level threading*. Contra kernel-level threading, in this model user space is the key to the system's threading support, as it implements the concept of a thread. A process with *N* threads will map to a single kernel process—hence *N:1*. This model requires little or no kernel support but significant user-space code, including a user-space scheduler to manage the threads and a mechanism to catch and handle I/O in a nonblocking fashion. The benefit of user-level threads is that context switches are nearly free, as the application itself can decide what thread to run and when, without involving the kernel. The downside is that, as there is only a single kernel entity backing the *N* threads, this model cannot utilize multiple processors and thus is unable to provide true parallelism. On modern hardware, this is a significant downside, particularly given that the benefit of reduced-cost context switching is of marginal value on Linux, which sports cheap context switching.

While there are user-level threading libraries for Linux, most libraries provide *1:1* threading, which is what we will discuss later in this chapter.

Hybrid Threading

What if we combine kernel- and user-level threading? Is it possible to achieve the true parallelism of the *1:1* model with the free context switches of the *N:1* model? Indeed it is, if you are willing to accept quite a bit of complexity. *N:M threading*, also known as *hybrid threading*, attempts to achieve the best of both worlds: the kernel provides a native thread concept, while user space also implements user threads. User space, perhaps in conjunction with the kernel, then decides how to map *N* user threads onto *M* kernel threads, where *N*>=*M*.

Approaches differ by implementation, but the typical strategy is to not back most of the user threads with a kernel thread. A process might contain hundreds of user threads but only a small number of kernel threads, with that small number a function of processors (with at least one kernel thread for each processor enabling the full utilization of the system) and blocking I/O. As you might imagine, this model is rather complex to implement. Given Linux's cheap context switches, most system developers do not feel this approach worthwhile and the *1:1* model remains popular for Linux.

Scheduler Activations is a solution that provides kernel support for user-level threads, enabling more performant execution of *N:M* threading. It began as an academic paper out of the University of Washington and was later adopted by both FreeBSD and NetBSD, becoming the core of their threading implementation. Scheduler Activations allow user-space control of and insight into the kernel's process scheduling, which makes the hybrid model more efficient and fixes several issues that can crop up in an implementation unassisted by the kernel.

Both FreeBSD and NetBSD have abandoned Scheduler Activations in preference for simpler *1:1* threading. You can view this as both a rejection of the complexity of the *N:M* model and a response to the ubiquity of the x86 architecture, which allows for relatively efficient context switches.

Coroutines and Fibers

Coroutines and *fibers* provide a unit of execution even lighter in weight than the thread (with the former being their name when they are a programming language construct, and the latter when they are a system construct). They are, like user-level threads, user-space phenomena but unlike user-level threads, there is little or no user-space support for their scheduling and execution. Instead, they are cooperatively scheduled, requiring an explicit *yield* in one to move to another. Coroutines and fibers are only subtly different from subroutines (normal C/C++ functions). Indeed, you can look at subroutines as a special case of coroutines. Coroutines and fibers are more about control of program flow than concurrency.

Linux does not natively support coroutines or fibers, again likely due to its already-fast context switch speed obviating the need for a construct with superior performance to the kernel thread. The Go programming language provides language-level support for coroutine-like constructs in Linux, called *Go-routines*. Coroutines enable differing programming paradigms and I/O models and, although beyond the scope of this book, are worth consideration.

Threading Patterns

The first and most important step in building a threaded application is deciding on a threading pattern, which will also be the processing and I/O model for your application. There are a myriad of abstractions and implementation details to settle on, but the two core programming patterns, of which you must pick one, are *thread-per-connection* and *event-driven*.

Thread-per-Connection

Thread-per-connection is a programming pattern in which a unit of work is assigned to one thread, and that thread is assigned at most one unit of work, for the duration of the unit of work's execution. The unit of work is however you break down your application's work: a request, a connection, etc. For this discussion, we'll say "connection" as that is the common terminology in describing this pattern.

Another way of describing this pattern is "run until completion." A thread grabs a connection or a request and processes it until done, at which time the thread is available to process another request anew. This has interesting implications for I/O and, indeed, I/O is one of the large differences between this and the event-driven pattern. In thread-per-connection, blocking I/O—indeed, any I/O—is permissible as the connection "owns" the thread. Blocking a thread can stall only the connection causing the blocking. In this manner, the thread-per-connection pattern uses the kernel to handle the scheduling of work and the management of I/O.

In this pattern, the number of threads is an implementation detail. We've discussed thread-per-connection thus far as if there is always a thread for every unit of work. That can be true, but most implementations like to put a bound on the number of threads they create. When the number of in-flight connections (and thus the number of threads) reaches a limit, connections are either queued or rejected until the number of in-flight connections drops below the limit.

Note there is nothing about this pattern that requires threading. Indeed, replace "thread" with "process" and you are describing the old-school Unix server. Apache's standard "fork" model, for example, follows this pattern. This is also the typical pattern for I/O in Java, although preferences are changing.

Event-Driven Threading

The event-driven pattern is a rejoinder to the thread-per-connection pattern. Consider the web server. In terms of computing power, modern hardware is capable of handling significant numbers of requests at once. In the thread-per-connection pattern, that is a lot of threads. Threads have fixed costs, most notably requiring both a kernel and user-space stack. These fixed costs place scalability limits on the number of threads in a given process, particularly on 32-bit systems. (The arguments against thread-per-connection are less relevant to 64-bit systems, but sufficiently valid that event-driven is still considered a superior choice even for 64-bit systems.) Systems might have the computing resources to handle several thousand in-flight connections, yet hit scalability limits in running that many concurrent threads.

