



C++

Concurrency IN ACTION

SECOND EDITION

Anthony Williams



MANNING



MEAP Edition
Manning Early Access Program
C++ Concurrency in Action
Second Edition
Version 6

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for the 2nd edition of *C++ Concurrency in Action*. I'm glad to get the opportunity to extend the first edition to cover the new concurrency and parallelism facilities from C++17 and the Technical Specification for Concurrency in C++ (the Concurrency TS). This is an intermediate level book designed for anyone who's writing

The book will cover designing multithreaded algorithms and data structures, with and without locks, using the parallel algorithms from C++17, testing concurrent programs, and the appendices will provide a reference to the multithreading parts of the C++ standard library.

As you're reading, I hope you'll take advantage of the Author Online forum. I'll be reading your comments and responding, and your feedback is helpful in the development process.

—Anthony Williams

brief contents

- 1 Hello, world of concurrency in C++!*
- 2 Managing threads*
- 3 Sharing data between threads*
- 4 Synchronizing concurrent operations*
- 5 The C++ memory model and operations on atomic types*
- 6 Designing lock-based concurrent data structures*
- 7 Designing lock-free concurrent data structures*
- 8 Designing concurrent code*
- 9 Advanced thread management*
- 10 Using the Standard Library Parallelism Facilities*
- 11 Testing and debugging multithreaded applications*

1

Hello, world of concurrency in C++!

This chapter covers

- **What is meant by concurrency and multithreading**
- **Why you might want to use concurrency and multithreading in your applications**
- **Some of the history of the support for concurrency in C++**
- **What a simple multithreaded C++ program looks like**

These are exciting times for C++ users. Thirteen years after the original C++ Standard was published in 1998, the C++ Standards Committee gave the language and its supporting library a major overhaul. The new C++ Standard (referred to as C++11 or C++0x) was published in 2011 and brought with it a whole swathe of changes that made working with C++ easier and more productive. The Committee also committed to a new “train model” of releases, with a new C++ Standard to be published every 3 years. So far, we've had 2 such publications: the C++14 Standard in 2014, and the C++17 Standard in 2017, as well as several “Technical Specifications” describing extensions to the C++ Standard.

One of the most significant new features in the C++11 Standard was the support of multithreaded programs. For the first time, the C++ Standard acknowledged the existence of multithreaded applications in the language and provided components in the library for writing multithreaded applications. This made it possible to write multithreaded C++ programs without relying on platform-specific extensions and thus allow writing portable multithreaded code with guaranteed behavior. It also came at a time when programmers are increasingly looking to concurrency in general, and multithreaded programming in particular, to improve application performance. The C++14 Standard and C++17 Standard have built upon this

baseline, to provide further support for writing multithreaded programs in C++, as have the Technical Specifications: there is a Technical Specification for Concurrency Extensions, and another for Parallelism, though the latter has been incorporated into C++17.

This book is about writing programs in C++ using multiple threads for concurrency and the C++ language features and library facilities that make that possible. I'll start by explaining what I mean by concurrency and multithreading and why you would want to use concurrency in your applications. After a quick detour into why you might *not* want to use it in your applications, we'll go through an overview of the concurrency support in C++, and we'll round off this chapter with a simple example of C++ concurrency in action. Readers experienced with developing multithreaded applications may wish to skip the early sections. In subsequent chapters we'll cover more extensive examples and look at the library facilities in more depth. The book will finish with an in-depth reference to all the C++ Standard Library facilities for multithreading and concurrency.

So, what do I mean by *concurrency* and *multithreading*?

1.1 What is concurrency?

At the simplest and most basic level, concurrency is about two or more separate activities happening at the same time. We encounter concurrency as a natural part of life; we can walk and talk at the same time or perform different actions with each hand, and of course we each go about our lives independently of each other—you can watch football while I go swimming, and so on.

1.1.1 Concurrency in computer systems

When we talk about concurrency in terms of computers, we mean a single system performing multiple independent activities in parallel, rather than sequentially, or one after the other. It isn't a new phenomenon: multitasking operating systems that allow a single desktop computer to run multiple applications at the same time through task switching have been commonplace for many years, as have high-end server machines with multiple processors that enable genuine concurrency. What *is* new is the increased prevalence of computers that can genuinely run multiple tasks in parallel rather than just giving the illusion of doing so.

Historically, most desktop computers have had one processor, with a single processing unit or core, and this remains true for many desktop machines today. Such a machine can really only perform one task at a time, but it can switch between tasks many times per second. By doing a bit of one task and then a bit of another and so on, it appears that the tasks are happening concurrently. This is called *task switching*. We still talk about *concurrency* with such systems; because the task switches are so fast, you can't tell at which point a task may be suspended as the processor switches to another one. The task switching provides an illusion of concurrency to both the user and the applications themselves. Because there is only an *illusion* of concurrency, the behavior of applications may be subtly different when executing in a single-processor task-switching environment compared to when executing in an environment with true concurrency. In particular, incorrect assumptions about the memory model (covered

in chapter 5) may not show up in such an environment. This is discussed in more depth in chapter 10.

Computers containing multiple processors have been used for servers and high-performance computing tasks for a number of years, and now computers based on processors with more than one core on a single chip (multicore processors) are becoming increasingly common as desktop machines too. Whether they have multiple processors or multiple cores within a processor (or both), these computers are capable of genuinely running more than one task in parallel. We call this *hardware concurrency*.

Figure 1.1 shows an idealized scenario of a computer with precisely two tasks to do, each divided into 10 equal-size chunks. On a dual-core machine (which has two processing cores), each task can execute on its own core. On a single-core machine doing task switching, the chunks from each task are interleaved. But they are also spaced out a bit (in the diagram this is shown by the gray bars separating the chunks being thicker than the separator bars shown for the dual-core machine); in order to do the interleaving, the system has to perform a *context switch* every time it changes from one task to another, and this takes time. In order to perform a context switch, the OS has to save the CPU state and instruction pointer for the currently running task, work out which task to switch to, and reload the CPU state for the task being switched to. The CPU will then potentially have to load the memory for the instructions and data for the new task into cache, which can prevent the CPU from executing any instructions, causing further delay.

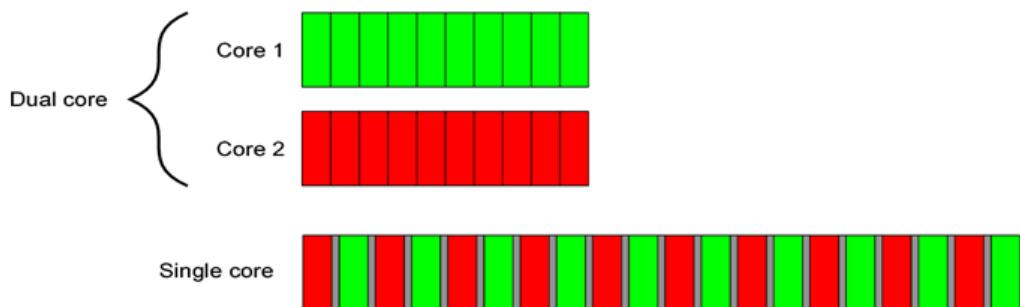


Figure 1.1 Two approaches to concurrency: parallel execution on a dual-core machine versus task switching on a single-core machine

Though the availability of concurrency in the hardware is most obvious with multiprocessor or multicore systems, some processors can execute multiple threads on a single core. The important factor to consider is really the number of *hardware threads*: the measure of how many independent tasks the hardware can genuinely run concurrently. Even with a system that has genuine hardware concurrency, it's easy to have more tasks than the hardware can run in parallel, so task switching is still used in these cases. For example, on a typical desktop computer there may be hundreds of tasks running, performing background operations, even

when the computer is nominally idle. It's the task switching that allows these background tasks to run and allows you to run your word processor, compiler, editor, and web browser (or any combination of applications) all at once. Figure 1.2 shows task switching among four tasks on a dual-core machine, again for an idealized scenario with the tasks divided neatly into equal-size chunks. In practice, many issues will make the divisions uneven and the scheduling irregular. Some of these issues are covered in chapter 8 when we look at factors affecting the performance of concurrent code.

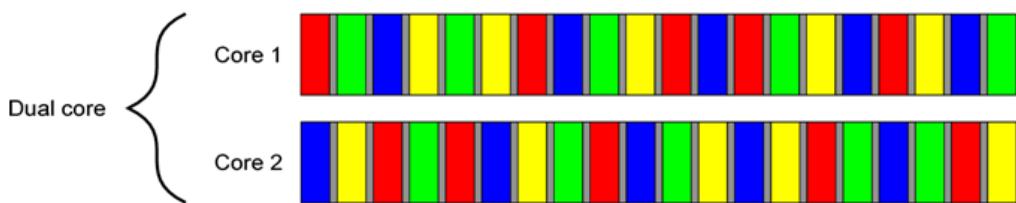


Figure 1.2 Task switching of four tasks on two cores

All the techniques, functions, and classes covered in this book can be used whether your application is running on a machine with one single-core processor or on a machine with many multicore processors and are not affected by whether the concurrency is achieved through task switching or by genuine hardware concurrency. But as you may imagine, how you make use of concurrency in your application may well depend on the amount of hardware concurrency available. This is covered in chapter 8, where I cover the issues involved with designing concurrent code in C++.

1.1.2 Approaches to concurrency

Imagine for a moment a pair of programmers working together on a software project. If your developers are in separate offices, they can go about their work peacefully, without being disturbed by each other, and they each have their own set of reference manuals. However, communication is not straightforward; rather than just turning around and talking to each other, they have to use the phone or email or get up and walk to each other's office. Also, you have the overhead of two offices to manage and multiple copies of reference manuals to purchase.

Now imagine that you move your developers into the same office. They can now talk to each other freely to discuss the design of the application, and they can easily draw diagrams on paper or on a whiteboard to help with design ideas or explanations. You now have only one office to manage, and one set of resources will often suffice. On the negative side, they might find it harder to concentrate, and there may be issues with sharing resources ("Where's the reference manual gone now?").

These two ways of organizing your developers illustrate the two basic approaches to concurrency. Each developer represents a thread, and each office represents a process. The

first approach is to have multiple single-threaded processes, which is similar to having each developer in their own office, and the second approach is to have multiple threads in a single process, which is like having two developers in the same office. You can combine these in an arbitrary fashion and have multiple processes, some of which are multithreaded and some of which are single-threaded, but the principles are the same. Let's now have a brief look at these two approaches to concurrency in an application.

CONCURRENCY WITH MULTIPLE PROCESSES

The first way to make use of concurrency within an application is to divide the application into multiple, separate, single-threaded processes that are run at the same time, much as you can run your web browser and word processor at the same time. These separate processes can then pass messages to each other through all the normal interprocess communication channels (signals, sockets, files, pipes, and so on), as shown in figure 1.3. One downside is that such communication between processes is often either complicated to set up or slow or both, because operating systems typically provide a lot of protection between processes to avoid one process accidentally modifying data belonging to another process. Another downside is that there's an inherent overhead in running multiple processes: it takes time to start a process, the operating system must devote internal resources to managing the process, and so forth.

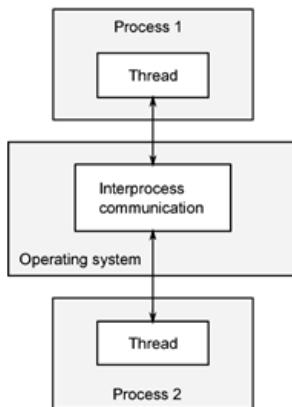


Figure 1.3 Communication between a pair of processes running concurrently

Of course, it's not all downside: the added protection operating systems typically provide between processes and the higher-level communication mechanisms mean that it can be easier to write *safe* concurrent code with processes rather than threads. Indeed, environments

such as that provided for the Erlang¹ programming language use processes as the fundamental building block of concurrency to great effect.

Using separate processes for concurrency also has an additional advantage—you can run the separate processes on distinct machines connected over a network. Though this increases the communication cost, on a carefully designed system it can be a cost-effective way of increasing the available parallelism and improving performance.

CONCURRENCY WITH MULTIPLE THREADS

The alternative approach to concurrency is to run multiple threads in a single process. Threads are much like lightweight processes: each thread runs independently of the others, and each thread may run a different sequence of instructions. But all threads in a process share the same address space, and most of the data can be accessed directly from all threads—global variables remain global, and pointers or references to objects or data can be passed around among threads. Although it's often possible to share memory among processes, this is complicated to set up and often hard to manage, because memory addresses of the same data aren't necessarily the same in different processes. Figure 1.4 shows two threads within a process communicating through shared memory.

The shared address space and lack of protection of data between threads makes the overhead associated with using multiple threads much smaller than that from using multiple processes, because the operating system has less bookkeeping to do. But the flexibility of shared memory also comes with a price: if data is accessed by multiple threads, the application programmer must ensure that the view of data seen by each thread is consistent whenever it is accessed. The issues surrounding sharing data between threads and the tools to use and guidelines to follow to avoid problems are covered throughout this book, notably in chapters 3, 4, 5, and 8. The problems are not insurmountable, provided suitable care is taken when writing the code, but they do mean that a great deal of thought must go into the communication between threads.

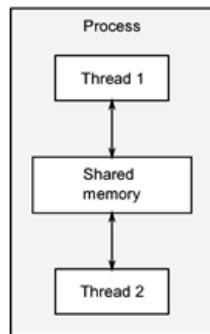


Figure 1.4 Communication between a pair of threads running concurrently in a single process

¹ See <https://www.erlang.org/>

The low overhead associated with launching and communicating between multiple threads within a process compared to launching and communicating between multiple single-threaded processes means that this is the favored approach to concurrency in mainstream languages including C++, despite the potential problems arising from the shared memory. In addition, the C++ Standard doesn't provide any intrinsic support for communication between processes, so applications that use multiple processes will have to rely on platform-specific APIs to do so. This book therefore focuses exclusively on using multithreading for concurrency, and future references to concurrency assume that this is achieved by using multiple threads.

There is another word that gets used a lot around multithreaded code: *parallelism*. It probably pays to clarify the differences.

1.1.3 Concurrency vs Parallelism

Concurrency and parallelism have largely overlapping meanings with respect to multithreaded code. Indeed, to many they mean essentially the same thing. The difference is primarily a matter of nuance, focus and intent. Both terms are about running multiple tasks simultaneously, using the available hardware, but parallelism is much more performance-oriented. People talk about *parallelism* when their primary concern is taking advantage of the available hardware to increase the performance of bulk data processing, whereas people talk about *concurrency* when their primary concern is separation of concerns, or responsiveness. This dichotomy is not cut and dried, and there is still considerable overlap in meaning, but it can help clarify discussions to know of this distinction. Throughout this book, there will be examples of both.

Having clarified what we mean by concurrency and parallelism, let's now look at why you would use concurrency in your applications.

1.2 Why use concurrency?

There are two main reasons to use concurrency in an application: separation of concerns and performance. In fact, I'd go so far as to say that they're pretty much the *only* reasons to use concurrency; anything else boils down to one or the other (or maybe even both) when you look hard enough (well, except for reasons like "because I want to").

1.2.1 Using concurrency for separation of concerns

Separation of concerns is almost always a good idea when writing software; by grouping related bits of code together and keeping unrelated bits of code apart, you can make your programs easier to understand and test, and thus less likely to contain bugs. You can use concurrency to separate distinct areas of functionality, even when the operations in these distinct areas need to happen at the same time; without the explicit use of concurrency you either have to write a task-switching framework or actively make calls to unrelated areas of code during an operation.

Consider a processing-intensive application with a user interface, such as a DVD player application for a desktop computer. Such an application fundamentally has two sets of

responsibilities: not only does it have to read the data from the disk, decode the images and sound, and send them to the graphics and sound hardware in a timely fashion so the DVD plays without glitches, but it must also take input from the user, such as when the user clicks Pause or Return To Menu, or even Quit. In a single thread, the application has to check for user input at regular intervals during the playback, thus conflating the DVD playback code with the user interface code. By using multithreading to separate these concerns, the user interface code and DVD playback code no longer have to be so closely intertwined; one thread can handle the user interface and another the DVD playback. There will have to be interaction between them, such as when the user clicks Pause, but now these interactions are directly related to the task at hand.

This gives the illusion of responsiveness, because the user interface thread can typically respond immediately to a user request, even if the response is simply to display a busy cursor or Please Wait message while the request is conveyed to the thread doing the work. Similarly, separate threads are often used to run tasks that must run continuously in the background, such as monitoring the filesystem for changes in a desktop search application. Using threads in this way generally makes the logic in each thread much simpler, because the interactions between them can be limited to clearly identifiable points, rather than having to intersperse the logic of the different tasks.

In this case, the number of threads is independent of the number of CPU cores available, because the division into threads is based on the conceptual design rather than an attempt to increase throughput.

1.2.2 Using concurrency for performance: task parallelism and data parallelism

Multiprocessor systems have existed for decades, but until recently they were mostly found only in supercomputers, mainframes, and large server systems. But chip manufacturers have increasingly been favoring multicore designs with 2, 4, 16, or more processors on a single chip over better performance with a single core. Consequently, multicore desktop computers, and even multicore embedded devices, are now increasingly prevalent. The increased computing power of these machines comes not from running a single task faster but from running multiple tasks in parallel. In the past, programmers have been able to sit back and watch their programs get faster with each new generation of processors, without any effort on their part. But now, as Herb Sutter put it, “The free lunch is over.”² *If software is to take advantage of this increased computing power, it must be designed to run multiple tasks concurrently.* Programmers must therefore take heed, and those who have hitherto ignored concurrency must now look to add it to their toolbox.

There are two ways to use concurrency for performance. The first, and most obvious, is to divide a single task into parts and run each in parallel, thus reducing the total runtime. This is *task parallelism*. Although this sounds straightforward, it can be quite a complex process,

² “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” Herb Sutter, *Dr. Dobb’s Journal*, 30(3), March 2005.
<http://www.gotw.ca/publications/concurrency-ddj.htm>.

because there may be many dependencies between the various parts. The divisions may be either in terms of processing—one thread performs one part of the algorithm while another thread performs a different part—or in terms of data—each thread performs the same operation on different parts of the data. This latter approach is called *data parallelism*.

Algorithms that are readily susceptible to such parallelism are frequently called *embarrassingly parallel*. Despite the implications that you might be embarrassed to have code so easy to parallelize, this is a good thing: other terms I've encountered for such algorithms are *naturally parallel* and *conveniently concurrent*. Embarrassingly parallel algorithms have good scalability properties—as the number of available hardware threads goes up, the parallelism in the algorithm can be increased to match. Such an algorithm is the perfect embodiment of the adage, “Many hands make light work.” For those parts of the algorithm that aren’t embarrassingly parallel, you might be able to divide the algorithm into a fixed (and therefore not scalable) number of parallel tasks. Techniques for dividing tasks between threads are covered in chapters 8 and 10.

The second way to use concurrency for performance is to use the available parallelism to solve bigger problems; rather than processing one file at a time, process 2 or 10 or 20, as appropriate. Although this is really just an application of *data parallelism*, by performing the same operation on multiple sets of data concurrently, there’s a different focus. It still takes the same amount of time to process one chunk of data, but now more data can be processed in the same amount of time. Obviously, there are limits to this approach too, and this won’t be beneficial in all cases, but the increase in throughput that comes from such an approach can actually make new things possible—increased resolution in video processing, for example, if different areas of the picture can be processed in parallel.

1.2.3 When not to use concurrency

It’s just as important to know *when not* to use concurrency as it is to know *when* to use it. Fundamentally, the only reason not to use concurrency is when the benefit is not worth the cost. Code using concurrency is harder to understand in many cases, so there’s a direct intellectual cost to writing and maintaining multithreaded code, and the additional complexity can also lead to more bugs. Unless the potential performance gain is large enough or separation of concerns clear enough to justify the additional development time required to get it right and the additional costs associated with maintaining multithreaded code, don’t use concurrency.