In seeking an alternative, system designers noticed that most of the threads are doing a lot of waiting: reading files, waiting for databases to return results, issuing remote procedure calls. Indeed, recall our discussion from "Multithreading" on page 212: using more threads than you have processors on the system does not provide any benefits to parallelism. Instead, such uses of threads reflect a programming abstraction, an ease of programming that can be replicated through a more formal flow of control model.

These observations begot *event-driven threading*. Since so much of many thread-per-connection workloads is simply waiting, let's decouple that waiting from threads. Instead, issue all I/O asynchronously (see "Asynchronous I/O" on page 123) and use multiplexed I/O (see "Multiplexed I/O" on page 51) to manage the flow of control in the server. In this model, request processing is converted into a series of asynchronous I/O requests and associated callbacks. These callbacks can be waited on via multiplexed I/O; the process of doing so is called an *event loop*. When the I/O requests are returned, the event loop hands the callback off to a waiting thread.

As with the thread-per-connection pattern, nothing about the event-driven pattern need be threaded. Indeed, the event loop could simply be the fall-through when a single-threaded process is done executing a callback. Threads need only be added to provide true parallelism. In this model, there is no reason to have more threads than processors.

Patterns ebb and flow in popularity, but the event-driven pattern is currently the preferred approach to designing a multithreaded server. Several popular alternatives to Apache, for example, have developed over the last few years, all of which are event-driven. In designing your threaded system software, I suggest you first consider the event-driven pattern: asynchronous I/O, callbacks, an event loop, and a small thread pool with just a thread per processor.

Concurrency, Parallelism, and Races

Threads create two related but distinct phenomena: concurrency and parallelism. Both are bittersweet, touching on the costs of threading as well as its benefits. *Concurrency*

is the ability of two or more threads to execute in overlapping time periods. *Parallelism* is the ability to execute two or more threads simultaneously. Concurrency can occur without parallelism: for example, multitasking on a single processor system. Parallelism (sometimes emphasized as *true parallelism*) is a specific form of concurrency requiring multiple processors (or a single processor capable of multiple engines of execution, such as a GPU). With concurrency, multiple threads make forward progress, but not necessarily simultaneously. With parallelism, threads literally execute in parallel, allowing multithreaded programs to utilize multiple processors.

Concurrency is a programming pattern, a way of approaching problems. Parallelism is a hardware feature, achievable through concurrency. Both are useful.

Race Conditions

It is concurrency that introduces most of the hardships of threading. By enabling overlapping execution, threads can execute in an unpredictable order with respect to each other. At times, this is fine. But what if threads need to share a resource? Accessing even something as simple as a word of memory becomes a "race," where program behavior differs depending on which thread "gets there first."

Formally, a *race condition* is a situation in which the unsynchronized access of a shared resource by two or more threads leads to erroneous program behavior.[1] The shared resource can be anything: the system's hardware, a kernel resource, or data in memory. The latter is the most common form and is called a *data race*. The window in which a race can occur—the region of code which should be synchronized—is called a *critical region*. Races are eliminated by *synchronizing* threads' access to critical regions. Before diving into methods of synchronization, let's discuss a couple example race conditions.

Real-world races

Imagine an automated teller machine (ATM), also called an automated banking machine (ABM) or a cash machine. Usage is simple: you walk up, swipe your card, enter your PIN, and enter a withdrawal amount. You then collect your money. Somewhere in there, the bank needs to verify that you actually have the funds in your account and, if so, deduct the amount of withdrawal. The algorithm looks something like this:

1. Does the account hold at least X units of currency?
2. If so, deduct X from the account's balance and disburse X to the user.
3. If not, return error.

1. A *benign race* occurs when unsynchronized access leads to unpredictable but not erroneous program behavior. Occasionally programmers decide not to synchronize shared data that isn't critical, such as statistical counters, in pursuit of performance. I recommend always synchronizing shared data.

The code in C might look something like this:

```
int withdraw (struct account *account, int amount)
{
        const int balance = account->balance;
        if (balance < amount)
                return -1;
        account->balance = balance - amount;

        disburse_money (amount);

        return 0;
}
```

There's a disastrous race in here if concurrent execution is possible. Imagine if the bank is executing this function twice, concurrently. Perhaps the customer is withdrawing funds at the same moment the bank is processing an online bill payment or assessing an egregious fee. What if the funds checks both happen at about the same time, before the account balance is updated and the money is dispersed? Both disbursements can happen, even if the account's balance is insufficient to handle both! For example, if there were $500 in the account and two withdraw requests came in for $200 and $400, they could both succeed, even though that would leave the account $100 in the red, which isn't something the code as written was intended to permit.

In fact, there is a second race in this function. Consider the store of the updated balance into the `account` structure. The two withdrawals can also race on updating the balance. Using our previous example, we'd end up with either $300 or $100 stored as the balance. Not only did this bank allow withdrawals to execute that should not, but it gifted this lucky customer with up to an extra $400.

Indeed, nearly every line of this function is inside a critical region. For this bank to survive as a viable business, it needs to synchronize access to the `withdraw()` function, ensuring that even if it is executed concurrently by two or more threads, the entire function is executed as one atomic unit: the bank needs to load the account balance, check for available funds, and debit the balance in one indivisible transaction.