Also, the performance gain might not be as large as expected; there’s an inherent overhead associated with launching a thread, because the OS has to allocate the associated kernel resources and stack space and then add the new thread to the scheduler, all of which takes time. If the task being run on the thread is completed quickly, the actual time taken by the task may be dwarfed by the overhead of launching the thread, possibly making the overall performance of the application worse than if the task had been executed directly by the spawning thread.

Furthermore, threads are a limited resource. If you have too many threads running at once, this consumes OS resources and may make the system as a whole run slower. Not only that, but using too many threads can exhaust the available memory or address space for a process, because each thread requires a separate stack space. This is particularly a problem for 32-bit processes with a flat architecture where there's a 4 GB limit in the available address space: if each thread has a 1 MB stack (as is typical on many systems), then the address space would be all used up with 4096 threads, without allowing for any space for code or static data or heap data. Although 64-bit (or larger) systems don't have this direct address-space limit, they still have finite resources: if you run too many threads, this will eventually cause problems. Though thread pools (see chapter 9) can be used to limit the number of threads, these are not a silver bullet, and they do have their own issues.

If the server side of a client/server application launches a separate thread for each connection, this works fine for a small number of connections, but can quickly exhaust system resources by launching too many threads if the same technique is used for a high-demand server that has to handle many connections. In this scenario, careful use of thread pools can provide optimal performance (see chapter 9).

Finally, the more threads you have running, the more context switching the operating system has to do. Each context switch takes time that could be spent doing useful work, so at some point adding an extra thread will actually *reduce* the overall application performance rather than increase it. For this reason, if you're trying to achieve the best possible performance of the system, it's necessary to adjust the number of threads running to take account of the available hardware concurrency (or lack of it).

Use of concurrency for performance is just like any other optimization strategy: it has potential to greatly improve the performance of your application, but it can also complicate the code, making it harder to understand and more prone to bugs. Therefore it's only worth doing for those performance-critical parts of the application where there's the potential for measurable gain. Of course, if the potential for performance gains is only secondary to clarity of design or separation of concerns, it may still be worth using a multithreaded design.

Assuming that you've decided you *do* want to use concurrency in your application, whether for performance, separation of concerns, or because it's "multithreading Monday," what does that mean for C++ programmers?

1.3 Concurrency and multithreading in C++

Standardized support for concurrency through multithreading is a relatively new thing for C++. It's only since the C++11 Standard that you've been able to write multithreaded code without resorting to platform-specific extensions. In order to understand the rationale behind lots of the decisions in the Standard C++ Thread Library, it's important to understand the history.

1.3.1 History of multithreading in C++

The 1998 C++ Standard doesn't acknowledge the existence of threads, and the operational effects of the various language elements are written in terms of a sequential abstract machine. Not only that, but the memory model isn't formally defined, so you can't write multithreaded applications without compiler-specific extensions to the 1998 C++ Standard.

Of course, compiler vendors are free to add extensions to the language, and the prevalence of C APIs for multithreading—such as those in the POSIX C standard and the Microsoft Windows API—has led many C++ compiler vendors to support multithreading with various platform-specific extensions. This compiler support is generally limited to allowing the use of the corresponding C API for the platform and ensuring that the C++ Runtime Library (such as the code for the exception-handling mechanism) works in the presence of multiple threads. Although very few compiler vendors have provided a formal multithreading-aware memory model, the actual behavior of the compilers and processors has been sufficiently good that a large number of multithreaded C++ programs have been written.

Not content with using the platform-specific C APIs for handling multithreading, C++ programmers have looked to their class libraries to provide object-oriented multithreading facilities. Application frameworks such as MFC and general-purpose C++ libraries such as Boost and ACE have accumulated sets of C++ classes that wrap the underlying platform-specific APIs and provide higher-level facilities for multithreading that simplify tasks. Although the precise details of the class libraries have varied considerably, particularly in the area of launching new threads, the overall shape of the classes has had a lot in common. One particularly important design that's common to many C++ class libraries, and that provides considerable benefit to the programmer, has been the use of the *Resource Acquisition Is Initialization* (RAII) idiom with locks to ensure that mutexes are unlocked when the relevant scope is exited.

For many cases, the multithreading support of existing C++ compilers combined with the availability of platform-specific APIs and platform-independent class libraries such as Boost and ACE provide a solid foundation on which to write multithreaded C++ code, and as a result there are probably millions of lines of C++ code written as part of multithreaded applications. But the lack of standard support means that there are occasions where the lack of a thread-aware memory model causes problems, particularly for those who try to gain higher performance by using knowledge of the processor hardware or for those writing cross-platform code where the actual behavior of the compilers varies between platforms.

1.3.2 Concurrency support in the C++11 standard

All this changed with the release of the C++11 Standard. Not only is there a thread-aware memory model, but the C++ Standard Library was extended to include classes for managing threads (see chapter 2), protecting shared data (see chapter 3), synchronizing operations between threads (see chapter 4), and low-level atomic operations (see chapter 5).

The C++11 Thread Library is heavily based on the prior experience accumulated through the use of the C++ class libraries mentioned previously. In particular, the Boost Thread

Library was used as the primary model on which the new library is based, with many of the classes sharing their names and structure with the corresponding ones from Boost. As the standard evolved, this has been a two-way flow, and the Boost Thread Library has itself changed to match the C++ Standard in many respects, so users transitioning from Boost should find themselves very much at home.

Concurrency support is just one of the changes with the C++11 Standard—as mentioned at the beginning of this chapter, there are many enhancements to the language itself to make programmers' lives easier. Although these are generally outside the scope of this book, some of those changes have had a direct impact on the Thread Library itself and the ways in which it can be used. Appendix A provides a brief introduction to these language features.

1.3.3 More support for Concurrency and Parallelism in C++14 and C++17

The only specific support for concurrency and parallelism added in C++14 was a new mutex type for protecting shared data (see chapter 3). However, C++17 adds considerably more: a full suite of parallel algorithms (see chapter 10) for starters. Of course, both of these Standards enhance the core language and the rest of the Standard Library, and these enhancements can simplify the writing of multithreaded code.

As mentioned above, there is also a Technical Specification for Concurrency, which describes extensions to the functions and classes provided by the C++ Standard, especially around synchronizing operations between threads (see chapter 4).

The support for atomic operations directly in C++ enables programmers to write efficient code with defined semantics without the need for platform-specific assembly language. This is a real boon for those trying to write efficient, portable code; not only does the compiler take care of the platform specifics, but the optimizer can be written to take into account the semantics of the operations, thus enabling better optimization of the program as a whole.

1.3.4 Efficiency in the C++ Thread Library

One of the concerns that developers involved in high-performance computing often raise regarding C++ in general, and C++ classes that wrap low-level facilities—such as those in the new Standard C++ Thread Library specifically—is that of efficiency. If you're after the utmost in performance, then it's important to understand the implementation costs associated with using any high-level facilities, compared to using the underlying low-level facilities directly. This cost is the *abstraction penalty*.

The C++ Standards Committee has been very aware of this when designing the C++ Standard Library in general and the Standard C++ Thread Library in particular; one of the design goals has been that there should be little or no benefit to be gained from using the lower-level APIs directly, where the same facility is to be provided. The library has therefore been designed to allow for efficient implementation (with a very low abstraction penalty) on most major platforms.

Another goal of the C++ Standards Committee has been to ensure that C++ provides sufficient low-level facilities for those wishing to work close to the metal for the ultimate

performance. To this end, along with the new memory model comes a comprehensive atomic operations library for direct control over individual bits and bytes and the inter-thread synchronization and visibility of any changes. These atomic types and the corresponding operations can now be used in many places where developers would previously have chosen to drop down to platform-specific assembly language. Code using the new standard types and operations is thus more portable and easier to maintain.

The C++ Standard Library also provides higher-level abstractions and facilities that make writing multithreaded code easier and less error prone. Sometimes the use of these facilities does come with a performance cost because of the additional code that must be executed. But this performance cost doesn't necessarily imply a higher abstraction penalty; in general the cost is no higher than would be incurred by writing equivalent functionality by hand, and the compiler may well inline much of the additional code anyway.

In some cases, the high-level facilities provide additional functionality beyond what may be required for a specific use. Most of the time this is not an issue: you don't pay for what you don't use. On rare occasions, this unused functionality will impact the performance of other code. If you're aiming for performance and the cost is too high, you may be better off handcrafting the desired functionality from lower-level facilities. In the vast majority of cases, the additional complexity and chance of errors far outweigh the potential benefits from a small performance gain. Even if profiling *does* demonstrate that the bottleneck is in the C++ Standard Library facilities, it may be due to poor application design rather than a poor library implementation. For example, if too many threads are competing for a mutex, it *will* impact the performance significantly. Rather than trying to shave a small fraction of time off the mutex operations, it would probably be more beneficial to restructure the application so that there's less contention on the mutex. Designing applications to reduce contention is covered in chapter 8.

In those very rare cases where the C++ Standard Library does not provide the performance or behavior required, it might be necessary to use platform-specific facilities.

1.3.5 Platform-specific facilities

Although the C++ Thread Library provides reasonably comprehensive facilities for multithreading and concurrency, on any given platform there will be platform-specific facilities that go beyond what's offered. In order to gain easy access to those facilities without giving up the benefits of using the Standard C++ Thread Library, the types in the C++ Thread Library may offer a `native_handle()` member function that allows the underlying implementation to be directly manipulated using a platform-specific API. By its very nature, any operations performed using the `native_handle()` are entirely platform dependent and out of the scope of this book (and the Standard C++ Library itself).

Of course, before even considering using platform-specific facilities, it's important to understand what the Standard Library provides, so let's get started with an example.

1.4 Getting started

OK, so you have a nice, shiny C++11/C++14/C++17 compiler. What next? What does a multithreaded C++ program look like? It looks pretty much like any other C++ program, with the usual mix of variables, classes, and functions. The only real distinction is that some functions might be running concurrently, so you need to ensure that shared data is safe for concurrent access, as described in chapter 3. Of course, in order to run functions concurrently, specific functions and objects must be used to manage the different threads.

1.4.1 Hello, Concurrent World

Let's start with a classic example: a program to print "Hello World." A really simple Hello, World program that runs in a single thread is shown here, to serve as a baseline when we move to multiple threads:

```
#include <iostream>
int main()
{
    std::cout<<"Hello World\n";
}
```

All this program does is write "Hello World" to the standard output stream. Let's compare it to the simple Hello, Concurrent World program shown in the following listing, which starts a separate thread to display the message.

Listing 1.1 A simple Hello, Concurrent World program

```
#include <iostream>
#include <thread>          #1
void hello()               #2
{
    std::cout<<"Hello Concurrent World\n";
}
int main()
{
    std::thread t(hello);   #3
    t.join();               #4
}
```

The first difference is the extra `#include <thread>` #1. The declarations for the multithreading support in the Standard C++ Library are in new headers: the functions and classes for managing threads are declared in `<thread>`, whereas those for protecting shared data are declared in other headers.

Second, the code for writing the message has been moved to a separate function #2. This is because every thread has to have an *initial function*, which is where the new thread of execution begins. For the initial thread in an application, this is `main()`, but for every other thread it's specified in the constructor of a `std::thread` object—in this case, the `std::thread` object named `t` #3 has the new function `hello()` as its initial function.

This is the next difference: rather than just writing directly to standard output or calling `hello()` from `main()`, this program launches a whole new thread to do it, bringing the thread

count to two—the initial thread that starts at `main()` and the new thread that starts at `hello()`.

After the new thread has been launched #3, the initial thread continues execution. If it didn't wait for the new thread to finish, it would merrily continue to the end of `main()` and thus end the program—possibly before the new thread had had a chance to run. This is why the call to `join()` is there #4—as described in chapter 2, this causes the calling thread (in `main()`) to wait for the thread associated with the `std::thread` object, in this case, `t`.

If this seems like a lot of work to go to just to write a message to standard output, it is—as described previously in section 1.2.3, it's generally not worth the effort to use multiple threads for such a simple task, especially if the initial thread has nothing to do in the meantime. Later in the book, we'll work through examples that show scenarios where there's a clear gain to using multiple threads.

1.5 Summary

In this chapter, I covered what is meant by concurrency and multithreading and why you'd choose to use it (or not) in your applications. I also covered the history of multithreading in C++ from the complete lack of support in the 1998 standard, through various platform-specific extensions, to proper multithreading support in the C++11 Standard, and on to the C++14 and C++17 standards and the Technical Specification for Concurrency. This support has come just in time to allow programmers to take advantage of the greater hardware concurrency becoming available with newer CPUs, as chip manufacturers choose to add more processing power in the form of multiple cores that allow more tasks to be executed concurrently, rather than increasing the execution speed of a single core.

I also showed how simple using the classes and functions from the C++ Standard Library can be, in the examples in section 1.4. In C++, using multiple threads isn't complicated in and of itself; the complexity lies in designing the code so that it behaves as intended.

After the taster examples of section 1.4, it's time for something with a bit more substance. In chapter 2 we'll look at the classes and functions available for managing threads.

2

Managing threads

This chapter covers

- Starting threads, and various ways of specifying code to run on a new thread
- Waiting for a thread to finish versus leaving it to run
- Uniquely identifying threads

OK, so you've decided to use concurrency for your application. In particular, you've decided to use multiple threads. What now? How do you launch these threads, how do you check that they've finished, and how do you keep tabs on them? The C++ Standard Library makes most thread-management tasks relatively easy, with just about everything managed through the `std::thread` object associated with a given thread, as you'll see. For those tasks that aren't so straightforward, the library provides the flexibility to build what you need from the basic building blocks.

In this chapter, I'll start by covering the basics: launching a thread, waiting for it to finish, or running it in the background. We'll then proceed to look at passing additional parameters to the thread function when it's launched and how to transfer ownership of a thread from one `std::thread` object to another. Finally, we'll look at choosing the number of threads to use and identifying particular threads.

2.1 Basic thread management

Every C++ program has at least one thread, which is started by the C++ runtime: the thread running `main()`. Your program can then launch additional threads that have another function as the entry point. These threads then run concurrently with each other and with the initial thread. Just as the program exits when the program returns from `main()`, when the specified entry point function returns, the thread exits. As you'll see, if you have a

`std::thread` object for a thread, you can wait for it to finish; but first you have to start it, so let's look at launching threads.

2.1.1 Launching a thread

As you saw in chapter 1, threads are started by constructing a `std::thread` object that specifies the task to run on that thread. In the simplest case, that task is just a plain, ordinary `void`-returning function that takes no parameters. This function runs on its own thread until it returns, and then the thread stops. At the other extreme, the task could be a function object that takes additional parameters and performs a series of independent operations that are specified through some kind of messaging system while it's running, and the thread stops only when it's signaled to do so, again via some kind of messaging system. It doesn't matter what the thread is going to do or where it's launched from, but starting a thread using the C++ Thread Library always boils down to constructing a `std::thread` object:

```
void do_some_work();
std::thread my_thread(do_some_work);
```

This is just about as simple as it gets. Of course, you have to make sure that the `<thread>` header is included so the compiler can see the definition of the `std::thread` class. As with much of the C++ Standard Library, `std::thread` works with any *callable* type, so you can pass an instance of a class with a function call operator to the `std::thread` constructor instead:

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};
background_task f;
std::thread my_thread(f);
```

In this case, the supplied function object is *copied* into the storage belonging to the newly created thread of execution and invoked from there. It's therefore essential that the copy behave equivalently to the original, or the result may not be what's expected.

One thing to consider when passing a function object to the thread constructor is to avoid what is dubbed "C++'s most vexing parse." If you pass a temporary rather than a named variable, then the syntax can be the same as that of a function declaration, in which case the compiler interprets it as such, rather than an object definition. For example,

```
std::thread my_thread(background_task());
```

declares a function `my_thread` that takes a single parameter (of type pointer to a function taking no parameters and returning a `background_task` object) and returns a `std::thread` object, rather than launching a new thread. You can avoid this by naming your function object

as shown previously, by using an extra set of parentheses, or by using the new uniform initialization syntax, for example:

```
std::thread my_thread((background_task()));      #1  
std::thread my_thread{background_task()};        #2
```

In the first example #1, the extra parentheses prevent interpretation as a function declaration, thus allowing `my_thread` to be declared as a variable of type `std::thread`. The second example #2 uses the new uniform initialization syntax with braces rather than parentheses, and thus would also declare a variable.

One type of callable object that avoids this problem is a *lambda expression*. This is a new feature from C++11 which essentially allows you to write a local function, possibly capturing some local variables and avoiding the need of passing additional arguments (see section 2.2). For full details on lambda expressions, see appendix A, section A.5. The previous example can be written using a lambda expression as follows:

```
std::thread my_thread([]  
    do_something();  
    do_something_else();  
});
```

Once you've started your thread, you need to explicitly decide whether to wait for it to finish (by joining with it—see section 2.1.2) or leave it to run on its own (by detaching it—see section 2.1.3). If you don't decide before the `std::thread` object is destroyed, then your program is terminated (the `std::thread` destructor calls `std::terminate()`). It's therefore imperative that you ensure that the thread is correctly joined or detached, even in the presence of exceptions. See section 2.1.3 for a technique to handle this scenario. Note that you only have to make this decision before the `std::thread` object is destroyed—the thread itself may well have finished long before you join with it or detach it, and if you detach it, then if the thread is still running, it will continue to do so, and may continue running long after the `std::thread` object is destroyed; it will only stop running when it finally returns from the thread function.

If you don't wait for your thread to finish, then you need to ensure that the data accessed by the thread is valid until the thread has finished with it. This isn't a new problem—even in single-threaded code it is undefined behavior to access an object after it's been destroyed—but the use of threads provides an additional opportunity to encounter such lifetime issues.

One situation in which you can encounter such problems is when the thread function holds pointers or references to local variables and the thread hasn't finished when the function exits. The following listing shows an example of just such a scenario.

Listing 2.1 A function that returns while a thread still has access to local variables

```
struct func  
{  
    int& i;  
    func(int& i_):i(i_){}  
    void operator()()
```