Before we look at how this bank can do that, let's consider an example that shows just how fundamental race conditions are. This bank withdrawal example was rather high level: indeed, we didn't even need to show the example code. A bank executive can understand that if you allow accountants to credit and debit accounts concurrently, the math can get screwed up. But race conditions exist at the most fundamental of levels, too.

Consider this very simple line of C code:

```
x++;  // x is an integer
```

This is the *post-increment* operator and we all know what it does: it takes the current value of `x`, increments it by one, and stores that new value back in `x`, with the value of

the expression being the updated `x`. How this compiles to machine code depends, of course, on the architecture, but we can imagine it might look something like this:

```
load x into register
add 1 to register
store register in x
```

Yes, even `x++` is a racy operation. Imagine two threads executing `x++` concurrently with `x=5`. Here is a desirable outcome:

Time	Thread 1	Thread 2
1	load x into register (5)	
2	add 1 to register (6)	
3	store register in x (6)	
4		load x into register (6)
5		add 1 to register (7)
6		store register in x (7)

This, too, is desirable:

Time	Thread 1	Thread 2
1		load x into register (5)
2		add 1 to register (6)
3		store register in x (6)
4	load x into register (6)	
5	add 1 to register (7)	
6	store register in x (7)	

We are lucky if that is the outcome. Nothing prevents this:

Time	Thread 1	Thread 2
1	load x into register (5)	
2	add 1 to register (6)	
3		load x into register (5)
4	store register in x (6)	
5		add 1 to register (6)
6		store register in x (6)

Many other combinations can also lead to unintended results. These examples exhibit concurrency but not parallelism. With parallelism, the threads can execute at the same time, adding even more combinations of peril:

Time	Thread 1	Thread 2
1	load x into register (5)	load x into register (5)
2	add 1 to register (6)	add 1 to register (6)
3	store register in x (6)	store register in x (6)

You get the picture. Even something as simple as adding one to a variable—a single line of C or C++—is fraught with races once we have multiple threads executing concurrently. We don't even need parallelism. A single processor machine can—and probably will—suffer these races. Race conditions are among the largest sources of programmer frustration and program bugs. Let's look at how programmers handle them.

Synchronization

The fundamental source of races is that critical regions are a window during which correct program behavior requires that threads do not interleave execution. To prevent race conditions, then, the programmer needs to synchronize access to that window, ensuring *mutually exclusive* access to the critical region.

In computer science, we say that an operation (or set of operations) is *atomic* if it is indivisible, unable to be interleaved with other operations. To the rest of the system, an atomic operation (or operations) appears to occur *instantaneously*. And that's the problem with critical regions: they are not indivisible, they don't occur instantaneously, they aren't atomic.

Mutexes

There are many techniques for making critical regions atomic, from solutions for single instructions up through large blocks of code. The most common technique is the *lock*, a mechanism for ensuring mutual exclusion within a critical region, rendering it atomic. Because locks enforce mutual exclusion, they are known in Pthreads (and elsewhere) as *mutexes*.[2]

A lock works similarly to its real-world namesake: imagine that a room is a critical region. Without a lock, people (threads) can come and go from the room (critical region) as they please. Specifically, there can be more than one person in the room at a time. So we put a door on the room and a lock on the door. We distribute but a single key to that door. When a person (a thread) comes to the door, they find the key sitting outside. They use the key to open the door, they go inside, and then they lock the door from the inside. No one else may enter. They can then go about their business in the room without interruption. No one else may occupy the room concurrently; it is a mutually exclusive

2. Which in turn are also known as *binary semaphores*.

resource. When the person is done with the room, they unlock the door and exit, leaving the key outside. The next person may then enter, lock the door behind them, and repeat.

A lock in the threading context works much the same. The programmer defines the lock and makes sure to acquire it before entering the critical region. The lock implementation ensures that only one thread can "hold" the lock at once. If it is in use by another thread, a new thread must wait for it before continuing. When done with a critical region, you release the lock, letting a waiting thread (if any) acquire the lock and proceed.

Recall from "Race Conditions" on page 219 our bank withdrawal example. Let's look at how a mutex would prevent the disastrous (at least for the bank) race condition we studied. We will discuss the actual mutex operations provided by Pthreads later (see "Pthread Mutexes" on page 235), but for now let's assume we have the functions `lock()` and `unlock()` to acquire and release, respectively, a mutex.

```
int withdraw (struct account *account, int amount)
{
        lock ();
        const int balance = account->balance;
        if (balance < amount) {
                unlock ();
                return -1;
        }
        account->balance = balance - amount;
        unlock ();

        disburse_money (amount);

        return 0;
}
```

We are locking only the part of the function that can race: the reading of the account balance, the sufficient funds check, and the updating of the account balance. Once the program knows it has a valid transaction and has updated the account balance, it can drop the lock and thus disburse the funds without enforcing mutual exclusion. The smaller you make the critical region, the better, as locks prevent concurrency and thus negate the benefits of threading.

Note there is nothing magic about locks. Nothing physically enforces the mutual exclusion. Locks are akin to a *gentlemen's agreement*. All threads must acquire the right locks in the right places. Nothing prevents a thread from not acquiring a necessary lock except conscientious programming.

Lock Data, Not Code

One of the most important programming patterns in multithreaded programming is *lock data, not code*. Although we have framed the discussion about race conditions around critical regions, a good programmer does not view code as the object of locking. You never say, "this lock protects this function." Instead, a good programmer associates data with locks. Shared data has an associated lock and accessing that data always requires that the associated lock be held.