```

{
    for(unsigned j=0;j<10000000;++j)
    {
        do_something(i);           #1
    }
}
};

void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();          #2
}                                #3

```

#1 Potential access to dangling reference

#2 Don't wait for thread to finish

#3 New thread might still be running

In this case, the new thread associated with `my_thread` will probably still be running when `oops` exits #3, because you've explicitly decided not to wait for it by calling `detach()`. #2. If the thread *is* still running, then we have the scenario shown in figure 2.1: the next call to `do_something(i)` #1 will access an already destroyed variable. This is just like normal single-threaded code—allowing a pointer or reference to a local variable to persist beyond the function exit is never a good idea—but it's easier to make the mistake with multithreaded code, because it isn't necessarily immediately apparent that this has happened.

Main thread	New thread
Construct <code>my_func</code> with reference to <code>some_local_state</code>	
Start new thread <code>my_thread</code>	
	<i>Started</i>
	Calls <code>func::operator()</code>
Detaches <code>my_thread</code>	Running <code>func::operator()</code> ; may call <code>do_something</code> with reference to <code>some_local_state</code>
Destroys <code>some_local_state</code>	Still running
Exits <code>oops</code>	Still running <code>func::operator()</code> ; may call <code>do_something</code> with reference to <code>some_local_state</code> => undefined behavior

Figure 2.1: Accessing a local variable with a detached thread after it has been destroyed

One common way to handle this scenario is to make the thread function self-contained and copy the data into the thread rather than sharing the data. If you use a callable object for your thread function, that object is itself copied into the thread, so the original object can be destroyed immediately. But you still need to be wary of objects containing pointers or references, such as that from listing 2.1. In particular, it's a bad idea to create a thread within a function that has access to the local variables in that function, unless the thread is guaranteed to finish before the function exits.

Alternatively, you can ensure that the thread has completed execution before the function exits by *joining* with the thread.

2.1.2 Waiting for a thread to complete

If you need to wait for a thread to complete, you can do this by calling `join()` on the associated `std::thread` instance. In the case of listing 2.1, replacing the call to `my_thread.detach()` before the closing brace of the function body with a call to `my_thread.join()` would therefore be sufficient to ensure that the thread was finished before the function was exited and thus before the local variables were destroyed. In this case, it would mean there was little point running the function on a separate thread, because the first thread wouldn't be doing anything useful in the meantime, but in real code the original thread would either have work to do itself or it would have launched several threads to do useful work before waiting for all of them to complete.

`join()` is simple and brute force—either you wait for a thread to finish or you don't. If you need more fine-grained control over waiting for a thread, such as to check whether a thread is finished, or to wait only a certain period of time, then you have to use alternative mechanisms such as condition variables and futures, which we'll look at in chapter 4. The act of calling `join()` also cleans up any storage associated with the thread, so the `std::thread` object is no longer associated with the now-finished thread; it isn't associated with any thread. This means that you can call `join()` only once for a given thread; once you've called `join()`, the `std::thread` object is no longer joinable, and `joinable()` will return `false`.

2.1.3 Waiting in exceptional circumstances

As mentioned earlier, you need to ensure that you've called either `join()` or `detach()` before a `std::thread` object is destroyed. If you're detaching a thread, you can usually call `detach()` immediately after the thread has been started, so this isn't a problem. But if you're intending to wait for the thread, you need to pick carefully the place in the code where you call `join()`. This means that the call to `join()` is liable to be skipped if an exception is thrown after the thread has been started but before the call to `join()`.

To avoid your application being terminated when an exception is thrown, you therefore need to make a decision on what to do in this case. In general, if you were intending to call `join()` in the non-exceptional case, you also need to call `join()` in the presence of an

exception to avoid accidental lifetime problems. The next listing shows some simple code that does just that.

Listing 2.2 Waiting for a thread to finish

```
struct func;           #A
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();          #1
        throw;
    }
    t.join();            #2
}
```

#A See definition in listing 2.1

The code in listing 2.2 uses a `try/catch` block to ensure that a thread with access to local state is finished before the function exits, whether the function exits normally #2 or by an exception #1. The use of `try/catch` blocks is verbose, and it's easy to get the scope slightly wrong, so this isn't an ideal scenario. If it's important to ensure that the thread must complete before the function exits—whether because it has a reference to other local variables or for any other reason—then it's important to ensure this is the case for all possible exit paths, whether normal or exceptional, and it's desirable to provide a simple, concise mechanism for doing so.

One way of doing this is to use the standard Resource Acquisition Is Initialization (RAII) idiom and provide a class that does the `join()` in its destructor, as in the following listing. See how it simplifies the function `f()`.

Listing 2.3 Using RAII to wait for a thread to complete

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):
        t(t_)
    {}
    ~thread_guard()
    {
        if(t.joinable())          #1
        {
            t.join();            #2
        }
    }
}
```

```

    thread_guard(thread_guard const&)=delete;           #3
    thread_guard& operator=(thread_guard const&)=delete;
};

struct func;                                #A
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);
    do_something_in_current_thread();
}
#4

```

#A See definition in listing 2.1

When the execution of the current thread reaches the end of `f` #4, the local objects are destroyed in reverse order of construction. Consequently, the `thread_guard` object `g` is destroyed first, and the thread is joined with in the destructor #2. This even happens if the function exits because `do_something_in_current_thread` throws an exception.

The destructor of `thread_guard` in listing 2.3 first tests to see if the `std::thread` object is `joinable()` #1 before calling `join()` #2. This is important, because `join()` can be called only once for a given thread of execution, so it would therefore be a mistake to do so if the thread had already been joined.

The copy constructor and copy-assignment operator are marked `=delete` #3 to ensure that they're not automatically provided by the compiler. Copying or assigning such an object would be dangerous, because it might then outlive the scope of the thread it was joining. By declaring them as deleted, any attempt to copy a `thread_guard` object will generate a compilation error. See appendix A, section A.2, for more about deleted functions.

If you don't need to wait for a thread to finish, you can avoid this exception-safety issue by *detaching* it. This breaks the association of the thread with the `std::thread` object and ensures that `std::terminate()` won't be called when the `std::thread` object is destroyed, even though the thread is still running in the background.

2.1.4 Running threads in the background

Calling `detach()` on a `std::thread` object leaves the thread to run in the background, with no direct means of communicating with it. It's no longer possible to wait for that thread to complete; if a thread becomes detached, it isn't possible to obtain a `std::thread` object that references it, so it can no longer be joined. Detached threads truly run in the background; ownership and control are passed over to the C++ Runtime Library, which ensures that the resources associated with the thread are correctly reclaimed when the thread exits.

Detached threads are often called *daemon threads* after the UNIX concept of a *daemon process* that runs in the background without any explicit user interface. Such threads are typically long-running; they may well run for almost the entire lifetime of the application, performing a background task such as monitoring the filesystem, clearing unused entries out of object caches, or optimizing data structures. At the other extreme, it may make sense to

use a detached thread where there's another mechanism for identifying when the thread has completed or where the thread is used for a "fire and forget" task.

As you've already seen in section 2.1.2, you detach a thread by calling the `detach()` member function of the `std::thread` object. After the call completes, the `std::thread` object is no longer associated with the actual thread of execution and is therefore no longer joinable:

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
```

In order to detach the thread from a `std::thread` object, there must be a thread to detach: you can't call `detach()` on a `std::thread` object with no associated thread of execution. This is exactly the same requirement as for `join()`, and you can check it in exactly the same way—you can only call `t.detach()` for a `std::thread` object `t` when `t.joinable()` returns `true`.

Consider an application such as a word processor that can edit multiple documents at once. There are many ways to handle this, both at the UI level and internally. One way that seems to be increasingly common at the moment is to have multiple independent top-level windows, one for each document being edited. Although these windows appear to be completely independent, each with its own menus and so forth, they're running within the same instance of the application. One way to handle this internally is to run each document-editing window in its own thread; each thread runs the same code but with different data relating to the document being edited and the corresponding window properties. Opening a new document therefore requires starting a new thread. The thread handling the request isn't going to care about waiting for that other thread to finish, because it's working on an unrelated document, so this makes it a prime candidate for running a detached thread.

The following listing shows a simple code outline for this approach.

Listing 2.4 Detaching a thread to handle other documents

```
void edit_document(std::string const& filename)
{
    open_document_and_display_gui(filename);
    while(!done_editing())
    {
        user_command cmd=get_user_input();
        if(cmd.type==open_new_document)
        {
            std::string const new_name=get_filename_from_user();
            std::thread t(edit_document,new_name);           #1
            t.detach();                                     #2
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
```

If the user chooses to open a new document, you prompt them for the document to open, start a new thread to open that document #1, and then detach it #2. Because the new thread is doing the same operation as the current thread but on a different file, you can reuse the same function (`edit_document`) with the newly chosen filename as the supplied argument.

This example also shows a case where it's helpful to pass arguments to the function used to start a thread: rather than just passing the name of the function to the `std::thread` constructor #1, you also pass in the filename parameter. Although other mechanisms could be used to do this, such as using a function object with member data instead of an ordinary function with parameters, the Thread Library provides you with an easy way of doing it.

2.2 Passing arguments to a thread function

As shown in listing 2.4, passing arguments to the callable object or function is fundamentally as simple as passing additional arguments to the `std::thread` constructor. But it's important to bear in mind that by default the arguments are *copied* into internal storage, where they can be accessed by the newly created thread of execution, and then passed to the callable object or function as rvalues as if they were temporaries. This is done even if the corresponding parameter in the function is expecting a reference. Here's a simple example:

```
void f(int i,std::string const& s);
std::thread t(f,3,"hello");
```

This creates a new thread of execution associated with `t`, which calls `f(3,"hello")`. Note that even though `f` takes a `std::string` as the second parameter, the string literal is passed as a `char const*` and converted to a `std::string` only in the context of the new thread. This is particularly important when the argument supplied is a pointer to an automatic variable, as follows:

```
void f(int i,std::string const& s);
void oops(int some_param)
{
    char buffer[1024];           #1
    sprintf(buffer, "%i",some_param);
    std::thread t(f,3,buffer);    #2
    t.detach();
}
```

In this case, it's the pointer to the local variable `buffer` #1 that's passed through to the new thread #2, and there's a significant chance that the function `oops` will exit before the buffer has been converted to a `std::string` on the new thread, thus leading to undefined behavior. The solution is to cast to `std::string` *before* passing the buffer to the `std::thread` constructor:

```
void f(int i,std::string const& s);
void not_oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,std::string(buffer));   #A
}
```

```
    t.detach();
}
```

#A Using std::string avoids dangling pointer

In this case, the problem is that you were relying on the implicit conversion of the pointer to the buffer into the `std::string` object expected as a function parameter, but this conversion happens too late because the `std::thread` constructor copies the supplied values as is, without converting to the expected argument type.

It's not possible to get the reverse scenario: the object is copied, and what you wanted was a non-`const` reference, since this won't compile. You might try and do this if the thread is updating a data structure that's passed in by reference, for example:

```
void update_data_for_widget(widget_id w,widget_data& data);      #1
void oops_again(widget_id w)
{
    widget_data data;
    std::thread t(update_data_for_widget,w,data);                  #2
    display_status();
    t.join();
    process_widget_data(data);
}
```

Although `update_data_for_widget` #1 expects the second parameter to be passed by reference, the `std::thread` constructor #2 doesn't know that; it's oblivious to the types of the arguments expected by the function and blindly copies the supplied values. However, the internal code passes copied arguments as rvalues in order to work with move-only types, and will thus try to call `update_data_for_widget` with an rvalue. This will therefore fail to compile as you can't pass an rvalue to a function expecting a non-`const` reference. For those of you familiar with `std::bind`, the solution will be readily apparent: you need to wrap the arguments that really need to be references in `std::ref`. In this case, if you change the thread invocation to

```
std::thread t(update_data_for_widget,w,std::ref(data));
```

and then `update_data_for_widget` will be correctly passed a reference to `data` rather than a temporary *copy* of `data`, and the code will now compile successfully.

If you're familiar with `std::bind`, the parameter-passing semantics will be unsurprising, because both the operation of the `std::thread` constructor and the operation of `std::bind` are defined in terms of the same mechanism. This means that, for example, you can pass a member function pointer as the function, provided you supply a suitable object pointer as the first argument:

```
class X
{
public:
    void do_lengthy_work();
};

X my_x;
std::thread t(&X::do_lengthy_work,&my_x);          #1
```

This code will invoke `my_x.do_lengthy_work()` on the new thread, because the address of `my_x` is supplied as the object pointer #1. You can also supply arguments to such a member function call: the third argument to the `std::thread` constructor will be the first argument to the member function and so forth.

Another interesting scenario for supplying arguments is where the arguments can't be copied but can only be *moved*: the data held within one object is transferred over to another, leaving the original object "empty." An example of such a type is `std::unique_ptr`, which provides automatic memory management for dynamically allocated objects. Only one `std::unique_ptr` instance can point to a given object at a time, and when that instance is destroyed, the pointed-to object is deleted. The *move constructor* and *move assignment operator* allow the ownership of an object to be transferred around between `std::unique_ptr` instances (see appendix A, section A.1.1, for more on move semantics). Such a transfer leaves the source object with a `NULL` pointer. This moving of values allows objects of this type to be accepted as function parameters or returned from functions. Where the source object is a temporary, the move is automatic, but where the source is a named value, the transfer must be requested directly by invoking `std::move()`. The following example shows the use of `std::move` to transfer ownership of a dynamic object into a thread:

```
void process_big_object(std::unique_ptr<big_object>);  
std::unique_ptr<big_object> p(new big_object);  
p->prepare_data(42);  
std::thread t(process_big_object, std::move(p));
```

By specifying `std::move(p)` in the `std::thread` constructor, the ownership of the `big_object` is transferred first into internal storage for the newly created thread and then into `process_big_object`.

Several of the classes in the Standard Thread Library exhibit the same ownership semantics as `std::unique_ptr`, and `std::thread` is one of them. Though `std::thread` instances don't own a dynamic object in the same way as `std::unique_ptr` does, they do own a resource: each instance is responsible for managing a thread of execution. This ownership can be transferred between instances, because instances of `std::thread` are *movable*, even though they aren't *copyable*. This ensures that only one object is associated with a particular thread of execution at any one time while allowing programmers the option of transferring that ownership between objects.

2.3 Transferring ownership of a thread

Suppose you want to write a function that creates a thread to run in the background but passes back ownership of the new thread to the calling function rather than waiting for it to complete, or maybe you want to do the reverse: create a thread and pass ownership in to some function that should wait for it to complete. In either case, you need to transfer ownership from one place to another.

This is where the move support of `std::thread` comes in. As described in the previous section, many resource-owning types in the C++ Standard Library such as `std::ifstream`

and `std::unique_ptr` are *movable* but not *copyable*, and `std::thread` is one of them. This means that the ownership of a particular thread of execution can be moved between `std::thread` instances, as in the following example. The example shows the creation of two threads of execution and the transfer of ownership of those threads among three `std::thread` instances, `t1`, `t2`, and `t3`:

```
void some_function();
void some_other_function();
std::thread t1(some_function);          #1
std::thread t2=std::move(t1);           #2
t1=std::thread(some_other_function);    #3
std::thread t3;                      #4
t3=std::move(t2);                   #5
t1=std::move(t3);                   #6
```

#6 This assignment will terminate program!

First, a new thread is started #1 and associated with `t1`. Ownership is then transferred over to `t2` when `t2` is constructed, by invoking `std::move()` to explicitly move ownership #2. At this point, `t1` no longer has an associated thread of execution; the thread running `some_function` is now associated with `t2`.

Then, a new thread is started and associated with a temporary `std::thread` object #3. The subsequent transfer of ownership into `t1` doesn't require a call to `std::move()` to explicitly move ownership, because the owner is a temporary object—moving from temporaries is automatic and implicit.

`t3` is default constructed #4, which means that it's created without any associated thread of execution. Ownership of the thread currently associated with `t2` is transferred into `t3` #5, again with an explicit call to `std::move()`, because `t2` is a named object. After all these moves, `t1` is associated with the thread running `some_other_function`, `t2` has no associated thread, and `t3` is associated with the thread running `some_function`.

The final move #6 transfers ownership of the thread running `some_function` back to `t1` where it started. But in this case `t1` already had an associated thread (which was running `some_other_function`), so `std::terminate()` is called to terminate the program. This is done for consistency with the `std::thread` destructor. You saw in section 2.1.1 that you must explicitly wait for a thread to complete or detach it before destruction, and the same applies to assignment: you can't just "drop" a thread by assigning a new value to the `std::thread` object that manages it.

The move support in `std::thread` means that ownership can readily be transferred out of a function, as shown in the following listing.

Listing 2.5 Returning a `std::thread` from a function

```
std::thread f()
{
    void some_function();
    return std::thread(some_function);
}
```

```

std::thread g()
{
    void some_other_function(int);
    std::thread t(some_other_function,42);
    return t;
}

```

Likewise, if ownership should be transferred into a function, it can just accept an instance of `std::thread` by value as one of the parameters, as shown here:

```

void f(std::thread t);
void g()
{
    void some_function();
    f(std::thread(some_function));
    std::thread t(some_function);
    f(std::move(t));
}

```

One benefit of the move support of `std::thread` is that you can build on the `thread_guard` class from listing 2.3 and have it actually take ownership of the thread. This avoids any unpleasant consequences should the `thread_guard` object outlive the thread it was referencing, and it also means that no one else can join or detach the thread once ownership has been transferred into the object. Because this would primarily be aimed at ensuring threads are completed before a scope is exited, I named this class `scoped_thread`. The implementation is shown in the following listing, along with a simple example.

Listing 2.6 `scoped_thread` and example usage

```

class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_):           #1
        t(std::move(t_))
    {
        if(!t.joinable())                           #2
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();          #3
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};
struct func;                      #A
void f()
{
    int some_local_state;
    scoped_thread t{std::thread(func(some_local_state))};  #4
    do_something_in_current_thread();                      #5
}

```

#A See listing 2.1

The example is similar to that from listing 2.3, but the new thread is passed in directly to the `scoped_thread` #4 rather than having to create a separate named variable for it. When the initial thread reaches the end of `f` #5, the `scoped_thread` object is destroyed and then joins with #3 the thread supplied to the constructor #1. Whereas with the `thread_guard` class from listing 2.3 the destructor had to check that the thread was still joinable, you can do that in the constructor #2 and throw an exception if it's not.

One of the proposals for C++17 was for a `joining_thread` class that would be similar to `std::thread`, except automatically join in the destructor much like `scoped_thread` does. This didn't get consensus in the committee, so was not accepted into the standard, but is relatively easy to write. One possible implementation is shown in listing 2.7.

Listing 2.7 A `joining_thread` class

```
class joining_thread
{
    std::thread t;
public:
    joining_thread() noexcept=default;
    template<typename Callable,typename ... Args>
    explicit joining_thread(Callable&& func,Args&& ... args):
        t(std::forward<Callable>(func),std::forward<Args>(args)...)
    {}
    explicit joining_thread(std::thread t_) noexcept:
        t(std::move(t_))
    {}
    joining_thread(joining_thread&& other) noexcept:
        t(std::move(other.t))
    {}
    joining_thread& operator=(joining_thread&& other) noexcept
    {
        if(joinable())
            join();
        t=std::move(other.t);
        return *this;
    }
    joining_thread& operator=(std::thread other) noexcept
    {
        if(joinable())
            join();
        t=std::move(other);
        return *this;
    }
    ~joining_thread() noexcept
    {
        if(joinable())
            join();
    }
}
```

```

        join();
    }
    void swap(joining_thread& other) noexcept
    {
        t.swap(other.t);
    }
    std::thread::id get_id() const noexcept{
        return t.get_id();
    }
    bool joinable() const noexcept
    {
        return t.joinable();
    }
    void join()
    {
        t.join();
    }
    void detach()
    {
        t.detach();
    }
    std::thread& as_thread() noexcept
    {
        return t;
    }
    const std::thread& as_thread() const noexcept
    {
        return t;
    }
};


```

The move support in `std::thread` also allows for containers of `std::thread` objects, if those containers are move aware (like the updated `std::vector<>`). This means that you can write code like that in the following listing, which spawns a number of threads and then waits for them to finish.

Listing 2.8 Spawn some threads and wait for them to finish

```

void do_work(unsigned id);
void f()
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)
    {
        threads.push_back(std::thread(do_work,i));    #A
    }
    std::for_each(threads.begin(),threads.end(),
                 std::mem_fn(&std::thread::join));    #B
}


```

```
}
```

#A Spawn threads
#B Call join() on each thread in turn

If the threads are being used to subdivide the work of an algorithm, this is often just what's required; before returning to the caller, all threads must have finished. Of course, the simple structure of listing 2.8 implies that the work done by the threads is self-contained, and the result of their operations is purely the side effects on shared data. If `f()` were to return a value to the caller that depended on the results of the operations performed by these threads, then as written this return value would have to be determined by examining the shared data after the threads had terminated. Alternative schemes for transferring the results of operations between threads are discussed in chapter 4.

Putting `std::thread` objects in a `std::vector` is a step toward automating the management of those threads: rather than creating separate variables for those threads and joining with them directly, they can be treated as a group. You can take this a step further by creating a dynamic number of threads determined at runtime, rather than creating a fixed number as in listing 2.8.

2.4 Choosing the number of threads at runtime

One feature of the C++ Standard Library that helps here is `std::thread::hardware_concurrency()`. This function returns an indication of the number of threads that can truly run concurrently for a given execution of a program. On a multicore system it might be the number of CPU cores, for example. This is only a hint, and the function might return 0 if this information is not available, but it can be a useful guide for splitting a task among threads.

Listing 2.9 shows a simple implementation of a parallel version of `std::accumulate`. In real code you'll probably want to use the parallel version of `std::reduce` described in chapter 10, rather than implementing it yourself, but this serves to show some basic ideas. It divides the work among the threads, with a minimum number of elements per thread in order to avoid the overhead of too many threads. Note that this implementation assumes that none of the operations will throw an exception, even though exceptions are possible; the `std::thread` constructor will throw if it can't start a new thread of execution, for example. Handling exceptions in such an algorithm is beyond the scope of this simple example and will be covered in chapter 8.

Listing 2.9 A naïve parallel version of `std::accumulate`