What's the difference? When you associate locks with code, the locking semantics are harder to understand. Over time, the relationship between the lock and the data can grow unclear, and programmers may introduce new uses of the data without the appropriate lock. By associating locks with data, you help keep that mapping clear.

Deadlocks

The cruel irony of threads is the chain of desires begetting pain, which prompt solutions that only beget more pain. We want threads for concurrency, but that concurrency introduces race conditions. So we introduce mutexes, but those mutexes introduce a new source of programming bug: the deadlock.

A *deadlock* is a situation in which two threads are waiting for the other to finish, and thus neither does. In the case of mutexes, a deadlock occurs where two threads are each waiting for a different mutex, which the other thread holds. A degenerate case is when a single thread is blocked, waiting for a mutex that it already holds. Debugging deadlocks is often tricky, as your program needn't crash. Instead, it simply stops making forward progress, as ever more of your threads wait for a day that will never come.

Multithreading Mishaps on Mars

There are many real-life tales of threading woes, but one of the more intriguing is that of Mars Pathfinder, which in July of 1997 had successfully arrived on the Martian surface, only to find its mission of analyzing Martian climate and geology interrupted by frequent system resets.

Mars Pathfinder was powered by a real-time, highly threaded, embedded kernel (not Linux). The kernel provided preemptive scheduling of threads. Like Linux, real-time threads have priorities, and a thread of a given priority would always run before lower priority threads. There were, among many other threads, three that are pertinent to this bug: a low-priority thread to gather meteorological data, a medium-priority thread to communicate with Earth, and a high-priority thread to manage storage throughout the rover. As seen earlier in this chapter (see "Race Conditions" on page 219), synchronization is critical to preventing data races, and thus the threads managed concurrency

via mutexes. Notably, a mutex synchronized the low-priority meteorological thread (which was generating data) against the high-priority storage thread (which was managing that data).

The meteorological thread ran infrequently, polling the various sensors on the spacecraft. The thread would then acquire the mutex, write the meteorological data to the storage subsystem, and finally release the mutex. The storage thread ran more often, responding to system events. Before managing the storage subsystem, it too would acquire the mutex. If it were unavailable, it would sleep until the meteorological thread released it.

So far, so good. Occasionally, however, the communication thread would wake up and run while the meteorological thread was holding and the storage thread was waiting for the mutex. As the communication thread was a higher priority than the meteorological thread, the former ran at the expense of the latter. Unfortunately, the communication thread was a long-running task: Mars is far away! Thus, for the entirety of the communication thread's operation, the meteorological thread was not run. This seems by design, as dictated by the given priorities. But the meteorological thread held a resource (the mutex) that the storage thread wanted. Consequently, a lower priority thread (communication) was indirectly running at the expense of a higher priority thread (storage). The system would eventually notice the storage thread wasn't making forward progress, determine there must be an issue, and perform a system reset. This is a classic example of a class of bugs known as *priority inversion*.

The fix is a technique known as *priority inheritance*, where the process holding a resource inherits the priority of the highest priority process waiting for that resource. In this case, the lower priority meteorological thread would have inherited the higher priority of the storage thread for the duration it held the mutex. This would have prevented the communication thread from preempting the meteorological thread, allowing the quick release of the mutex and scheduling of the storage thread. If you don't find this a cautionary tale, stick to single-threaded programming!

Deadlock avoidance

Avoiding deadlocks is important, and the only consistent, safe way to do so is by designing locking into your multithreaded program from day one. It is important to associate mutexes with data, not code, and have a clear hierarchy of data (and thus mutexes). For example, a simple form of deadlock is known as the *ABBA deadlock* or the *deadly embrace*. This occurs when one thread acquires mutex *A* followed by mutex *B*, while another thread acquires mutex *B* followed by *A* (hence *ABBA*). With the right timing, both threads can successfully acquire their first mutex: Thread 1 holds *A* and Thread 2 holds *B*. When they then go to acquire the other mutex, they find it taken by the other thread, and each block, waiting for the mutex's release. Because each thread that holds a mutex is also waiting for a mutex, neither is ever released, and the threads deadlock.

Fixing this requires clear rules: Mutex *A* must always be acquired before *B*. As the complexity of your program, and thus of its synchronization, grows, it only becomes more difficult to enforce these rules. Start early and design things cleanly.

Pthreads

The Linux kernel provides only the underlying primitives that enable threading, such as the `clone()` system call. The bulk of any threading library is in user space. Many large software projects define their own threading library: Android, Apache, GNOME, and Mozilla all provide their own threading library, for example, and languages such as C++11 and Java provide standard library support for threads. Nonetheless, POSIX standardized a threading library with IEEE Std 1003.1c-1995, also known as POSIX 1995 or POSIX.1c. Developers call this standard *POSIX threads* or, for short, *Pthreads*. Pthreads remains the predominant threading solution for both C and C++ on Unix systems.

Linux Threading Implementations

Pthreads, as a standard, is just a bunch of words on a page. In Linux, the implementation of that standard is provided by *glibc*, Linux's C library. Over time, *glibc* has provided two different implementations of Pthreads: LinuxThreads and NPTL.