```
template<typename Iterator,typename T>
struct accumulate_block
{
    void operator()(Iterator first,Iterator last,T& result)
    {
        result=std::accumulate(first,last,result);
    }
};
```

```
template<typename Iterator,typename T>
```

```

T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)                                     #1
        return init;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;    #2
    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();
    unsigned long const num_threads=                 #3
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;      #4
    std::vector<T> results(num_threads);
    std::vector<std::thread> threads(num_threads-1);       #5
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);                #6
        threads[i]=std::thread(                         #7
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;                            #8
    }
    accumulate_block<Iterator,T>()
        block_start,last,results[num_threads-1]);      #9
    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));             @10
    return std::accumulate(results.begin(),results.end(),init);  #11
}

```

Although this is quite a long function, it's actually straightforward. If the input range is empty #1, you just return the initial value `init`. Otherwise, there's at least one element in the range, so you can divide the number of elements to process by the minimum block size in order to give the maximum number of threads #2. This is to avoid creating 32 threads on a 32-core machine when you have only five values in the range.

The number of threads to run is the minimum of your calculated maximum and the number of hardware threads #3. You don't want to run more threads than the hardware can support (which is called *oversubscription*), because the context switching will mean that more threads will decrease the performance. If the call to `std::thread:: hardware_concurrency()` returned 0, you'd simply substitute a number of your choice; in this case I've chosen 2. You don't want to run too many threads, because that would slow things down on a single-core machine, but likewise you don't want to run too few, because then you'd be passing up the available concurrency.

The number of entries for each thread to process is the length of the range divided by the number of threads #4. If you're worrying about the case where the number doesn't divide evenly, don't—you'll handle that later.

Now that you know how many threads you have, you can create a `std::vector<T>` for the intermediate results and a `std::vector<std::thread>` for the threads #5. Note that you need to launch one fewer thread than `num_threads`, because you already have one.

Launching the threads is just a simple loop: advance the `block_end` iterator to the end of the current block #6 and launch a new thread to accumulate the results for this block #7. The start of the next block is the end of this one #8.

After you've launched all the threads, this thread can then process the final block #9. This is where you take account of any uneven division: you know the end of the final block must be last, and it doesn't matter how many elements are in that block.

Once you've accumulated the results for the last block, you can wait for all the threads you spawned with `std::for_each` #10, as in listing 2.8, and then add up the results with a final call to `std::accumulate` #11.

Before you leave this example, it's worth pointing out that where the addition operator for the type `T` is not associative (such as for `float` or `double`), the results of this `parallel_accumulate` may vary from those of `std::accumulate`, because of the grouping of the range into blocks. Also, the requirements on the iterators are slightly more stringent: they must be at least *forward iterators*, whereas `std::accumulate` can work with single-pass *input iterators*, and `T` must be *default constructible* so that you can create the `results` vector. These sorts of requirement changes are common with parallel algorithms; by their very nature they're different in some manner in order to make them parallel, and this has consequences on the results and requirements. Implementing parallel algorithms is covered in more depth in chapter 8, and the standard supplied ones from C++17 (the equivalent to the `parallel_accumulate` described here being the parallel form of `std::reduce`). It's also worth noting that because you can't return a value directly from a thread, you must pass in a reference to the relevant entry in the `results` vector. Alternative ways of returning results from threads are addressed through the use of *futures* in chapter 4.

In this case, all the information required by each thread was passed in when the thread was started, including the location in which to store the result of its calculation. This isn't always the case: sometimes it's necessary to be able to identify the threads in some way for part of the processing. You could pass in an identifying number, such as the value of `i` in listing 2.8, but if the function that needs the identifier is several levels deep in the call stack and could be called from any thread, it's inconvenient to have to do it that way. When we were designing the C++ Thread Library we foresaw this need, and so each thread has a unique identifier.

2.5 Identifying threads

Thread identifiers are of type `std::thread::id` and can be retrieved in two ways. First, the identifier for a thread can be obtained from its associated `std::thread` object by calling the `get_id()` member function. If the `std::thread` object doesn't have an associated thread of execution, the call to `get_id()` returns a default-constructed `std::thread::id` object, which indicates "not any thread." Alternatively, the identifier for the current thread can be obtained by calling `std::this_thread::get_id()`, which is also defined in the `<thread>` header.

Objects of type `std::thread::id` can be freely copied and compared; they wouldn't be of much use as identifiers otherwise. If two objects of type `std::thread::id` are equal, they

represent the same thread, or both are holding the “not any thread” value. If two objects aren’t equal, they represent different threads, or one represents a thread and the other is holding the “not any thread” value.

The Thread Library doesn’t limit you to checking whether thread identifiers are the same or not; objects of type `std::thread::id` offer the complete set of comparison operators, which provide a total ordering for all distinct values. This allows them to be used as keys in associative containers, or sorted, or compared in any other way that you as a programmer may see fit. The comparison operators provide a total order for all non-equal values of `std::thread::id`, so they behave as you’d intuitively expect: if $a < b$ and $b < c$, then $a < c$, and so forth. The Standard Library also provides `std::hash<std::thread::id>` so that values of type `std::thread::id` can be used as keys in the new unordered associative containers too.

Instances of `std::thread::id` are often used to check whether a thread needs to perform some operation. For example, if threads are used to divide work as in listing 2.9, the initial thread that launched the others might need to perform its work slightly differently in the middle of the algorithm. In this case it could store the result of `std::this_thread::get_id()` before launching the other threads, and then the core part of the algorithm (which is common to all threads) could check its own thread ID against the stored value:

```
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
    if(std::this_thread::get_id()==master_thread)
    {
        do_master_thread_work();
    }
    do_common_work();
}
```

Alternatively, the `std::thread::id` of the current thread could be stored in a data structure as part of an operation. Later operations on that same data structure could then check the stored ID against the ID of the thread performing the operation to determine what operations are permitted/required.

Similarly, thread IDs could be used as keys into associative containers where specific data needs to be associated with a thread and alternative mechanisms such as thread-local storage aren’t appropriate. Such a container could, for example, be used by a controlling thread to store information about each of the threads under its control or for passing information between threads.

The idea is that `std::thread::id` will suffice as a generic identifier for a thread in most circumstances; it’s only if the identifier has semantic meaning associated with it (such as being an index into an array) that alternatives should be necessary. You can even write out an instance of `std::thread::id` to an output stream such as `std::cout`:

```
std::cout<<std::this_thread::get_id();
```

The exact output you get is strictly implementation dependent; the only guarantee given by the standard is that thread IDs that compare as equal should produce the same output, and

those that are not equal should give different output. This is therefore primarily useful for debugging and logging, but the values have no semantic meaning, so there's not much more that could be said anyway.

2.6 Summary

In this chapter I covered the basics of thread management with the C++ Standard Library: starting threads, waiting for them to finish, and *not* waiting for them to finish because you want them to run in the background. You also saw how to pass arguments into the thread function when a thread is started, how to transfer the responsibility for managing a thread from one part of the code to another, and how groups of threads can be used to divide work. Finally, I discussed identifying threads in order to associate data or behavior with specific threads that's inconvenient to associate through alternative means. Although you can do quite a lot with purely independent threads that each operate on separate data, as in listing 2.9 for example, sometimes it's desirable to share data among threads while they're running. Chapter 3 discusses the issues surrounding sharing data directly among threads, while chapter 4 covers more general issues surrounding synchronizing operations with and without shared data.

3

Sharing data between threads

This chapter covers

- Problems with sharing data between threads
- Protecting data with mutexes
- Alternative facilities for protecting shared data

One of the key benefits of using threads for concurrency is the potential to easily and directly share data between them, so now that we've covered starting and managing threads, let's look at the issues surrounding shared data.

Imagine for a moment that you're sharing an apartment with a friend. There's only one kitchen and only one bathroom. Unless you're particularly friendly, you can't both use the bathroom at the same time, and if your roommate occupies the bathroom for a long time, it can be frustrating if you need to use it. Likewise, though it might be possible to both cook meals at the same time, if you have a combined oven and grill, it's just not going to end well if one of you tries to grill some sausages at the same time as the other is baking a cake. Furthermore, we all know the frustration of sharing a space and getting halfway through a task only to find that someone has borrowed something we need or changed something from the way we left it.

It's the same with threads. If you're sharing data between threads, you need to have rules for which thread can access which bit of data when, and how any updates are communicated to the other threads that care about that data. The ease with which data can be shared between multiple threads in a single process is not just a benefit—it can also be a big drawback. Incorrect use of shared data is one of the biggest causes of concurrency-related bugs, and the consequences can be far worse than sausage-flavored cakes.

This chapter is about sharing data safely between threads in C++, avoiding the potential problems that can arise, and maximizing the benefits.

3.1 Problems with sharing data between threads

When it comes down to it, the problems with sharing data between threads are all due to the consequences of modifying data. *If all shared data is read-only, there's no problem, because the data read by one thread is unaffected by whether or not another thread is reading the same data.* However, if data is shared between threads, and one or more threads start modifying the data, there's a lot of potential for trouble. In this case, you must take care to ensure that everything works out OK.

One concept that's widely used to help programmers reason about their code is that of *invariants*—statements that are always true about a particular data structure, such as “this variable contains the number of items in the list.” These invariants are often broken during an update, especially if the data structure is of any complexity or the update requires modification of more than one value.

Consider a doubly linked list, where each node holds a pointer to both the next node in the list and the previous one. One of the invariants is that if you follow a “next” pointer from one node (A) to another (B), the “previous” pointer from that node (B) points back to the first node (A). In order to remove a node from the list, the nodes on either side have to be updated to point to each other. Once one has been updated, the invariant is broken until the node on the other side has been updated too; after the update has completed, the invariant holds again.

The steps in deleting an entry from such a list are shown in figure 3.1:

1. Identify the node to delete (N).
2. Update the link from the node prior to N to point to the node after N.
3. Update the link from the node after N to point to the node prior to N.
4. Delete node N.

As you can see, between steps b and c, the links going in one direction are inconsistent with the links going in the opposite direction, and the invariant is broken.

The simplest potential problem with modifying data that's shared between threads is that of broken invariants. If you don't do anything special to ensure otherwise, if one thread is reading the doubly linked list while another is removing a node, it's quite possible for the reading thread to see the list with a node only partially removed (because only one of the links has been changed, as in step b of figure 3.1), so the invariant is broken. The consequences of this broken invariant can vary; if the other thread is just reading the list items from left to right in the diagram, it will skip the node being deleted. On the other hand, if the second thread is trying to delete the rightmost node in the diagram, it might end up permanently corrupting the data structure and eventually crashing the program. Whatever the outcome, this is an example of one of the most common causes of bugs in concurrent code: a *race condition*.

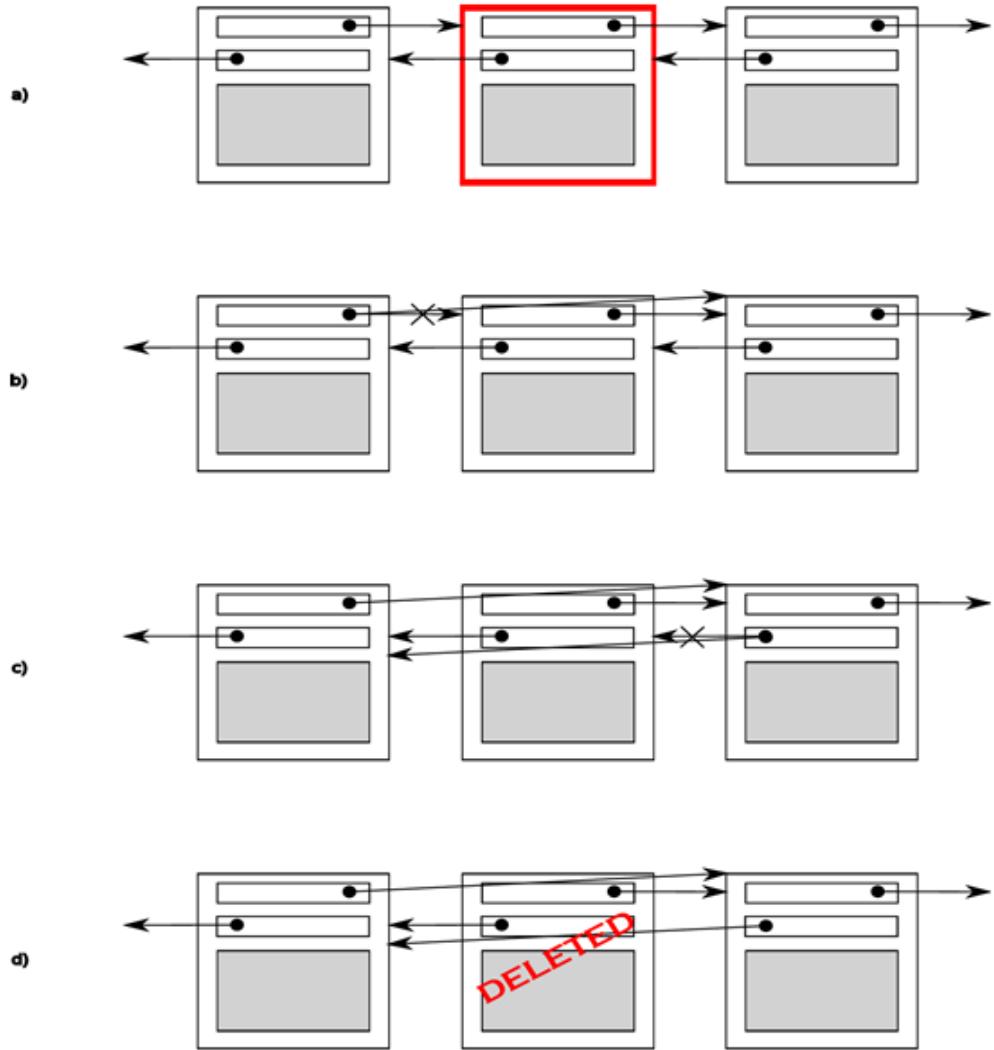


Figure 3.1 Deleting a node from a doubly linked list

3.1.1 Race conditions

Suppose you're buying tickets to see a movie at the cinema. If it's a big cinema, multiple cashiers will be taking money, so more than one person can buy tickets at the same time. If someone at another cashier's desk is also buying tickets for the same movie as you are, which seats are available for you to choose from depends on whether the other person actually books first or you do. If there are only a few seats left, this difference can be quite crucial: it

might literally be a race to see who gets the last tickets. This is an example of a *race condition*: which seats you get (or even whether you get tickets) depends on the relative ordering of the two purchases.

In concurrency, a race condition is anything where the outcome depends on the relative ordering of execution of operations on two or more threads; the threads race to perform their respective operations. Most of the time, this is quite benign because all possible outcomes are acceptable, even though they may change with different relative orderings. For example, if two threads are adding items to a queue for processing, it generally doesn't matter which item gets added first, provided that the invariants of the system are maintained. It's when the race condition leads to broken invariants that there's a problem, such as with the doubly linked list example just mentioned. When talking about concurrency, the term *race condition* is usually used to mean a *problematic* race condition; benign race conditions aren't so interesting and aren't a cause of bugs. The C++ Standard also defines the term *data race* to mean the specific type of race condition that arises because of concurrent modification to a single object (see section 5.1.2 for details); data races cause the dreaded *undefined behavior*.

Problematic race conditions typically occur where completing an operation requires modification of two or more distinct pieces of data, such as the two link pointers in the example. Because the operation must access two separate pieces of data, these must be modified in separate instructions, and another thread could potentially access the data structure when only one of them has been completed. Race conditions can often be hard to find and hard to duplicate because the window of opportunity is small. If the modifications are done as consecutive CPU instructions, the chance of the problem exhibiting on any one run-through is very small, even if the data structure is being accessed by another thread concurrently. As the load on the system increases, and the number of times the operation is performed increases, the chance of the problematic execution sequence occurring also increases. It's almost inevitable that such problems will show up at the most inconvenient time. Because race conditions are generally timing sensitive, they can often disappear entirely when the application is run under the debugger, because the debugger affects the timing of the program, even if only slightly.

If you're writing multithreaded programs, race conditions can easily be the bane of your life; a great deal of the complexity in writing software that uses concurrency comes from avoiding problematic race conditions.

3.1.2 Avoiding problematic race conditions

There are several ways to deal with problematic race conditions. The simplest option is to wrap your data structure with a protection mechanism, to ensure that only the thread actually performing a modification can see the intermediate states where the invariants are broken. From the point of view of other threads accessing that data structure, such modifications either haven't started or have completed. The C++ Standard Library provides several such mechanisms, which are described in this chapter.

Another option is to modify the design of your data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants. This is generally referred to as *lock-free programming* and is difficult to get right. If you're working at this level, the nuances of the memory model and identifying which threads can potentially see which set of values can get complicated. The memory model is covered in chapter 5, and lock-free programming is discussed in chapter 7.

Another way of dealing with race conditions is to handle the updates to the data structure as a *transaction*, just as updates to a database are done within a transaction. The required series of data modifications and reads is stored in a transaction log and then committed in a single step. If the commit can't proceed because the data structure has been modified by another thread, the transaction is restarted. This is termed *software transactional memory (STM)*, and it's an active research area at the time of writing. This won't be covered in this book, because there's no direct support for STM in C++ (though there is a Technical Specification for Transactional Memory Extensions to C++¹). However, the basic idea of doing something privately and then committing in a single step is something that I'll come back to later.

The most basic mechanism for protecting shared data provided by the C++ Standard is the *mutex*, so we'll look at that first.

3.2 Protecting shared data with mutexes

So, you have a shared data structure such as the linked list from the previous section, and you want to protect it from race conditions and the potential broken invariants that can ensue. Wouldn't it be nice if you could mark all the pieces of code that access the data structure as *mutually exclusive*, so that if any thread was running one of them, any other thread that tried to access that data structure had to wait until the first thread was finished? That would make it impossible for a thread to see a broken invariant except when it was the thread doing the modification.

Well, this isn't a fairy tale wish—it's precisely what you get if you use a synchronization primitive called a *mutex* (*mutual exclusion*). Before accessing a shared data structure, you *lock* the mutex associated with that data, and when you've finished accessing the data structure, you *unlock* the mutex. The Thread Library then ensures that once one thread has locked a specific mutex, all other threads that try to lock the same mutex have to wait until the thread that successfully locked the mutex unlocks it. This ensures that all threads see a self-consistent view of the shared data, without any broken invariants.

Mutexes are the most general of the data-protection mechanisms available in C++, but they're not a silver bullet; it's important to structure your code to protect the right data (see section 3.2.2) and avoid race conditions inherent in your interfaces (see section 3.2.3).

¹ ISO/IEC TS 19841:2015 – Technical Specification for C++ Extensions for Transactional Memory
http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=66343

Mutexes also come with their own problems, in the form of a *deadlock* (see section 3.2.4) and protecting either too much or too little data (see section 3.2.8). Let's start with the basics.

3.2.1 Using mutexes in C++

In C++, you create a mutex by constructing an instance of `std::mutex`, lock it with a call to the member function `lock()`, and unlock it with a call to the member function `unlock()`. However, it isn't recommended practice to call the member functions directly, because this means that you have to remember to call `unlock()` on every code path out of a function, including those due to exceptions. Instead, the Standard C++ Library provides the `std::lock_guard` class template, which implements that RAII idiom for a mutex; it locks the supplied mutex on construction and unlocks it on destruction, thus ensuring a locked mutex is always correctly unlocked. The following listing shows how to protect a list that can be accessed by multiple threads using a `std::mutex`, along with `std::lock_guard`. Both of these are declared in the `<mutex>` header.

Listing 3.1 Protecting a list with a mutex

```
#include <list>
#include <mutex>
#include <algorithm>
std::list<int> some_list;           #1
std::mutex some_mutex;             #2
void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex);      #3
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{
    std::lock_guard<std::mutex> guard(some_mutex);      #4
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

In listing 3.1, there's a single global variable #1, and it's protected with a corresponding global instance of `std::mutex` #2. The use of `std::lock_guard<std::mutex>` in `add_to_list()` #3 and again in `list_contains()` #4 means that the accesses in these functions are mutually exclusive: `list_contains()` will never see the list partway through a modification by `add_to_list()`.

C++17 has a new feature called class template argument deduction, which means that for simple class templates like `std::lock_guard`, the template argument list can often be omitted. #3 and #4 can thus be reduced to:

```
std::lock_guard guard(some_mutex);
```

on a C++17 compiler. For clarity of code, and compatibility with older compilers, I'll continue to specify the template arguments in other code snippets.

Although there are occasions where this use of global variables is appropriate, in the majority of cases it's common to group the mutex and the protected data together in a class rather than use global variables. This is a standard application of object-oriented design rules: by putting them in a class, you're clearly marking them as related, and you can encapsulate the functionality and enforce the protection. In this case, the functions `add_to_list` and `list_contains` would become member functions of the class, and the mutex and protected data would both become private members of the class, making it much easier to identify which code has access to the data and thus which code needs to lock the mutex. If all the member functions of the class lock the mutex before accessing any other data members and unlock it when done, the data is nicely protected from all comers.

Well, that's not *quite* true, as the astute among you will have noticed: if one of the member functions returns a pointer or reference to the protected data, then it doesn't matter that the member functions all lock the mutex in a nice orderly fashion, because you've just blown a big hole in the protection. *Any code that has access to that pointer or reference can now access (and potentially modify) the protected data without locking the mutex.* Protecting data with a mutex therefore requires careful interface design, to ensure that the mutex is locked before there's any access to the protected data and that there are no backdoors.

3.2.2 Structuring code for protecting shared data

As you've just seen, protecting data with a mutex is not quite as easy as just slapping a `std::lock_guard` object in every member function; one stray pointer or reference, and all that protection is for nothing. At one level, checking for stray pointers or references is easy; as long as none of the member functions return a pointer or reference to the protected data to their caller either via their return value or via an out parameter, the data is safe. If you dig a little deeper, it's not that straightforward—nothing ever is. As well as checking that the member functions don't pass out pointers or references to their callers, it's also important to check that they don't pass such pointers or references *in* to functions they call that aren't under your control. This is just as dangerous: those functions might store the pointer or reference in a place where it can later be used without the protection of the mutex. Particularly dangerous in this regard are functions that are supplied at runtime via a function argument or other means, as in the next listing.

Listing 3.2 Accidentally passing out a reference to protected data

```
class some_data
{
    int a;
    std::string b;
public:
    void do_something();
};
class data_wrapper
{
private:
    some_data data;
    std::mutex m;
```

```

public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> l(m);
        func(data);                                #1
    }
};

some_data* unprotected;
void malicious_function(some_data& protected_data)
{
    unprotected=&protected_data;
}
data_wrapper x;
void foo()
{
    x.process_data(malicious_function);          #2
    unprotected->do_something();                 #3
}

```

#1 Pass “protected” data to user-supplied function
#2 Pass in a malicious function
#3 Unprotected access to protected data

In this example, the code in `process_data` looks harmless enough, nicely protected with `std::lock_guard`, but the call to the user-supplied function `func` #1 means that `foo` can pass in `malicious_function` to bypass the protection #2 and then call `do_something()` without the mutex being locked #3.

Fundamentally, the problem with this code is that it hasn’t done what you set out to do: mark all the pieces of code that access the data structure as *mutually exclusive*. In this case, it missed the code in `foo()` that calls `unprotected->do_something()`. Unfortunately, this part of the problem isn’t something the C++ Thread Library can help you with; it’s up to you as programmers to lock the right mutex to protect your data. On the upside, you have a guideline to follow, which will help you in these cases: *Don’t pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions.*

Although this is a common mistake when trying to use mutexes to protect shared data, it’s far from the only potential pitfall. As you’ll see in the next section, it’s still possible to have race conditions, even when data is protected with a mutex.

3.2.3 Spotting race conditions inherent in interfaces

Just because you’re using a mutex or other mechanism to protect shared data, you’re not necessarily protected from race conditions; you still have to ensure that the appropriate data is protected. Consider the doubly linked list example again. In order for a thread to safely delete a node, you need to ensure that you’re preventing concurrent accesses to three nodes: the node being deleted and the nodes on either side. If you protected accesses to the pointers of each node individually, you’d be no better off than with code that used no mutexes, because the race condition could still happen—it’s not the individual nodes that need

protecting for the individual steps but the whole data structure, for the whole delete operation. The easiest solution in this case is to have a single mutex that protects the entire list, as in listing 3.1.

Just because individual operations on the list are safe, you're not out of the woods yet; you can still get race conditions, even with a really simple interface. Consider a stack data structure like the `std::stack` container adapter shown in listing 3.3. Aside from the constructors and `swap()`, there are only five things you can do to a `std::stack`: you can `push()` a new element onto the stack, `pop()` an element off the stack, read the `top()` element, check whether it's `empty()`, and read the number of elements—the `size()` of the stack. If you change `top()` so that it returns a copy rather than a reference (so you're following the guideline from section 3.2.2) and protect the internal data with a mutex, this interface is still inherently subject to race conditions. This problem is not unique to a mutex-based implementation; it's an interface problem, so the race conditions would still occur with a lock-free implementation.

Listing 3.3 The interface to the `std::stack` container adapter

```
template<typename T,typename Container=std::deque<T> >
class stack
{
public:
    explicit stack(const Container&);
    explicit stack(Container&& = Container());
    template <class Alloc> explicit stack(const Alloc&);
    template <class Alloc> stack(const Container&, const Alloc&);
    template <class Alloc> stack(Container&&, const Alloc&);
    template <class Alloc> stack(stack&&, const Alloc&);
    bool empty() const;
    size_t size() const;
    T& top();
    T const& top() const;
    void push(T const&);
    void push(T&&);
    void pop();
    void swap(stack&&);
    template <class... Args> void emplace(Args&&... args); #A
};
```

#A New in C++14

The problem here is that the results of `empty()` and `size()` can't be relied on. Although they might be correct at the time of the call, once they've returned, other threads are free to access the stack and might `push()` new elements onto or `pop()` the existing ones off of the stack before the thread that called `empty()` or `size()` could use that information.

In particular, if the `stack` instance is *not shared*, it's safe to check for `empty()` and then call `top()` to access the top element if the stack is not empty, as follows:

```
stack<int> s;
if(!s.empty())      #1
{
```

```

int const value=s.top();      #2
s.pop();                      #2
do_something(value);
}

```

Not only is it safe in single-threaded code, it's expected: calling `top()` on an empty stack is undefined behavior. With a shared `stack` object, *this call sequence is no longer safe*, because there might be a call to `pop()` from another thread that removes the last element in between the call to `empty()` #1 and the call to `top()`#2. This is therefore a classic race condition, and the use of a mutex internally to protect the stack contents doesn't prevent it; it's a consequence of the interface.

What's the solution? Well, this problem happens as a consequence of the design of the interface, so the solution is to change the interface. However, that still begs the question: what changes need to be made? In the simplest case, you could just declare that `top()` will throw an exception if there aren't any elements in the stack when it's called. Though this directly addresses this issue, it makes for more cumbersome programming, because now you need to be able to catch an exception, even if the call to `empty()` returned `false`. This essentially makes the call to `empty()` an optimization to avoid the overhead of throwing an exception if the stack is already empty (though if the state changes between the call to `empty()` and the call to `top()`, then the exception will still be thrown), rather than a necessary part of the design.

If you look closely at the previous snippet, there's also potential for another race condition but this time between the call to `top()` #2 and the call to `pop()` #3. Consider two threads running the previous snippet of code and both referencing the same `stack` object, `s`. This isn't an unusual situation; when using threads for performance, it's quite common to have several threads running the same code on different data, and a shared `stack` object is ideal for dividing work between them (though more commonly, a queue is used for this purpose — see the examples in chapters 6 and 7). Suppose that initially the stack has two elements, so you don't have to worry about the race between `empty()` and `top()` on either thread, and consider the potential execution patterns.

If the stack is protected by a mutex internally, only one thread can be running a stack member function at any one time, so the calls get nicely interleaved, while the calls to `do_something()` can run concurrently. One possible execution is then as shown in table 3.1.

Table 3.1 A possible ordering of operations on a stack from two threads

Thread A	Thread B
<code>if(!s.empty())</code>	

```

        if(!s.empty())

        int const value=s.top();

        int const value=s.top();

        s.pop();

        do_something(value);           s.pop();

        do_something(value);

```

As you can see, if these are the only threads running, there's nothing in between the two calls to `top()` to modify the stack, so both threads will see the same value. Not only that, but *there are no calls to top() between the calls to pop()*. Consequently, one of the two values on the stack is discarded without ever having been read, whereas the other is processed twice. This is yet another race condition and far more insidious than the undefined behavior of the `empty()/top()` race; there's never anything obviously wrong going on, and the consequences of the bug are likely far removed from the cause, although they obviously depend on exactly what `do_something()` really does.

This calls for a more radical change to the interface, one that combines the calls to `top()` and `pop()` under the protection of the mutex. Tom Cargill² pointed out that a combined call can lead to issues if the copy constructor for the objects on the stack can throw an exception. This problem was dealt with fairly comprehensively from an exception-safety point of view by Herb Sutter,³ but the potential for race conditions brings something new to the mix.

For those of you who aren't aware of the issue, consider a `stack<vector<int>>`. Now, a `vector` is a dynamically sized container, so when you copy a `vector` the library has to allocate some more memory from the heap in order to copy the contents. If the system is heavily loaded, or there are significant resource constraints, this memory allocation can fail, so the copy constructor for `vector` might throw a `std::bad_alloc` exception. This is especially likely if the `vector` contains a lot of elements. If the `pop()` function was defined to return the value popped, as well as remove it from the stack, you have a potential problem: the value being

² Tom Cargill, "Exception Handling: A False Sense of Security," in *C++ Report* 6, no. 9 (November-December 1994). Also available at http://www.informit.com/content/images/020163371x/supplements/Exception_Handling_Article.html.

³ Herb Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* (Addison Wesley Professional, 1999).

popped is returned to the caller only *after* the stack has been modified, but the process of copying the data to return to the caller might throw an exception. If this happens, the data just popped is lost; it has been removed from the stack, but the copy was unsuccessful! The designers of the `std::stack` interface helpfully split the operation in two: get the top element (`top()`) and then remove it from the stack (`pop()`), so that if you can't safely copy the data, it stays on the stack. If the problem was lack of heap memory, maybe the application can free some memory and try again.

Unfortunately, it's precisely this split that you're trying to avoid in eliminating the race condition! Thankfully, there are alternatives, but they aren't without cost.

OPTION 1: PASS IN A REFERENCE

The first option is to pass a reference to a variable in which you wish to receive the popped value as an argument in the call to `pop()`:

```
std::vector<int> result;
some_stack.pop(result);
```

This works well for many cases, but it has the distinct disadvantage that it requires the calling code to construct an instance of the stack's value type prior to the call, in order to pass this in as the target. For some types this is impractical, because constructing an instance is expensive in terms of time or resources. For other types this isn't always possible, because the constructors require parameters that aren't necessarily available at this point in the code. Finally, it requires that the stored type is assignable. This is an important restriction: many user-defined types do not support assignment, though they may support move construction or even copy construction (and thus allow return by value).

OPTION 2: REQUIRE A NO-THROW COPY CONSTRUCTOR OR MOVE CONSTRUCTOR

There's only an exception safety problem with a value-returning `pop()` if the return by value can throw an exception. Many types have copy constructors that don't throw exceptions, and with the new rvalue-reference support in the C++ Standard (see appendix A, section A.1), many more types will have a move constructor that doesn't throw exceptions, even if their copy constructor does. One valid option is to restrict the use of your thread-safe stack to those types that can safely be returned by value without throwing an exception.

Although this is safe, it's not ideal. Even though you can detect at compile time the existence of a copy or move constructor that doesn't throw an exception using the `std::is_nothrow_copy_constructible` and `std::is_nothrow_move_constructible` type traits, it's quite limiting. There are many more user-defined types with copy constructors that can throw and don't have move constructors than there are types with copy and/or move constructors that can't throw (although this might change as people get used to the rvalue-reference support in C++11). It would be unfortunate if such types couldn't be stored in your thread-safe stack.

OPTION 3: RETURN A POINTER TO THE POPPED ITEM

The third option is to return a pointer to the popped item rather than return the item by value. The advantage here is that pointers can be freely copied without throwing an exception, so you've avoided Cargill's exception problem. The disadvantage is that returning a pointer requires a means of managing the memory allocated to the object, and for simple types such as integers, the overhead of such memory management can exceed the cost of just returning the type by value. For any interface that uses this option, `std::shared_ptr` would be a good choice of pointer type; not only does it avoid memory leaks, because the object is destroyed once the last pointer is destroyed, but the library is in full control of the memory allocation scheme and doesn't have to use `new` and `delete`. This can be important for optimization purposes: requiring that each object in the stack be allocated separately with `new` would impose quite an overhead compared to the original non-thread-safe version.

OPTION 4: PROVIDE BOTH OPTION 1 AND EITHER OPTION 2 OR 3

Flexibility should never be ruled out, especially in generic code. If you've chosen option 2 or 3, it's relatively easy to provide option 1 as well, and this provides users of your code the ability to choose whichever option is most appropriate for them for very little additional cost.

EXAMPLE DEFINITION OF A THREAD-SAFE STACK

Listing 3.4 shows the class definition for a stack with no race conditions in the interface and that implements options 1 and 3: there are two overloads of `pop()`, one that takes a reference to a location in which to store the value and one that returns a `std::shared_ptr<T>`. It has a simple interface, with only two functions: `push()` and `pop()`.

Listing 3.4 An outline class definition for a thread-safe stack

```
#include <exception>
#include <memory> #A
struct empty_stack: std::exception
{
    const char* what() const noexcept;
};
template<typename T>
class threadsafe_stack
{
public:
    threadsafe_stack();
    threadsafe_stack(const threadsafe_stack&); #A
    threadsafe_stack& operator=(const threadsafe_stack&) = delete; #1
    void push(T new_value);
    std::shared_ptr<T> pop();
    void pop(T& value);
    bool empty() const;
};

#A For std::shared_ptr<T>
#1 Assignment operator is deleted
```

By paring down the interface you allow for maximum safety; even operations on the whole stack are restricted. The stack itself can't be assigned, because the assignment operator is deleted #1 (see appendix A, section A.2), and there's no `swap()` function. It can, however, be copied, assuming the stack elements can be copied. The `pop()` functions throw an `empty_stack` exception if the stack is empty, so everything still works even if the stack is modified after a call to `empty()`. As mentioned in the description of option 3, the use of `std::shared_ptr` allows the stack to take care of the memory-allocation issues and avoid excessive calls to `new` and `delete` if desired. Your five stack operations have now become three: `push()`, `pop()`, and `empty()`. Even `empty()` is superfluous. This simplification of the interface allows for better control over the data; you can ensure that the mutex is locked for the entirety of an operation. The following listing shows a simple implementation that's a wrapper around `std::stack<>`.

Listing 3.5 A fleshed-out class definition for a thread-safe stack

```
#include <exception>
#include <memory>
#include <mutex>
#include <stack>
struct empty_stack: std::exception
{
    const char* what() const throw();
};
template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;                                #1
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();           #A
        std::shared_ptr<T> const res(std::make_shared<T>(data.top())); #B
        data.pop();
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
```

```

        value=data.top();
        data.pop();
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};

#1 Copy performed in constructor body
#A Check for empty before trying to pop value
#B Allocate return value before modifying stack

```

This stack implementation is actually *copyable*—the copy constructor locks the mutex in the source object and then copies the internal stack. You do the copy in the constructor body #1 rather than the member initializer list in order to ensure that the mutex is held across the copy.

As the discussion of `top()` and `pop()` shows, problematic race conditions in interfaces essentially arise because of locking at too small a granularity; the protection doesn't cover the entirety of the desired operation. Problems with mutexes can also arise from locking at too large a granularity; the extreme situation is a single global mutex that protects all shared data. In a system where there's a significant amount of shared data, this can eliminate any performance benefits of concurrency, because the threads are forced to run one at a time, even when they're accessing different bits of data. The first versions of the Linux kernel that were designed to handle multi-processor systems used a single global kernel lock. Although this worked, it meant that a two-processor system typically had much worse performance than two single-processor systems, and performance on a four-processor system was nowhere near that of four single-processor systems. There was too much contention for the kernel, so the threads running on the additional processors were unable to perform useful work. Later revisions of the Linux kernel have moved to a more fine-grained locking scheme, so the performance of a four-processor system is much nearer the ideal of four times that of a single-processor system, because there's far less contention.

One issue with fine-grained locking schemes is that sometimes you need more than one mutex locked in order to protect all the data in an operation. As described previously, sometimes the right thing to do is increase the granularity of the data covered by the mutexes, so that only one mutex needs to be locked. However, sometimes that's undesirable, such as when the mutexes are protecting separate instances of a class. In this case, locking at the next level up would mean either leaving the locking to the user or having a single mutex that protected all instances of that class, neither of which is particularly desirable.

If you end up having to lock two or more mutexes for a given operation, there's another potential problem lurking in the wings: *deadlock*. This is almost the opposite of a race condition: rather than two threads racing to be first, each one is waiting for the other, so neither makes any progress.

3.2.4 Deadlock: the problem and a solution

Imagine that you have a toy that comes in two parts, and you need both parts to play with it—a toy drum and drumstick, for example. Now imagine that you have two small children, both of whom like playing with it. If one of them gets both the drum and the drumstick, that child can merrily play the drum until tiring of it. If the other child wants to play, they have to wait, however sad that makes them. Now imagine that the drum and the drumstick are buried (separately) in the toy box, and your children both decide to play with them at the same time, so they go rummaging in the toy box. One finds the drum and the other finds the drumstick. Now they're stuck: unless one decides to be nice and let the other play, each will hold onto whatever they have and demand that the other give them the other piece, so neither gets to play.

Now imagine that you have not children arguing over toys but threads arguing over locks on mutexes: each of a pair of threads needs to lock both of a pair of mutexes to perform some operation, and each thread has one mutex and is waiting for the other. Neither thread can proceed, because each is waiting for the other to release its mutex. This scenario is called *deadlock*, and it's the biggest problem with having to lock two or more mutexes in order to perform an operation.

The common advice for avoiding deadlock is to always lock the two mutexes in the same order: if you always lock mutex A before mutex B, then you'll never deadlock. Sometimes this is straightforward, because the mutexes are serving different purposes, but other times it's not so simple, such as when the mutexes are each protecting a separate instance of the same class. Consider, for example, an operation that exchanges data between two instances of the same class; in order to ensure that the data is exchanged correctly, without being affected by concurrent modifications, the mutexes on both instances must be locked. However, if a fixed order is chosen (for example, the mutex for the instance supplied as the first parameter, then the mutex for the instance supplied as the second parameter), this can backfire: all it takes is for two threads to try to exchange data between the same two instances with the parameters swapped, and you have deadlock!