LinuxThreads is Linux's original Pthread implementation, providing *1:1* threading. It was first included in *glibc* with version 2.0, although it was available as an external library prior. LinuxThreads was designed for a kernel that provided very little support for threading: other than the `clone()` system call to create a new thread, LinuxThreads implemented POSIX threading using existing Unix interfaces. For example, LinuxThreads handles thread-to-thread communication using signals (see Chapter 10). Due to the lack of kernel support for Pthreads, LinuxThreads required a "manager" thread to coordinate activity, scaled poorly to large numbers of threads, and was imperfect in its conformance to the POSIX standard.

Native POSIX Thread Library (*NPTL*) superseded LinuxThreads and remains the standard Linux Pthread implementation. It was introduced in Linux 2.6 and *glibc* 2.3. Like LinuxThreads, NPTL provides *1:1* threading based around the `clone()` system call and the kernel's model that threads are just like any other process, except they share certain resources. Unlike LinuxThreads, NPTL capitalizes on additional kernel interfaces new to the 2.6 kernel, including the `futex()` system call for thread synchronization, the `exit_group()` system call for terminating all the threads in a process, and kernel support for thread-local storage (TLS). NPTL resolves LinuxThreads' nonconformance issues and vastly improves threading scalability, allowing for the creation of thousands of threads in a single process without slowdown.

NGPT

A competitor and early alternative to NPTL was *Next Generation POSIX Threads* (*NGPT*). Like NPTL, NGPT attempted to overcome the limitations of LinuxThreads and improve scalability. Unlike NPTL and LinuxThreads, however, NGPT implemented *N:M* threading. As is often the case in Linux, the simpler solution won out and NGPT is but a sidebar in history.

Although LinuxThreads-based systems are growing a bit long in the tooth, they can still be found. As NPTL is such a significant improvement over LinuxThreads, strongly consider upgrading such systems to NPTL (as if you needed another reason to not use such an ancient system) or, failing that, sticking to single-threaded programming.

The Pthread API

The Pthread API defines everything needed—albeit at a rather low level—to build a multithreaded program. Providing over 100 interfaces, the Pthread API is large. Due to its size and ungainliness, Pthreads is not without detractors. Nonetheless, it is *the* core threading library on Unix systems and is worth learning even if you use a different threading solution, as most are built on top of Pthreads.

The Pthread API is defined in `<pthread.h>`. Every function in the API is prefixed by `pthread_`. For example, the function to create a thread is called `pthread_create()` (we will study it soon, in "Creating Threads" on page 228). Pthread functions may be broken into two large groupings:

Thread management
: Functions to create, destroy, join, and detach threads. We will cover all of these in this chapter.

Synchronization
: Functions to manage the synchronization of threads, including mutexes, condition variables, and barriers. We will cover mutexes in this chapter.

Linking Pthreads

Although Pthreads is provided by *glibc*, it is in a separate library, *libpthread*, and thus requires explicit linkage. With *gcc*, this is automated by the *-pthread* flag, which ensures the proper library is linked into your executable:

```
gcc -Wall -Werror -pthread beard.c -o beard
```

If you build and link your binary with multiple invocations to *gcc* you'll want to provide `-pthread` to all of them: the flag also effects the preprocessor, by setting certain preprocessor defines that control thread safety.

Creating Threads

When your program is first run and executes the `main()` function, it is single threaded. Indeed, other than the compiler enabling some thread safety options and the linker linking in the Pthreads library, your process isn't any different from any other. From this initial thread, sometimes called the *default* or *master thread*, you must create one or more additional threads to become multithreaded.

Pthreads provides a single function to define and launch a new thread, `pthread_create()`:

```
#include <pthread.h>

int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg);
```

Upon successful invocation, a new thread is created and begins executing the function provided by `start_routine`, passed the sole argument `arg`. The function will store the thread ID, used to represent the new thread, in the `pthread_t` pointed at by `thread`, if it is not `NULL` (we will discuss thread IDs in the next section, "Thread IDs" on page 229).

The `pthread_attr_t` object pointed at by `attr` is used to change the default *thread attributes* of the newly created thread. Most invocations of `pthread_create()` pass `NULL` for `attr`, receiving the default attributes. Thread attributes let programs change many aspects of threads, such as their stack size, schedulizing parameters, and initial detached state. A full discussion of thread attributes is outside the scope of this chapter; the Pthread *man pages* are a good resource.

The `start_routine` must have the following signature:

```
void * start_thread (void *arg);
```

Thus, the thread begins life by executing a function that accepts a `void` pointer as its sole argument and returns a `void` pointer as its return value. Similar to `fork()`, the new thread inherits most attributes, capabilities, and state from its parent. Unlike `fork()`, threads *share* the resources of their parent instead of receiving a copy. The most notable shared resource is, of course, the process address space, but threads also share (in lieu of receiving copies of) signal handlers and open files.

Code using this function should pass `-pthread` to *gcc*. That is true for all the Pthread functions and I won't mention it again.

On error, `pthread_create()` returns a nonzero error code directly (without the use of `errno`) and the contents of `thread` are undefined. Likely errors include:

`EAGAIN`
: The invoking process lacks sufficient resources to create a new thread. Usually this is caused by the process hitting either a per-user or system-wide thread limit.

`EINVAL`
: The `pthread_attr_t` object pointed at by `attr` contains invalid attributes.

`EPERM`
: The `pthread_attr_t` object pointed at by `attr` contains attributes for which the invoking process does not have permission to instill.