Thankfully, the C++ Standard Library has a cure for this in the form of `std::lock`—a function that can lock two or more mutexes at once without risk of deadlock. The example in the next listing shows how to use this for a simple swap operation.

Listing 3.6 Using `std::lock()` and `std::lock_guard` in a swap operation

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);
class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd){}
    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
```

```

        return;
    std::lock(lhs.m, rhs.m);           #1
    std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);   #2
    std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);   #3
    swap(lhs.some_detail, rhs.some_detail);
}
};

```

First, the arguments are checked to ensure they are different instances, because attempting to acquire a lock on a `std::mutex` when you already hold it is undefined behavior. (A mutex that does permit multiple locks by the same thread is provided in the form of `std::recursive_mutex`. See section 3.3.3 for details.) Then, the call to `std::lock()` #1 locks the two mutexes, and two `std::lock_guard` instances are constructed #2, #3, one for each mutex. The `std::adopt_lock` parameter is supplied in addition to the mutex to indicate to the `std::lock_guard` objects that the mutexes are already locked, and they should just adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor.

This ensures that the mutexes are correctly unlocked on function exit in the general case where the protected operation might throw an exception; it also allows for a simple return. Also, it's worth noting that locking either `lhs.m` or `rhs.m` inside the call to `std::lock` can throw an exception; in this case, the exception is propagated out of `std::lock`. If `std::lock` has successfully acquired a lock on one mutex and an exception is thrown when it tries to acquire a lock on the other mutex, this first lock is released automatically: `std::lock` provides all-or-nothing semantics with regard to locking the supplied mutexes.

C++17 provides additional support for this scenario, in the form of the a new RAII template, `std::scoped_lock<>`. This is exactly equivalent to `std::lock_guard<>`, except that it is a *variadic template*, accepting a *list* of mutex types as template parameters, and a *list* of mutexes as constructor arguments. The mutexes supplied to the constructor are locked using the same algorithm as `std::lock`, so that when the constructor completes they are all locked, and they are then all unlocked in the destructor. The `swap()` operation from listing 3.6 can thus be rewritten as follows:

```

void swap(X& lhs, X& rhs)
{
    if(&lhs==&rhs)
        return;
    std::scoped_lock guard(lhs.m, rhs.m); #1
    swap(lhs.some_detail, rhs.some_detail);
}

```

This example uses another feature added with C++17: automatic deduction of class template parameters. If you have a C++17 compiler (which is likely if you're using `std::scoped_lock`, since that is a C++17 library facility), the C++17 implicit class template parameter deduction mechanism will choose the correct mutex types from the types of the

objects passed to the constructor at object (#1). This line is thus equivalent to the fully specified version:

```
std::scoped_lock<std::mutex, std::mutex> guard(lhs.m, rhs.m);
```

The existence of `std::scoped_lock` means that most of the cases where you would have used `std::lock` prior to C++17 can now be written using `std::scoped_lock`, with less potential for mistakes, which can only be a good thing!

Although `std::lock` (and `std::scoped_lock<>`) can help you avoid deadlock in those cases where you need to acquire two or more locks together, it doesn't help if they're acquired separately. In that case you have to rely on your discipline as developers to ensure you don't get deadlock. This isn't easy: deadlocks are one of the nastiest problems to encounter in multithreaded code and are often unpredictable, with everything working fine the majority of the time. There are, however, some relatively simple rules that can help you to write deadlock-free code.

3.2.5 Further guidelines for avoiding deadlock

Deadlock doesn't just occur with locks, although that's the most frequent cause; you can create deadlock with two threads and no locks just by having each thread call `join()` on the `std::thread` object for the other. In this case, neither thread can make progress because it's waiting for the other to finish, just like the children fighting over their toys. This simple cycle can occur anywhere that a thread can wait for another thread to perform some action if the other thread can simultaneously be waiting for the first thread, and it isn't limited to two threads: a cycle of three or more threads will still cause deadlock. The guidelines for avoiding deadlock all boil down to one idea: don't wait for another thread if there's a chance it's waiting for you. The individual guidelines provide ways of identifying and eliminating the possibility that the other thread is waiting for you.

AVOID NESTED LOCKS

The first idea is the simplest: don't acquire a lock if you already hold one. If you stick to this guideline, it's impossible to get a deadlock from the lock usage alone because each thread only ever holds a single lock. You could still get deadlock from other things (like the threads waiting for each other), but mutex locks are probably the most common cause of deadlock. If you need to acquire multiple locks, do it as a single action with `std::lock` in order to acquire them without deadlock.

AVOID CALLING USER-SUPPLIED CODE WHILE HOLDING A LOCK

This is a simple follow-on from the previous guideline. Because the code is user supplied, you have no idea what it could do; it could do anything, including acquiring a lock. If you call user-supplied code while holding a lock, and that code acquires a lock, you've violated the guideline on avoiding nested locks and could get deadlock. Sometimes this is unavoidable; if

you're writing generic code such as the stack in section 3.2.3, every operation on the parameter type or types is user-supplied code. In this case, you need a new guideline.

ACQUIRE LOCKS IN A FIXED ORDER

If you absolutely must acquire two or more locks, and you can't acquire them as a single operation with `std::lock`, the next-best thing is to acquire them in the same order in every thread. I touched on this in section 3.2.4 as one way of avoiding deadlock when acquiring two mutexes: the key is to define the order in a way that's consistent between threads. In some cases, this is relatively easy. For example, look at the stack from section 3.2.3—the mutex is internal to each stack instance, but the operations on the data items stored in a stack require calling user-supplied code. You can, however, add the constraint that none of the operations on the data items stored in the stack should perform any operation on the stack itself. This puts the burden on the user of the stack, but it's rather uncommon for the data stored in a container to access that container, and it's quite apparent when this is happening, so it's not a particularly difficult burden to carry.

In other cases, this isn't so straightforward, as you discovered with the swap operation in section 3.2.4. At least in that case you could lock the mutexes simultaneously, but that's not always possible. If you look back at the linked list example from section 3.1, you'll see that one possibility for protecting the list is to have a mutex per node. Then, in order to access the list, threads must acquire a lock on every node they're interested in. For a thread to delete an item, it must then acquire the lock on three nodes: the node being deleted and the nodes on either side, because they're all being modified in some way. Likewise, to traverse the list a thread must keep hold of the lock on the current node while it acquires the lock on the next one in the sequence, in order to ensure that the next pointer isn't modified in the meantime. Once the lock on the next node has been acquired, the lock on the first can be released because it's no longer necessary.

This hand-over-hand locking style allows multiple threads to access the list, provided each is accessing a different node. However, in order to prevent deadlock, the nodes must always be locked in the same order: if two threads tried to traverse the list in opposite orders using hand-over-hand locking, they could deadlock with each other in the middle of the list. If nodes A and B are adjacent in the list, the thread going one way will try to hold the lock on node A and try to acquire the lock on node B. A thread going the other way would be holding the lock on node B and trying to acquire the lock on node A—a classic scenario for deadlock, as shown in figure 3.2.

Thread 1	Thread 2
Lock master entry mutex	
Read head node pointer	

Lock head node mutex

Unlock master entry mutex

Lock master entry mutex

Read head→next pointer

Read tail node pointer

Unlock head node mutex

Lock tail node mutex

Lock next node mutex

Read tail→prev pointer

Read next→next pointer

Unlock tail node mutex

...

...

Lock node A mutex

Lock node C mutex

Read A→next pointer (which is B)

Read C→next pointer (which is B)

Lock node B mutex

Block trying to lock node B mutex

Unlock node C mutex

Read B→prev pointer (which is A)

Block trying to lock node A mutex

Deadlock!

Figure 3.2: Deadlock with threads traversing a list in opposite orders

Likewise, when deleting node B that lies between nodes A and C, if that thread acquires the lock on B before the locks on A and C, it has the potential to deadlock with a thread traversing the list. Such a thread would try to lock either A or C first (depending on the direction of traversal) but would then find that it couldn't obtain a lock on B because the thread doing the deleting was holding the lock on B and trying to acquire the locks on A and C.

One way to prevent deadlock here is to define an order of traversal, so a thread must always lock A before B and B before C. This would eliminate the possibility of deadlock at the expense of disallowing reverse traversal. Similar conventions can often be established for other data structures.

USE A LOCK HIERARCHY

Although this is really a particular case of defining lock ordering, a lock hierarchy can provide a means of checking that the convention is adhered to at runtime. The idea is that you divide your application into layers and identify all the mutexes that may be locked in any given layer. When code tries to lock a mutex, it isn't permitted to lock that mutex if it already holds a lock from a lower layer. You can check this at runtime by assigning layer numbers to each mutex and keeping a record of which mutexes are locked by each thread. This is a common pattern, but the C++ Standard Library does not provide direct support for it, so we will need to write a custom `hierarchical_mutex` type, the code for which is shown in listing 3.8.

The following listing shows an example of two threads using a hierarchical mutex.

Listing 3.7 Using a lock hierarchy to prevent deadlock

```
hierarchical_mutex high_level_mutex(10000);      #1
hierarchical_mutex low_level_mutex(5000);           #2
hierarchical_mutex other_mutex(6000);               #3
int do_low_level_stuff();
int low_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(low_level_mutex);  #4
    return do_low_level_stuff();
}
void high_level_stuff(int some_param);
void high_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(high_level_mutex);   #6
    high_level_stuff(low_level_func());                         #5
}
void thread_a()          #7
{
    high_level_func();
}

void do_other_stuff();
void other_stuff()
{
    high_level_func();        #10
    do_other_stuff();
}
void thread_b()          #8
{
    std::lock_guard<hierarchical_mutex> lk(other_mutex);       #9
    other_stuff();
}
```

This code has 3 instances of `hierarchical_mutex`, (#1, #2 and #3), which are constructed with progressively lower hierarchy numbers. Since the mechanism is defined so

that if you hold a lock on a `hierarchical_mutex` then you can only acquire a lock on another `hierarchical_mutex` with a lower hierarchy number, this then imposes restrictions on what the code can do.

Assuming `do_low_level_stuff` doesn't lock any mutexes, `low_level_func` is the bottom of our hierarchy, and locks the `low_level_mutex` (#4). `high_level_func` calls `low_level_func` (#5) while holding a lock on `high_level_mutex` (#6), but that's OK, because the hierarchy level of `high_level_mutex` (#1: 10000) is higher than that of `low_level_mutex` (#2: 5000).

`thread_a()` #7 thus abides by the rules, so it runs fine.

On the other hand, `thread_b()` #8 disregards the rules and therefore will fail at runtime.

First off, it locks the `other_mutex` #9, which has a hierarchy value of only 6000 #3. This means it should be somewhere mid-way in the hierarchy. When `other_stuff()` calls `high_level_func()` #10, it's thus violating the hierarchy: `high_level_func()` tries to acquire the `high_level_mutex`, which has a value of 10000, considerably more than the current hierarchy value of 6000. The `hierarchical_mutex` will therefore report an error, possibly by throwing an exception or aborting the program. Deadlocks between hierarchical mutexes are thus impossible, because the mutexes themselves enforce the lock ordering. This does mean that you can't hold two locks at the same time if they're the same level in the hierarchy, so hand-over-hand locking schemes require that each mutex in the chain have a lower hierarchy value than the prior one, which may be impractical in some cases.

This example also demonstrates another point, the use of the `std::lock_guard<>` template with a user-defined mutex type. `hierarchical_mutex` is not part of the standard but is easy to write; a simple implementation is shown in listing 3.8. Even though it's a user-defined type, it can be used with `std::lock_guard<>` because it implements the three member functions required to satisfy the mutex concept: `lock()`, `unlock()`, and `try_lock()`. You haven't yet seen `try_lock()` used directly, but it's fairly simple: if the lock on the mutex is held by another thread, it returns `false` rather than waiting until the calling thread can acquire the lock on the mutex. It may also be used by `std::lock()` internally, as part of the deadlock-avoidance algorithm.

The implementation of `hierarchical_mutex` uses a thread-local variable to store the current hierarchy value. This value is thus accessible to all mutex instances, but has a different value on each thread. This allows the code to check the behavior of each thread separately, and the code for each mutex can check whether or not the current thread is allowed to lock that mutex.

Listing 3.8 A simple hierarchical mutex

```
class hierarchical_mutex
{
    std::mutex internal_mutex;
    unsigned long const hierarchy_value;
    unsigned long previous_hierarchy_value;
    static thread_local unsigned long this_thread_hierarchy_value;      #1
    void check_for_hierarchyViolation()
    {
        if (this_thread_hierarchy_value < previous_hierarchy_value)
            throw std::runtime_error("Hierarchy violation detected!");
        previous_hierarchy_value = this_thread_hierarchy_value;
    }
}
```

```

{
    if(this_thread_hierarchy_value <= hierarchy_value)      #2
    {
        throw std::logic_error("mutex hierarchy violated");
    }
}
void update_hierarchy_value()
{
    previous_hierarchy_value=this_thread_hierarchy_value;    #3
    this_thread_hierarchy_value=hierarchy_value;
}
public:
    explicit hierarchical_mutex(unsigned long value):
        hierarchy_value(value),
        previous_hierarchy_value(0)
    {}
    void lock()
    {
        check_for_hierarchyViolation();
        internal_mutex.lock();                      #4
        update_hierarchy_value();                  #5
    }
    void unlock()
    {
        if(this_thread_hierarchy_value!=hierarchy_value)
            throw std::logic_error("mutex hierarchy violated"); #9
        this_thread_hierarchy_value=previous_hierarchy_value;   #6
        internal_mutex.unlock();
    }
    bool try_lock()
    {
        check_for_hierarchyViolation();
        if(!internal_mutex.try_lock())           #7
            return false;
        update_hierarchy_value();
        return true;
    }
};
thread_local unsigned long
hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);    #8

```

The key here is the use of the `thread_local` value representing the hierarchy value for the current thread: `this_thread_hierarchy_value` #1. It's initialized to the maximum value #8, so initially any mutex can be locked. Because it's declared `thread_local`, every thread has its own copy, so the state of the variable in one thread is entirely independent of the state of the variable when read from another thread. See appendix A, section A.8, for more information about `thread_local`.

So, the first time a thread locks an instance of `hierarchical_mutex` the value of `this_thread_hierarchy_value` is `ULONG_MAX`. By its very nature, this is greater than any other value, so the check in `check_for_hierarchyViolation()` #2 passes. With that check out of the way, `lock()` delegates to the internal mutex for the actual locking #4. Once this lock has succeeded, you can update the hierarchy value #5.

If you now lock *another* hierarchical_mutex while holding the lock on this first one, the value of `this_thread_hierarchy_value` reflects the hierarchy value of the first mutex. The hierarchy value of this second mutex must now be less than that of the mutex already held in order for the check #2 to pass.

Now, it's important to save the previous value of the hierarchy value for the current thread so you can restore it in `unlock()` #6; otherwise you'd never be able to lock a mutex with a higher hierarchy value again, even if the thread didn't hold any locks. Because you store this previous hierarchy value only when you hold the `internal_mutex` #3, and you restore it before you unlock the internal mutex #6, you can safely store it in the `hierarchical_mutex` itself, because it's safely protected by the lock on the internal mutex. In order to avoid the hierarchy getting confused due to out-of-order unlocking, we throw at #9 if the mutex being unlocked is not the most recently locked one. Other mechanisms are possible, but this is the simplest.

`try_lock()` works the same as `lock()` except that if the call to `try_lock()` on the `internal_mutex` fails #7, then you don't own the lock, so you don't update the hierarchy value and return `false` rather than `true`.

Although detection is a runtime check, it's at least not timing dependent—you don't have to wait around for the rare conditions that cause deadlock to show up. Also, the design process required to divide the application and mutexes in this way can help eliminate many possible causes of deadlock before they even get written. It might be worth performing the design exercise even if you then don't go as far as actually writing the runtime checks.

EXTENDING THESE GUIDELINES BEYOND LOCKS

As I mentioned back at the beginning of this section, deadlock doesn't just occur with locks; it can occur with any synchronization construct that can lead to a wait cycle. It's therefore worth extending these guidelines to cover those cases too. For example, just as you should avoid acquiring nested locks if possible, it's a bad idea to wait for a thread while holding a lock, because that thread might need to acquire the lock in order to proceed. Similarly, if you're going to wait for a thread to finish, it might be worth identifying a thread hierarchy, such that a thread waits only for threads lower down the hierarchy. One simple way to do this is to ensure that your threads are joined in the same function that started them, as described in sections 3.1.2 and 3.3.

Once you've designed your code to avoid deadlock, `std::lock()` and `std::lock_guard` cover most of the cases of simple locking, but sometimes more flexibility is required. For those cases, the Standard Library provides the `std::unique_lock` template. Like `std::lock_guard`, this is a class template parameterized on the mutex type, and it also provides the same RAII-style lock management as `std::lock_guard` but with a bit more flexibility.

3.2.6 Flexible locking with `std::unique_lock`

`std::unique_lock` provides a bit more flexibility than `std::lock_guard` by relaxing the invariants; a `std::unique_lock` instance doesn't always own the mutex that it's associated

with. First off, just as you can pass `std::adopt_lock` as a second argument to the constructor to have the lock object manage the lock on a mutex, you can also pass `std::defer_lock` as the second argument to indicate that the mutex should remain unlocked on construction. The lock can then be acquired later by calling `lock()` on the `std::unique_lock` object (*not* the mutex) or by passing the `std::unique_lock` object itself to `std::lock()`. Listing 3.6 could just as easily have been written as shown in listing 3.9, using `std::unique_lock` and `std::defer_lock` #1 rather than `std::lock_guard` and `std::adopt_lock`. The code has the same line count and is essentially equivalent, apart from one small thing: `std::unique_lock` takes more space and is a fraction slower to use than `std::lock_guard`. The flexibility of allowing a `std::unique_lock` instance *not* to own the mutex comes at a price: this information has to be stored, and it has to be updated.

Listing 3.9 Using `std::lock()` and `std::unique_lock` in a swap operation

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd){}
    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock); #1
        std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock); #1
        std::lock(lock_a,lock_b);                                #2
        swap(lhs.some_detail,rhs.some_detail);
    }
};

#1 std::defer_lock leaves mutexes unlocked
#2 Mutexes are locked here
```

In listing 3.9, the `std::unique_lock` objects could be passed to `std::lock()` #2 because `std::unique_lock` provides `lock()`, `try_lock()`, and `unlock()` member functions. These forward to the member functions of the same name on the underlying mutex to do the actual work and just update a flag inside the `std::unique_lock` instance to indicate whether the mutex is currently owned by that instance. This flag is necessary in order to ensure that `unlock()` is called correctly in the destructor. If the instance *does* own the mutex, the destructor *must* call `unlock()`, and if the instance *does not* own the mutex, it *must not* call `unlock()`. This flag can be queried by calling the `owns_lock()` member function. Unless you're going to be transferring lock ownership around or doing something else that requires `std::unique_lock`, you're still better off using the C++17 variadic `std::scoped_lock` if it's available to you (see section 3.2.4).

As you might expect, this flag has to be stored somewhere. Therefore, the size of a `std::unique_lock` object is typically larger than that of a `std::lock_guard` object, and there's also a slight performance penalty when using `std::unique_lock` over `std::lock_guard` because the flag has to be updated or checked, as appropriate. If `std::lock_guard` is sufficient for your needs, I'd therefore recommend using it in preference. That said, there are cases where `std::unique_lock` is a better fit for the task at hand, because you need to make use of the additional flexibility. One example is deferred locking, as you've already seen; another case is where the ownership of the lock needs to be transferred from one scope to another.