Example usage:

```
pthread_t tread;
int ret;

ret = pthread_create (&thread, NULL, start_routine, NULL);
if (!ret) {
        errno = ret;
        perror("pthread_create");
        return -1;
}

/* a new thread is created and running start_routine concurrently ... */
```

We'll look at a full-program example once we build up a few more techniques.

Thread IDs

The *thread ID* (*TID*) is the thread analogue to the process ID (PID). While the PID is assigned by the Linux kernel, the TID is just assigned in the Pthread library.[3] It is an opaque type represented by `pthread_t`, and POSIX does not require it to be an arithmetic type. As we've seen, the TID of a new thread is provided via the `thread` argument in a successful call to `pthread_create()`. A thread can obtain its TID at runtime via the `pthread_self()` function:

```
#include <pthread.h>

pthread_t pthread_self (void);
```

Usage is simple, because the function cannot fail:

```
const pthread_t me = pthread_self ();
```

3. To the Linux kernel, threads are just processes that happen to share resources, so the kernel references each thread via a unique PID, just like any other process. User-space programs can obtain this PID via the `gettid()` system call, but only occasionally is this value useful. Programmers should use the Pthread ID concept to reference their threads.

Comparing thread IDs

Because the Pthread standard does not require `pthread_t` to be an arithmetic type, there is no guarantee that the equality operator will work. Consequently, to compare thread IDs, the Pthread library needs to provide a special interface:

```
#include <pthread.h>

int pthread_equal (pthread_t t1, pthread_t t2);
```

If the provided thread IDs are equal, `pthread_equal()` returns a nonzero value. If the provided thread IDs aren't equal, it returns 0; it cannot fail. Here's a simple example:

```
int ret;

ret = pthread_equal(thing1, thing2);
if (ret != 0)
        printf("The TIDs are equal!\n");
else
        printf("The TIDs are unequal!\n");
```

Terminating Threads

The counterpart to thread creation is thread termination. Thread termination is similar to process termination, except that when a thread terminates, the rest of the threads in the process continue executing. In some threading patterns, such as *thread-per-connection* (see "Thread-per-Connection" on page 217), threads are frequently created and destroyed.

Threads may terminate under several circumstances, all of which have analogues to process termination:

- If a thread returns from its start routine, it terminates. This is akin to "falling off the end" of `main()`.
- If a thread invokes the `pthread_exit()` function (discussed subsequently), it terminates. This is akin to calling `exit()`.
- If the thread is canceled by another thread via the `pthread_cancel()` function, it terminates. This is akin to being sent the `SIGKILL` signal via `kill()`.

These three examples kill only the thread in question. *All* of the threads in a process are killed, and thus the entire process is killed, in the following circumstances:

- The process returns from its `main()` function.
- The process terminates via `exit()`.
- The process executes a new binary image via `execve()`.

Signals can kill a process or an individual thread, depending on how they're issued. Pthreads make signal handling rather complicated; it is best to minimize the use of signals in multithreaded programs. See Chapter 10 for a full treatment of signals.

Terminating yourself

The easiest way for a thread to terminate itself is to "fall off the end" of its start routine. Often you want to terminate a thread deep in a function call stack, far from your start routine. For those cases, Pthreads provides `pthread_exit()`, the thread equivalent of `exit()`:

```
#include <pthread.h>

void pthread_exit (void *retval);
```

Upon invocation, the calling thread is terminated. `retval` is provided to any thread waiting on the terminating thread's death (see "Joining and Detaching Threads" on page 233), again similar to `exit()`. There is no chance of error.

Usage:

```
/* Goodbye, cruel world! */
pthread_exit (NULL);
```

Terminating others

Pthreads calls the termination of threads by other threads *cancellation*. It provides the `pthread_cancel()` function to do so:

```
#include <pthread.h>

int pthread_cancel (pthread_t thread);
```

A successful call to `pthread_cancel()` sends a cancellation request to the thread represented by the thread ID `thread`. Whether and when a thread is cancellable depends on its *cancellation state* and *cancellation type*, respectively. On success, `pthread_cancel()` returns zero. Note that success is only demonstrative of successfully processing the cancellation request. The actual termination occurs asynchronously. On error, `pthread_cancel()` returns `ESRCH`, indicating that `thread` was invalid.

If and when a thread is cancellable is a bit complicated. A thread's cancellation state is either *enabled* or *disabled*. The default for new threads is enabled. If a thread has disabled cancellation, the request is queued until it is enabled. Otherwise, the cancellation type dictates when cancellation occurs. Threads can change their state via `pthread_setcancelstate()`:

```
#include <pthread.h>

int pthread_setcancelstate (int state, int *oldstate);
```

On success, the cancellation state of the invoking thread is set to `state` and the old state is stored in `oldstate`.[4] `state` may be `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`, to enable or disable cancellation, respectively.

On error, `pthread_setcancelstate()` returns `EINVAL`, noting an invalid `state` value.

A thread's cancellation type is either *asynchronous* or *deferred*, with the latter being the default. With asynchronous cancellation, a thread may be killed at any point after the issuing of a cancel request. With deferred cancellation, a thread may only be killed at specific *cancellation points*, which are functions in the Pthreads or C library that represent safe points from which to terminate the caller. Asynchronous cancellation is useful only in certain situations because it can leave the process in an undefined state. For example, what if the canceled thread were in the middle of a critical region? For proper program behavior, asynchronous cancellation should be used only from threads that take care to never use shared resources and call only signal-safe functions (see "Guaranteed-Reentrant Functions" on page 349). Threads can change their type via `pthread_setcanceltype()`:

```
#include <pthread.h>

int pthread_setcanceltype (int type, int *oldtype);
```

On success, the cancellation type of the invoking thread is set to `type` and the old type is stored in `oldtype`.[5] `type` may be `PTHREAD_CANCEL_ASYNCHRONOUS` or `PTHREAD_CANCEL_DEFERRED`, to use asynchronous or deferred cancellation, respectively.