3.2.7 Transferring mutex ownership between scopes

Because `std::unique_lock` instances don't have to own their associated mutexes, the ownership of a mutex can be transferred between instances by *moving* the instances around. In some cases such transfer is automatic, such as when returning an instance from a function, and in other cases you have to do it explicitly by calling `std::move()`. Fundamentally this depends on whether the source is an *lvalue*—a real variable or reference to one—or an *rvalue*—a temporary of some kind. Ownership transfer is automatic if the source is an rvalue and must be done explicitly for an lvalue in order to avoid accidentally transferring ownership away from a variable. `std::unique_lock` is an example of a type that's *movable* but not *copyable*. See appendix A, section A.1.1, for more about move semantics.

One possible use is to allow a function to lock a mutex and transfer ownership of that lock to the caller, so the caller can then perform additional actions under the protection of the same lock. The following code snippet shows an example of this: the function `get_lock()` locks the mutex and then prepares the data before returning the lock to the caller:

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;           #1
}
void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock());      #2
    do_something();
}
```

Because `lk` is an automatic variable declared within the function, it can be returned directly #1 without a call to `std::move()`; the compiler takes care of calling the move constructor. The `process_data()` function can then transfer ownership directly into its own `std::unique_lock` instance #2, and the call to `do_something()` can rely on the data being correctly prepared without another thread altering the data in the meantime.

Typically this sort of pattern would be used where the mutex to be locked is dependent on the current state of the program or on an argument passed in to the function that returns the `std::unique_lock` object. One such usage is where the lock isn't returned directly but is a

data member of a gateway class used to ensure correctly locked access to some protected data. In this case, all access to the data is through this gateway class: when you wish to access the data, you obtain an instance of the gateway class (by calling a function such as `get_lock()` in the preceding example), which acquires the lock. You can then access the data through member functions of the gateway object. When you're finished, you destroy the gateway object, which releases the lock and allows other threads to access the protected data. Such a gateway object may well be movable (so that it can be returned from a function), in which case the lock object data member also needs to be movable.

The flexibility of `std::unique_lock` also allows instances to relinquish their locks before they're destroyed. You can do this with the `unlock()` member function, just like for a mutex: `std::unique_lock` supports the same basic set of member functions for locking and unlocking as a mutex does, in order that it can be used with generic functions such as `std::lock`. The ability to release a lock before the `std::unique_lock` instance is destroyed means that you can optionally release it in a specific code branch if it's apparent that the lock is no longer required. This can be important for the performance of the application; holding a lock for longer than required can cause a drop in performance, because other threads waiting for the lock are prevented from proceeding for longer than necessary.

3.2.8 Locking at an appropriate granularity

The granularity of a lock is something I touched on earlier, in section 3.2.3: the lock granularity is a hand-waving term to describe the amount of data protected by a single lock. A fine-grained lock protects a small amount of data, and a coarse-grained lock protects a large amount of data. Not only is it important to choose a sufficiently coarse lock granularity to ensure the required data is protected, but it's also important to ensure that a lock is held only for the operations that actually require it. We all know the frustration of waiting in the checkout line in a supermarket with a cart full of groceries only for the person currently being served to suddenly realize that they forgot some cranberry sauce and then leave everybody waiting while they go and find some, or for the cashier to be ready for payment and the customer to only then start rummaging in their purse for their wallet. Everything proceeds much more easily if everybody gets to the checkout with everything they want and with an appropriate means of payment ready.

The same applies to threads: if multiple threads are waiting for the same resource (the cashier at the checkout), then if any thread holds the lock for longer than necessary, it will increase the total time spent waiting (don't wait until you've reached the checkout to start looking for the cranberry sauce). Where possible, lock a mutex only while actually accessing the shared data; try to do any processing of the data outside the lock. In particular, don't do any really time-consuming activities like file I/O while holding a lock. File I/O is typically hundreds (if not thousands) of times slower than reading or writing the same volume of data from memory. So unless the lock is really intended to protect access to the file, performing I/O while holding the lock will delay *other* threads unnecessarily (because they'll block while

waiting to acquire the lock), potentially eliminating any performance gain from the use of multiple threads.

`std::unique_lock` works well in this situation, because you can call `unlock()` when the code no longer needs access to the shared data and then call `lock()` again if access is required later in the code:

```
void get_and_process_data()
{
    std::unique_lock<std::mutex> my_lock(the_mutex);
    some_class data_to_process=get_next_data_chunk();
    my_lock.unlock();                                #1
    result_type result=process(data_to_process);
    my_lock.lock();                                 #2
    write_result(data_to_process,result);
}
```

#1 Don't need mutex locked across call to process()

#2 Relock mutex to write result

You don't need the mutex locked across the call to `process()`, so you manually unlock it before the call #1 and then lock it again afterward #2.

Hopefully it's obvious that if you have one mutex protecting an entire data structure, not only is there likely to be more contention for the lock, but also the potential for reducing the time that the lock is held is less. More of the operation steps will require a lock on the same mutex, so the lock must be held longer. This double whammy of a cost is thus also a double incentive to move toward finer-grained locking wherever possible.

As this example shows, locking at an appropriate granularity isn't only about the amount of data locked; it's also about how long the lock is held and what operations are performed while the lock is held. *In general, a lock should be held for only the minimum possible time needed to perform the required operations.* This also means that time-consuming operations such as acquiring another lock (even if you know it won't deadlock) or waiting for I/O to complete shouldn't be done while holding a lock unless absolutely necessary.

In listings 3.6 and 3.9, the operation that required locking the two mutexes was a swap operation, which obviously requires concurrent access to both objects. Suppose instead you were trying to compare a simple data member that was just a plain `int`. Would this make a difference? `ints` are cheap to copy, so you could easily copy the data for each object being compared while only holding the lock for that object and then compare the copied values. This would mean that you were holding the lock on each mutex for the minimum amount of time and also that you weren't holding one lock while locking another. The following listing shows a class `Y` for which this is the case and a sample implementation of the equality comparison operator.

Listing 3.10 Locking one mutex at a time in a comparison operator

```
class Y
{
private:
```

```

int some_detail;
mutable std::mutex m;
int get_detail() const
{
    std::lock_guard<std::mutex> lock_a(m);      #1
    return some_detail;
}
public:
Y(int sd):some_detail(sd){}
friend bool operator==(Y const& lhs, Y const& rhs)
{
    if(&lhs==&rhs)
        return true;
    int const lhs_value=lhs.get_detail();          #2
    int const rhs_value=rhs.get_detail();          #3
    return lhs_value==rhs_value;                  #4
}
};

```

In this case, the comparison operator first retrieves the values to be compared by calling the `get_detail()` member function #2, #3. This function retrieves the value while protecting it with a lock #1. The comparison operator then compares the retrieved values #4. Note, however, that as well as reducing the locking periods so that only one lock is held at a time (and thus eliminating the possibility of deadlock), *this has subtly changed the semantics of the operation* compared to holding both locks together. In listing 3.10, if the operator returns `true`, it means that the value of `lhs.some_detail` at one point in time is equal to the value of `rhs.some_detail` at another point in time. The two values could have been changed in any way in between the two reads; the values could have been swapped in between #2 and #3, for example, thus rendering the comparison meaningless. The equality comparison might thus return `true` to indicate that the values were equal, even though there was never an instant in time when the values were actually equal. It's therefore important to be careful when making such changes that the semantics of the operation are not changed in a problematic fashion: *If you don't hold the required locks for the entire duration of an operation, you're exposing yourself to race conditions.*

Sometimes, there just isn't an appropriate level of granularity because not all accesses to the data structure require the same level of protection. In this case, it might be appropriate to use an alternative mechanism, instead of a plain `std::mutex`.

3.3 Alternative facilities for protecting shared data

Although they're the most general mechanism, mutexes aren't the only game in town when it comes to protecting shared data; there are alternatives that provide more appropriate protection in specific scenarios.

One particularly extreme (but remarkably common) case is where the shared data needs protection only from concurrent access while it's being initialized, but after that no explicit synchronization is required. This might be because the data is read-only once created, and so there are no possible synchronization issues, or it might be because the necessary protection is performed implicitly as part of the operations on the data. In either case, locking a mutex

after the data has been initialized, purely in order to protect the initialization, is unnecessary and a needless hit to performance. It's for this reason that the C++ Standard provides a mechanism purely for protecting shared data during initialization.

3.3.1 Protecting shared data during initialization

Suppose you have a shared resource that's so expensive to construct that you want to do so only if it's actually required; maybe it opens a database connection or allocates a lot of memory. *Lazy initialization* such as this is common in single-threaded code—each operation that requires the resource first checks to see if it has been initialized and then initializes it before use if not:

```
std::shared_ptr<some_resource> resource_ptr;
void foo()
{
    if(!resource_ptr)
    {
        resource_ptr.reset(new some_resource);      #1
    }
    resource_ptr->do_something();
}
```

If the shared resource itself is safe for concurrent access, the only part that needs protecting when converting this to multithreaded code is the initialization #1, but a naive translation such as that in the following listing can cause unnecessary serialization of threads using the resource. This is because each thread must wait on the mutex in order to check whether the resource has already been initialized.

Listing 3.11 Thread-safe lazy initialization using a mutex

```
std::shared_ptr<some_resource> resource_ptr;
std::mutex resource_mutex;
void foo()
{
    std::unique_lock<std::mutex> lk(resource_mutex);   #A
    if(!resource_ptr)
    {
        resource_ptr.reset(new some_resource);      #B
    }
    lk.unlock();
    resource_ptr->do_something();
}
```

#A All threads are serialized here

#B Only the initialization needs protection

This code is common enough, and the unnecessary serialization problematic enough, that many people have tried to come up with a better way of doing this, including the infamous *Double-Checked Locking* pattern: the pointer is first read without acquiring the lock #1 (in the code below), and the lock is acquired only if the pointer is `NULL`. The pointer is then checked

again once the lock has been acquired #2 (hence the *double-checked* part) in case another thread has done the initialization between the first check and this thread acquiring the lock:

```
void undefined_behaviour_with_double_checked_locking()
{
    if(!resource_ptr)                      #1
    {
        std::lock_guard<std::mutex> lk(resource_mutex);
        if(!resource_ptr)                  #2
        {
            resource_ptr.reset(new some_resource);   #3
        }
    }
    resource_ptr->do_something();      #4
}
```

Unfortunately, this pattern is infamous for a reason: it has the potential for nasty race conditions, because the read outside the lock #1 isn't synchronized with the write done by another thread inside the lock #3. This therefore creates a race condition that covers not just the pointer itself but also the object pointed to; even if a thread sees the pointer written by another thread, it might not see the newly created instance of `some_resource`, resulting in the call to `do_something()` #4 operating on incorrect values. This is an example of the type of race condition defined as a *data race* by the C++ Standard and thus specified as *undefined behavior*. It's is therefore quite definitely something to avoid. See chapter 5 for a detailed discussion of the memory model, including what constitutes a *data race*.

The C++ Standards Committee also saw that this was an important scenario, and so the C++ Standard Library provides `std::once_flag` and `std::call_once` to handle this situation. Rather than locking a mutex and explicitly checking the pointer, every thread can just use `std::call_once`, safe in the knowledge that the pointer will have been initialized by some thread (in a properly synchronized fashion) by the time `std::call_once` returns. The necessary synchronization data is stored in the `std::once_flag` instance; each instance of `std::once_flag` corresponds to a different initialization. Use of `std::call_once` will typically have a lower overhead than using a mutex explicitly, especially when the initialization has already been done, so should be used in preference where it matches the required functionality. The following example shows the same operation as listing 3.11, rewritten to use `std::call_once`. In this case, the initialization is done by calling a function, but it could just as easily have been done with an instance of a class with a function call operator. Like most of the functions in the standard library that take functions or predicates as aruments, `std::call_once` works with any function or callable object.

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;          #1
void init_resource()
{
    resource_ptr.reset(new some_resource);
}
void foo()
{
    std::call_once(resource_flag,init_resource);  #A
```

```
    resource_ptr->do_something();  
}
```

#A Initialization is called exactly once

In this example, both the `std::once_flag` #1 and data being initialized are namespace-scope objects, but `std::call_once()` can just as easily be used for lazy initialization of class members, as in the following listing.

Listing 3.12 Thread-safe lazy initialization of a class member using `std::call_once`

```
class X  
{  
private:  
    connection_info connection_details;  
    connection_handle connection;  
    std::once_flag connection_init_flag;  
    void open_connection()  
    {  
        connection=connection_manager.open(connection_details);  
    }  
public:  
    X(connection_info const& connection_details_):  
        connection_details(connection_details_)  
    {}  
    void send_data(data_packet const& data)      #1  
    {  
        std::call_once(connection_init_flag,&X::open_connection,this);  #2  
        connection.send_data(data);  
    }  
    data_packet receive_data()          #3  
    {  
        std::call_once(connection_init_flag,&X::open_connection,this);#2  
        return connection.receive_data();  
    }  
};
```

In that example, the initialization is done either by the first call to `send_data()` #1 or by the first call to `receive_data()` #3. The use of the member function `open_connection()` to initialize the data also requires that the `this` pointer be passed in. Just as for other functions in the Standard Library that accept callable objects, such as the constructor for `std::thread` and `std::bind()`, this is done by passing an additional argument to `std::call_once()` #2.

It's worth noting that, like `std::mutex`, `std::once_flag` instances can't be copied or moved, so if you use them as a class member like this, you'll have to explicitly define these special member functions should you require them.

One scenario where there's a potential race condition over initialization is that of a local variable declared with `static`. The initialization of such a variable is defined to occur the first time control passes through its declaration; for multiple threads calling the function, this means there's the potential for a race condition to define first. On many pre-C++11 compilers this race condition is problematic in practice, because multiple threads may believe they're first and try to initialize the variable, or threads may try to use it after initialization has started

on another thread but before it's finished. In C++11 this problem is solved: the initialization is defined to happen on exactly one thread, and no other threads will proceed until that initialization is complete, so the race condition is just over which thread gets to do the initialization rather than anything more problematic. This can be used as an alternative to `std::call_` once for those cases where a single global instance is required:

```
class my_class;
my_class& get_my_class_instance()
{
    static my_class instance;      #1
    return instance;
}
```

#1 Initialization guaranteed to be thread-safe

Multiple threads can then call `get_my_class_instance()` safely #1, without having to worry about race conditions on the initialization.

Protecting data only for initialization is a special case of a more general scenario: that of a rarely updated data structure. For most of the time, such a data structure is read-only and can therefore be merrily read by multiple threads concurrently, but on occasion the data structure may need updating. What's needed here is a protection mechanism that acknowledges this fact.

3.3.2 Protecting rarely updated data structures

Consider a table used to store a cache of DNS entries for resolving domain names to their corresponding IP addresses. Typically, a given DNS entry will remain unchanged for a long period of time—in many cases DNS entries remain unchanged for years. Although new entries may be added to the table from time to time as users access different websites, this data will therefore remain largely unchanged throughout its life. It's important that the validity of the cached entries be checked periodically, but this still requires an update only if the details have actually changed.

Although updates are rare, they can still happen, and if this cache is to be accessed from multiple threads, it will need to be appropriately protected during updates to ensure that none of the threads reading the cache see a broken data structure.

In the absence of a special-purpose data structure that exactly fits the desired usage and that's specially designed for concurrent updates and reads (such as those in chapters 6 and 7), such an update requires that the thread doing the update have exclusive access to the data structure until it has completed the operation. Once the change is complete, the data structure is again safe for multiple threads to access concurrently. Using a `std::mutex` to protect the data structure is therefore overly pessimistic, because it will eliminate the possible concurrency in reading the data structure when it isn't undergoing modification; what's needed is a different kind of mutex. This new kind of mutex is typically called a *reader-writer* mutex, because it allows for two different kinds of usage: exclusive access by a single "writer" thread or shared, concurrent access by multiple "reader" threads.

The C++17 Standard Library provides two such mutexes out of the box, `std::shared_mutex` and `std::shared_timed_mutex`. C++14 only features `std::shared_timed_mutex`, and C++11 didn't provide either. If you're struck with a pre-C++14 compiler, then you could use the implementation provided by the Boost library, which is based on the original proposal. The difference between the `std::shared_mutex` and `std::shared_timed_mutex` is that `std::shared_timed_mutex` supports additional operations (as described in section 4.3), so `std::shared_mutex` might offer a performance benefit on some platforms, if you don't need the additional operations.

As you'll see in chapter 8, the use of such a mutex isn't a panacea, and the performance is dependent on the number of processors involved and the relative workloads of the reader and update threads. It's therefore important to profile the performance of the code on the target system to ensure that there's actually a benefit to the additional complexity.

Rather than using an instance of `std::mutex` for the synchronization, you use an instance of `std::shared_mutex`. For the update operations, `std::lock_guard<std::shared_mutex>` and `std::unique_lock<std::shared_mutex>` can be used for the locking, in place of the corresponding `std::mutex` specializations. These ensure exclusive access, just as with `std::mutex`. Those threads that don't need to update the data structure can instead use `std::shared_lock<std::shared_mutex>` to obtain *shared* access. This RAI class template was added in C++14, and is used just the same as `std::unique_lock`, except that multiple threads may have a shared lock on the same `std::shared_mutex` at the same time. The only constraint is that if any thread has a shared lock, a thread that tries to acquire an exclusive lock will block until all other threads have relinquished their locks, and likewise if any thread has an exclusive lock, no other thread may acquire a shared or exclusive lock until the first thread has relinquished its lock.

The following listing shows a simple DNS cache like the one just described, using a `std::map` to hold the cached data, protected using a `std::shared_mutex`.

Listing 3.13 Protecting a data structure with a `std::shared_mutex`

```
#include <map>
#include <string>
#include <mutex>
#include <shared_mutex>
class dns_entry;
class dns_cache
{
    std::map<std::string,dns_entry> entries;
    mutable std::shared_mutex entry_mutex;
public:
    dns_entry find_entry(std::string const& domain) const
    {
        std::shared_lock<std::shared_mutex> lk(entry_mutex);    #1
        std::map<std::string,dns_entry>::const_iterator const it=
            entries.find(domain);
        return (it==entries.end())?dns_entry():it->second;
    }
    void update_or_add_entry(std::string const& domain,
                           dns_entry const& dns_details)
```

```

    {
        std::lock_guard<std::shared_mutex> lk(entry_mutex);      #2
        entries[domain]=dns_details;
    }
};

```

In listing 3.13, `find_entry()` uses an instance of `std::shared_lock<>` to protect it for shared, read-only access #1; multiple threads can therefore call `find_entry()` simultaneously without problems. On the other hand, `update_or_add_entry()` uses an instance of `std::lock_guard<>` to provide exclusive access while the table is updated #2; not only are other threads prevented from doing updates in a call `update_ or_add_entry()`, but threads that call `find_entry()` are blocked too.

3.3.3 Recursive locking

With `std::mutex`, it's an error for a thread to try to lock a mutex it already owns, and attempting to do so will result in *undefined behavior*. However, in some circumstances it would be desirable for a thread to reacquire the same mutex several times without having first released it. For this purpose, the C++ Standard Library provides `std::recursive_mutex`. It works just like `std::mutex`, except that you can acquire multiple locks on a single instance from the same thread. You must release all your locks before the mutex can be locked by another thread, so if you call `lock()` three times, you must also call `unlock()` three times. Correct use of `std::lock_guard <std::recursive_mutex>` and `std::unique_lock<std::recursive_mutex>` will handle this for you.

Most of the time, if you think you want a recursive mutex, you probably need to change your design instead. A common use of recursive mutexes is where a class is designed to be accessible from multiple threads concurrently, so it has a mutex protecting the member data. Each public member function locks the mutex, does the work, and then unlocks the mutex. However, sometimes it's desirable for one public member function to call another as part of its operation. In this case, the second member function will also try to lock the mutex, thus leading to undefined behavior. The quick-and-dirty solution is to change the mutex to a recursive mutex. This will allow the mutex lock in the second member function to succeed and the function to proceed.