On error, `pthread_setcanceltype()` returns `EINVAL`, noting an invalid `type` value.

Let's consider an example of one thread terminating another. First, the to-terminate thread enables cancellation and sets the type to deferred (these are the defaults, so this acts only as an example):

```
int unused;
int ret;

ret = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &unused);
if (ret) {
        errno = ret;
        perror ("pthread_setcancelstate");
        return -1;
}

ret = pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, &unused);
if (ret) {
```

4. Linux allows for a `NULL` `oldstate`, but POSIX does not. Portable programs should always pass a valid pointer here, if only to ignore it.

5. As with `pthread_setcancelstate()`, it is not portable to pass `NULL` for `oldtype`, although Linux allows it.

```
        errno = ret;
        perror ("pthread_setcanceltype");
        return -1;
}
```

Next, a different thread sends the cancellation request:

```
int ret;

/* `thread' is the thread ID of the to-terminate thread */
ret = pthread_cancel (thread);
if (ret) {
        errno = ret;
        perror ("pthread_cancel");
        return -1;
}
```

Joining and Detaching Threads

Given that threads can be easily created and destroyed, there must be some way to synchronize threads against the termination of other threads—the threading equivalent of `wait()`. Indeed, there is: the joining of threads.

Joining threads

Joining allows one thread to block while waiting for the termination of another:

```
#include <pthread.h>

int pthread_join (pthread_t thread, void **retval);
```

Upon successful invocation, the invoking thread is blocked until the thread specified by `thread` terminates (if `thread` has already terminated, `pthread_join()` returns immediately). Once `thread` terminates, the invoking thread is woken up and, if `retval` is not `NULL`, provided the return value the terminated thread passed to `pthread_exit()` or returned from its start routine. We then say the threads are *joined*. Joining allows threads to synchronize their execution against the lifetime of other threads. All threads in Pthreads are peers; any thread may join any other. A single thread can join many threads (in fact, as we'll see, this is often how the main thread waits for the threads it has created), but only one thread should try to join any particular thread; multiple threads should not attempt to join with any one other.

On error, `pthread_join()` returns one of the following nonzero error codes:

`EDEADLK`
: A deadlock was detected: `thread` is already waiting to join the caller or `thread` *is* the caller.

`EINVAL`
: The thread specified by `thread` is not joinable (see next section).

ESRCH
: The thread specified by `thread` is invalid.

Example usage:

```
int ret;

/* join with `thread' and we don't care about its return value */
ret = pthread_join (thread, NULL);
if (ret) {
        errno = ret;
        perror ("pthread_join");
        return -1;
}
```

Detaching threads

By default, threads are created as *joinable*. Threads may, however, *detach*, rendering them no longer joinable. Because threads consume system resources until joined, just as processes consume system resources until their parent calls `wait()`, threads that you do not intend to join should be detached.

```
#include <pthread.h>

int pthread_detach (pthread_t thread);
```

On success, `pthread_detach()` detaches the thread specified by `thread` and returns zero. Results are undefined if you call `pthread_detach()` on a thread that is already detached. On error, the function returns `ESRCH` indicating that `thread` is invalid.

Either `pthread_join()` or `pthread_detach()` should be called *on each thread in a process* so that system resources are released when the thread terminates. (Of course, if the entire process exits, all threading resources are freed, but it remains good practice to explicitly join or detach all threads.)

A Threading Example

The following full-program example ties together the interfaces discussed thus far. It creates two threads (for a total of three), starting both threads in the same start routine, `start_thread()`. It differentiates their behavior in the start routine by providing differing arguments. It then joins both threads; if it did not, the main thread could exit before the other threads, terminating the entire process.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void * start_thread (void *message)
{
        printf ("%s\n", (const char *) message);
```

```
        return message;
}

int main (void)
{
        pthread_t thing1, thing2;
        const char *message1 = "Thing 1";
        const char *message2 = "Thing 2";

        /* Create two threads, each with a different message. */
        pthread_create (&thing1, NULL, start_thread, (void *) message1);
        pthread_create (&thing2, NULL, start_thread, (void *) message2);

        /*
         * Wait for the threads to exit. If we didn't join here,
         * we'd risk terminating this main thread before the
         * other two threads finished.
         */
        pthread_join (thing1, NULL);
        pthread_join (thing2, NULL);

        return 0;
}
```

This is a full program. If you save it as *example.c*, you can compile it with this command:

```
gcc -Wall -O2 -pthread example.c -o example
```

and then run it like so:

```
./example
```

yielding:

```
Thing 1
Thing 2
```

or perhaps:

```
Thing 2
Thing 1
```

but never gibberish. Why no gibberish? Because `printf()` is thread-safe.

Pthread Mutexes

Recall from "Mutexes" on page 222 that the primary method of ensuring mutual exclusion is the mutex. For all their power and importance, mutexes are in fact rather easy to use.

Initializing mutexes

Mutexes are represented by the `pthread_mutex_t` object. Like most of the objects in the Pthread API, it is meant to be an opaque structure provided to the various mutex interfaces. Although you can dynamically create mutexes, most uses are static:

```
/* define and initialize a mutex named `mutex' */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

This snippet defines and initializes a mutex named `mutex`. That is all we have to do to start using it.

Locking mutexes

Locking (also called *acquiring*) a Pthreads mutex is accomplished via the `pthread_mutex_lock()` function:

```
#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);
```

A successful call to `pthread_mutex_lock()` will block the calling thread until the mutex pointed at by `mutex` becomes available. Once available, the calling thread will wake up and this function will return zero. If the mutex is available on invocation, the function will return immediately.

On error, the function returns one of the following nonzero error codes:

`EDEADLK`
: The invoking thread already holds the requested mutex. This error code is not guaranteed by default; attempting to acquire an already-held mutex may result in deadlock (see "Deadlocks" on page 224).

`EINVAL`
: The mutex pointed at by `mutex` is invalid.

Callers tend not to check the return value, since well-formed code should not generate any errors at runtime. An example of usage is:

```
pthread_mutex_lock (&mutex);
```

Unlocking mutexes

The counterpart to locking is unlocking, or *releasing*, the mutex.

```
#include <pthread.h>

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

A successful call to `pthread_mutex_unlock()` releases the mutex pointed at by `mutex` and returns zero. The call does not block; the mutex is released immediately.

On error, the function returns a nonzero error code, including:

`EINVAL`
: The mutex pointed at by `mutex` is invalid.

`EPERM`
: The invoking process does not hold the mutex pointed at by `mutex`. This error code is not guaranteed; attempting to release a mutex you do not hold is a bug.

As with locking, users tend not to check the return value:

```
pthread_mutex_unlock (&mutex);
```

Scoped Locks

Resource Acquisition Is Initialization (*RAII*) is a C++ programming pattern—one of the most powerful patterns in the language. RAII efficiently deals with resource allocation and deallocation by tying the lifetime of the resource to the lifetime of a scoped object. While RAII was created to deal with resource cleanup after an exception is thrown, it is most powerful as a way to manage resources. For example, RAII lets you create a "scoped file" object where the file is opened when the object is created and automatically closed when the object falls out of scope. Similarly, we can create a "scoped lock" that acquires a mutex on creation and automatically releases the mutex when it falls out of scope:

```
class ScopedMutex {
    public:
        ScopedMutex (pthread_mutex_t& mutex)
            :mutex_ (mutex)
        {
            pthread_mutex_lock (&mutex_);
        }

        ~ScopedMutex ()
        {
            pthread_mutex_unlock (&mutex_);
        }

    private:
        pthread_mutex_t& mutex_;
};
```

To use this, one just calls `ScopedMutex m(mutex)`. The lock is automatically released when `m` falls out of scope. This makes unwinding functions and handling errors convenient and free of `goto` statements.

Mutex example

Let's look at a simple snippet that utilizes a mutex to ensure synchronization. Recall the banking withdraw example in "Real-world races" on page 219. Our imaginary bank suffered a serious race condition, enabling unintended behavior. Here's how we could fix the withdraw function using Pthread's mutexes:

```
static pthread_mutex_t the_mutex = PTHREAD_MUTEX_INITIALIZER;

int withdraw (struct account *account, int amount)
{
        pthread_mutex_lock (&the_mutex);
        const int balance = account->balance;
        if (balance < amount) {
                pthread_mutex_unlock (&the_mutex);
                return -1;
        }
        account->balance = balance - amount;
        pthread_mutex_unlock (&the_mutex);

        disburse_money (amount);

        return 0;
}
```

This example uses `pthread_mutex_lock()` to acquire a mutex and then `pthread_mutex_unlock()` to eventually release it. This serves to eliminate the race condition, but it introduces a single point of contention in the bank: only one customer can withdraw money at a time! That is quite a bottleneck; for a too-big-to-fail bank, that's a failure.

Thus most uses of locks avoid *global locks* and instead associate locks with specific instances of data structures. This is called *fine-grained locking*. It can make your locking semantics more complicated, particularly around deadlock avoidance, but is key in scaling to the number of cores on modern machines.

In this example, instead of defining a global `the_mutex` lock, we define a mutex inside of the `account` structure, giving each account its own lock. This works well as the data within the critical region is only the `account` structure. By locking only the account being debited, we allow the bank to process *other* customers' withdraws in parallel.

```
int withdraw (struct account *account, int amount)
{
        pthread_mutex_lock (&account->mutex);
        const int balance = account->balance;
        if (balance < amount) {
                pthread_mutex_unlock (&account->mutex);
                return -1;
        }
        account->balance = balance - amount;
        pthread_mutex_unlock (&account->mutex);

        disburse_money (amount);

        return 0;
}
```

Further Study

A single chapter cannot hope to do justice to the POSIX threading API, which, if we are being kind, is a full-featured and powerful library with myriad interfaces to learn. (If we are feeling grouchy, we might say that POSIX threads are overly complicated and cumbersome.) Many large-scale system applications define their own threading interfaces, as mechanisms such as thread pools and work queues are a more relevant level of abstraction in system software than what POSIX provides. In that case, the background on threading in this chapter is a perfect introduction to your in-house solution.

If you do require a deeper dive into Pthreads, I recommend the further reading in Appendix B. The relevant man pages are also particularly helpful.