However, such usage is *not recommended*, because it can lead to sloppy thinking and bad design. In particular, the class invariants are typically broken while the lock is held, which means that the second member function needs to work even when called with the invariants broken. It's usually better to extract a new private member function that's called from both member functions, which does not lock the mutex (it expects it to already be locked). You can then think carefully about the circumstances under which that new function can be called and the state of the data under those circumstances.

3.4 Summary

In this chapter I discussed how problematic race conditions can be disastrous when sharing data between threads and how to use `std::mutex` and careful interface design to avoid them.

You saw that mutexes aren't a panacea and do have their own problems in the form of deadlock, though the C++ Standard Library provides a tool to help avoid that in the form of `std::lock()`. You then looked at some further techniques for avoiding deadlock, followed by a brief look at transferring lock ownership and issues surrounding choosing the appropriate granularity for locking. Finally, I covered the alternative data-protection facilities provided for specific scenarios, such as `std::call_once()`, and `std::shared_mutex`.

One thing that I haven't covered yet, however, is waiting for input from other threads. Our thread-safe stack just throws an exception if the stack is empty, so if one thread wanted to wait for another thread to push a value on the stack (which is, after all, one of the primary uses for a thread-safe stack), it would have to repeatedly try to pop a value, retrying if an exception gets thrown. This consumes valuable processing time in performing the check, without actually making any progress; indeed, the constant checking might *hamper* progress by preventing the other threads in the system from running. What's needed is some way for a thread to wait for another thread to complete a task without consuming CPU time in the process. Chapter 4 builds on the facilities I've discussed for protecting shared data and introduces the various mechanisms for synchronizing operations between threads in C++; chapter 6 shows how these can be used to build larger reusable data structures.

4

Synchronizing concurrent operations

This chapter covers

- Waiting for an event
- Waiting for one-off events with futures
- Waiting with a time limit
- Using synchronization of operations to
- simplify code

In the last chapter, we looked at various ways of protecting data that's shared between threads. But sometimes you don't just need to protect the data but also to synchronize actions on separate threads. One thread might need to wait for another thread to complete a task before the first thread can complete its own, for example. In general, it's common to want a thread to wait for a specific event to happen or a condition to be `true`. Although it would be possible to do this by periodically checking a "task complete" flag or something similar stored in shared data, this is far from ideal. The need to synchronize operations between threads like this is such a common scenario that the C++ Standard Library provides facilities to handle it, in the form of *condition variables* and *futures*. These facilities are extended in the Concurrency TS, which provides additional operations for *futures*, alongside new synchronization facilities in the form of *latches* and *barriers*.

In this chapter I'll discuss how to wait for events with condition variables, futures, latches and barriers, and how to use them to simplify the synchronization of operations.

4.1 Waiting for an event or other condition

Suppose you're traveling on an overnight train. One way to ensure you get off at the right station would be to stay awake all night and pay attention to where the train stops. You wouldn't miss your station, but you'd be tired when you got there. Alternatively, you could look at the timetable to see when the train is supposed to arrive, set your alarm a bit before, and go to sleep. That would be OK; you wouldn't miss your stop, but if the train got delayed, you'd wake up too early. There's also the possibility that your alarm clock's batteries would die, and you'd sleep too long and miss your station. What would be ideal is if you could just go to sleep and have somebody or something wake you up when the train gets to your station, whenever that is.

How does that relate to threads? Well, if one thread is waiting for a second thread to complete a task, it has several options. First, it could just keep checking a flag in shared data (protected by a mutex) and have the second thread set the flag when it completes the task. This is wasteful on two counts: the thread consumes valuable processing time repeatedly checking the flag, and when the mutex is locked by the waiting thread, it can't be locked by any other thread. Both of these work against the thread doing the waiting: if the waiting thread is running, this limits the execution resources available to run the thread being waited for, and while the waiting thread has locked the mutex protecting the flag in order to check it, the thread being waited for is unable to lock the mutex to set the flag when it's done. This is akin to staying awake all night talking to the train driver: he has to drive the train more slowly because you keep distracting him, so it takes longer to get there. Similarly, the waiting thread is consuming resources that could be used by other threads in the system and may end up waiting longer than necessary.

A second option is to have the waiting thread sleep for small periods between the checks using the `std::this_thread::sleep_for()` function (see section 4.3):

```
bool flag;
std::mutex m;
void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();                      #1
        std::this_thread::sleep_for(std::chrono::milliseconds(100)); #2
        lk.lock();           #3
    }
}

#1 Unlock the mutex
#2 Sleep for 100 ms
#3 Relock the mutex
```

In the loop, the function unlocks the mutex #1 before the sleep #2 and locks it again afterward #3, so another thread gets a chance to acquire it and set the flag.

This is an improvement, because the thread doesn't waste processing time while it's sleeping, but it's hard to get the sleep period right. Too short a sleep in between checks and the thread still wastes processing time checking; too long a sleep and the thread will keep on sleeping even when the task it's waiting for is complete, introducing a delay. It's rare that this oversleeping will have a direct impact on the operation of the program, but it could mean dropped frames in a fast-paced game or overrunning a time slice in a real-time application.

The third, and preferred, option is to use the facilities from the C++ Standard Library to wait for the event itself. The most basic mechanism for waiting for an event to be triggered by another thread (such as the presence of additional work in the pipeline mentioned previously) is the *condition variable*. Conceptually, a condition variable is associated with some event or other *condition*, and one or more threads can *wait* for that condition to be satisfied. When some thread has determined that the condition is satisfied, it can then *notify* one or more of the threads waiting on the condition variable, in order to wake them up and allow them to continue processing.

4.1.1 Waiting for a condition with condition variables

The Standard C++ Library provides not one but *two* implementations of a condition variable: `std::condition_variable` and `std::condition_variable_any`. Both of these are declared in the `<condition_variable>` library header. In both cases, they need to work with a mutex in order to provide appropriate synchronization; the former is limited to working with `std::mutex`, whereas the latter can work with anything that meets some minimal criteria for being mutex-like, hence the `_any` suffix. Because `std::condition_variable_any` is more general, there's the potential for additional costs in terms of size, performance, or operating system resources, so `std::condition_variable` should be preferred unless the additional flexibility is required.

So, how do you use a `std::condition_variable` to handle the example in the introduction—how do you let the thread that's waiting for work sleep until there's data to process? The following listing shows one way you could do this with a condition variable.

Listing 4.1 Waiting for data to process with a std::condition_variable

```
std::mutex mut;
std::queue<data_chunk> data_queue;    #1
std::condition_variable data_cond;
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        {
            std::lock_guard<std::mutex> lk(mut);
            data_queue.push(data);          #2
        }
        data_cond.notify_one();    #3
    }
}
void data_processing_thread()
```

```

{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);      #4
        data_cond.wait(
            lk,[<]{}{return !data_queue.empty();});   #5
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();          #6
        process(data);
        if(is_last_chunk(data))
            break;
    }
}

```

First off, you have a queue #1 that's used to pass the data between the two threads. When the data is ready, the thread preparing the data locks the mutex protecting the queue using a `std::lock_guard` and pushes the data onto the queue #2. It then calls the `notify_one()` member function on the `std::condition_variable` instance to notify the waiting thread (if there is one) #3. Note that we put the code to push the data onto the queue in a smaller scope, so we notify the condition variable **after** unlocking the mutex — this is so that if the waiting thread wakes immediately, it doesn't then have to block again, waiting for us to unlock the mutex.

On the other side of the fence, you have the processing thread. This thread first locks the mutex, but this time with a `std::unique_lock` rather than a `std::lock_guard` #4—you'll see why in a minute. The thread then calls `wait()` on the `std::condition_variable`, passing in the lock object and a lambda function that expresses the condition being waited for #5. Lambda functions are a new feature in C++11 that allows you to write an anonymous function as part of another expression, and they're ideally suited for specifying predicates for standard library functions such as `wait()`. In this case, the simple lambda function `[<]{}{return !data_queue.empty();}` checks to see if the `data_queue` is not `empty()`—that is, there's some data in the queue ready for processing. Lambda functions are described in more detail in appendix A, section A.5.

The implementation of `wait()` then checks the condition (by calling the supplied lambda function) and returns if it's satisfied (the lambda function returned `true`). If the condition isn't satisfied (the lambda function returned `false`), `wait()` unlocks the mutex and puts the thread in a blocked or waiting state. When the condition variable is notified by a call to `notify_one()` from the data-preparation thread, the thread wakes from its slumber (unblocks it), reacquires the lock on the mutex, and checks the condition again, returning from `wait()` with the mutex still locked if the condition has been satisfied. If the condition hasn't been satisfied, the thread unlocks the mutex and resumes waiting. This is why you need the `std::unique_lock` rather than the `std::lock_guard`—the waiting thread must unlock the mutex while it's waiting and lock it again afterward, and `std::lock_guard` doesn't provide that flexibility. If the mutex remained locked while the thread was sleeping, the data-preparation thread wouldn't be able to lock the mutex to add an item to the queue, and the waiting thread would never be able to see its condition satisfied.

Listing 4.1 uses a simple lambda function for the wait #5, which checks to see if the queue is not empty, but any function or callable object could be passed. If you already have a function to check the condition (perhaps because it's more complicated than a simple test like this), then this function can be passed in directly; there's no need to wrap it in a lambda. During a call to `wait()`, a condition variable may check the supplied condition any number of times; however, it always does so with the mutex locked and will return immediately if (and only if) the function provided to test the condition returns `true`. When the waiting thread reacquires the mutex and checks the condition, if it isn't in direct response to a notification from another thread, it's called a *spurious wake*. Because the number and frequency of any such spurious wakes are by definition indeterminate, it isn't advisable to use a function with side effects for the condition check. If you do so, you must be prepared for the side effects to occur multiple times.

Fundamentally, `std::condition_variable::wait` is just **an optimization over a busy-wait**. Indeed, a conforming (though less than ideal) implementation technique is just a simple loop:

```
template<typename Predicate>
void minimal_wait(std::unique_lock<std::mutex>& lk, Predicate pred){
    while(!pred()){
        lk.unlock();
        lk.lock();
    }
}
```

Your code must be prepared to work with such a minimal implementation of `wait()`, as well as an implementation that **only** wakes up if `notify_one()` or `notify_all()` is called.

The flexibility to unlock a `std::unique_lock` isn't just used for the call to `wait()`; it's also used once you have the data to process but before processing it #6. Processing data can potentially be a time-consuming operation, and as you saw in chapter 3, it's a bad idea to hold a lock on a mutex for longer than necessary.

Using a queue to transfer data between threads as in listing 4.1 is a common scenario. Done well, the synchronization can be limited to the queue itself, which greatly reduces the possible number of synchronization problems and race conditions. In view of this, let's now work on extracting a generic thread-safe queue from listing 4.1.

4.1.2 Building a thread-safe queue with condition variables

If you're going to be designing a generic queue, it's worth spending a few minutes thinking about the operations that are likely to be required, as you did with the thread-safe stack back in section 3.2.3. Let's look at the C++ Standard Library for inspiration, in the form of the `std::queue<T>` container adaptor shown in the following listing.

Listing 4.2 `std::queue` interface

```
template <class T, class Container = std::deque<T> >
class queue {
public:
```

```

explicit queue(const Container&);
explicit queue(Container&& = Container());
template <class Alloc> explicit queue(const Alloc&);
template <class Alloc> queue(const Container&, const Alloc&);
template <class Alloc> queue(Container&&, const Alloc&);
template <class Alloc> queue(queue&&, const Alloc&);
void swap(queue& q);
bool empty() const;
size_type size() const;
T& front();
const T& front() const;
T& back();
const T& back() const;
void push(const T& x);
void push(T&& x);
void pop();
template <class... Args> void emplace(Args&&... args);
};

```

If you ignore the construction, assignment and swap operations, you're left with three groups of operations: those that query the state of the whole queue (`empty()` and `size()`), those that query the elements of the queue (`front()` and `back()`), and those that modify the queue (`push()`, `pop()` and `emplace()`). This is the same as you had back in section 3.2.3 for the stack, and therefore you have the same issues regarding race conditions inherent in the interface. Consequently, you need to combine `front()` and `pop()` into a single function call, much as you combined `top()` and `pop()` for the stack. The code from listing 4.1 adds a new nuance, though: when using a queue to pass data between threads, the receiving thread often needs to wait for the data. Let's provide two variants on `pop():try_pop()`, which tries to pop the value from the queue but always returns immediately (with an indication of failure) even if there wasn't a value to retrieve, and `wait_and_pop()`, which will wait until there's a value to retrieve. If you take your lead for the signatures from the stack example, your interface looks like the following.

Listing 4.3 The interface of your `threadsafe_queue`

```

#include <memory>          #A
template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);

    threadsafe_queue& operator=(const threadsafe_queue&) = delete;      #B
    void push(T new_value);
    bool try_pop(T& value);    #1
    std::shared_ptr<T> try_pop();           #2
    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();
    bool empty() const;
};

```

#A For `std::shared_ptr`

#B Disallow assignment for simplicity

As you did for the stack, you've cut down on the constructors and eliminated assignment in order to simplify the code. You've also provided two versions of both `try_pop()` and `wait_for_pop()`, as before. The first overload of `try_pop()` #1 stores the retrieved value in the referenced variable, so it can use the return value for status; it returns `true` if it retrieved a value and `false` otherwise (see section A.2). The second overload #2 can't do this, because it returns the retrieved value directly. But the returned pointer can be set to `NULL` if there's no value to retrieve.

So, how does all this relate to listing 4.1? Well, you can extract the code for `push()` and `wait_and_pop()` from there, as shown in the next listing.

Listing 4.4 Extracting `push()` and `wait_and_pop()` from listing 4.1

```
#include <queue>
#include <mutex>
#include <condition_variable>
template<typename T>
class threadsafe_queue
{
private:
    std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
};
threadsafe_queue<data_chunk> data_queue;    #1
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data);          #2
    }
}
void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data);  #3
        process(data);
    }
}
```

```

        if(is_last_chunk(data))
            break;
    }
}

```

The mutex and condition variable are now contained within the `threadsafe_queue` instance, so separate variables are no longer required #1, and no external synchronization is required for the call to `push()` #2. Also, `wait_and_pop()` takes care of the condition variable `wait` #3.

The other overload of `wait_and_pop()` is now trivial to write, and the remaining functions can be copied almost verbatim from the stack example in listing 3.5. The final queue implementation is shown here.

Listing 4.5 Full class definition for a thread-safe queue using condition variables

```

#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;      #1
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }
    bool try_pop(T& value)

```

```

    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=data_queue.front();
        data_queue.pop();
        return true;
    }
    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};

#1 The mutex must be mutable

```

Even though `empty()` is a `const` member function, and the other parameter to the copy constructor is a `const` reference, other threads may have non-`const` references to the object, and be calling mutating member functions, so we still need to lock the mutex. Since locking a mutex is a mutating operation, the mutex object must be marked `mutable` #1 so it can be locked in `empty()` and in the copy constructor.

Condition variables are also useful where there's more than one thread waiting for the same event. If the threads are being used to divide the workload, and thus only one thread should respond to a notification, exactly the same structure as shown in listing 4.1 can be used: just run multiple instances of the data-processing thread. When new data is ready, the call to `notify_one()` will trigger one of the threads currently executing `wait()` to check its condition and thus return from `wait()` (because you've just added an item to the `data_queue`). There's no guarantee which thread will be notified or even if there's a thread waiting to be notified; all the processing threads might be still processing data.

Another possibility is that several threads are waiting for the same event, and all of them need to respond. This can happen where shared data is being initialized, and the processing threads can all use the same data but need to wait for it to be initialized (although there are potentially better mechanisms for this, such as `std::call_once`; see section 3.3.1 in chapter 3 for a discussion of the options), or where the threads need to wait for an update to shared data, such as a periodic reinitialization. In these cases, the thread preparing the data can call the `notify_all()` member function on the condition variable rather than `notify_one()`. As the name suggests, this causes *all* the threads currently executing `wait()` to check the condition they're waiting for.

If the waiting thread is going to wait only once, so when the condition is `true` it will never wait on this condition variable again, a condition variable might not be the best choice of synchronization mechanisms. This is especially true if the condition being waited for is the availability of a particular piece of data. In this scenario, a *future* might be more appropriate.

4.2 Waiting for one-off events with futures

Suppose you're going on vacation abroad by plane. Once you get to the airport and clear the various check-in procedures, you still have to wait for notification that your flight is ready for boarding, possibly for several hours. Yes, you might be able to find some means of passing the time, such as reading a book, surfing the internet, or eating in an overpriced airport café, but fundamentally you're just waiting for one thing: the signal that it's time to get on the plane. Not only that, but a given flight goes only once; the next time you're going on vacation, you'll be waiting for a different flight.

The C++ Standard Library models this sort of one-off event with something called a *future*. If a thread needs to wait for a specific one-off event, it somehow obtains a future representing this event. The thread can then periodically wait on the future for short periods of time to see if the event has occurred (check the departures board) while performing some other task (eating in the overpriced café) in between checks. Alternatively, it can do another task until it needs the event to have happened before it can proceed and then just wait for the future to become *ready*. A future may have data associated with it (such as which gate your flight is boarding at), or it may not. Once an event has happened (and the future has become *ready*), the future can't be reset.

There are two sorts of futures in the C++ Standard Library, implemented as two class templates declared in the `<future>` library header: *unique futures* (`std::future<>`) and *shared futures* (`std::shared_future<>`). These are modeled after `std::unique_ptr` and `std::shared_ptr`. An instance of `std::future` is the one and only instance that refers to its associated event, whereas multiple instances of `std::shared_future` may refer to the same event. In the latter case, all the instances will become *ready* at the same time, and they may all access any data associated with the event. This associated data is the reason these are templates; just like `std::unique_ptr` and `std::shared_ptr`, the template parameter is the type of the associated data. The `std::future<void>`, `std::shared_future<void>` template specializations should be used where there's no associated data. Although futures are used to communicate between threads, the future objects themselves don't provide synchronized accesses. If multiple threads need to access a single future object, they must protect access via a mutex or other synchronization mechanism, as described in chapter 3. However, as you'll see in section 4.2.5, multiple threads may each access their own copy of a `std::shared_future<>` without further synchronization, even if they all refer to the same asynchronous result.

The Concurrency TS provides extended versions of these class templates in the `std::experimental` namespace: `std::experimental::future<>` and `std::experimental::shared_future<>`. These behave identically to their counterparts in

namespace `std`, but they have additional member functions to provide additional facilities. It is important to note that the name `std::experimental` does not imply anything about the quality of the code (I would hope that the implementation will be the same quality as everything else shipped from your library vendor), but highlights that these are non-standard classes and functions, and thus may not have exactly the same syntax and semantics if and when they are finally adopted into a future C++ Standard. To use these facilities you must include the `<experimental/future>` header.

The most basic of one-off events is the result of a calculation that has been run in the background. Back in chapter 2 you saw that `std::thread` doesn't provide an easy means of returning a value from such a task, and I promised that this would be addressed in chapter 4 with futures—now it's time to see how.

4.2.1 Returning values from background tasks

Suppose you have a long-running calculation that you expect will eventually yield a useful result but for which you don't currently need the value. Maybe you've found a way to determine the answer to Life, the Universe, and Everything, to pinch an example from Douglas Adams.¹ You could start a new thread to perform the calculation, but that means you have to take care of transferring the result back, because `std::thread` doesn't provide a direct mechanism for doing so. This is where the `std::async` function template (also declared in the `<future>` header) comes in.

You use `std::async` to start an *asynchronous task* for which you don't need the result right away. Rather than giving you back a `std::thread` object to wait on, `std::async` returns a `std::future` object, which will eventually hold the return value of the function. When you need the value, you just call `get()` on the future, and the thread blocks until the future is *ready* and then returns the value. The following listing shows a simple example.

Listing 4.6 Using `std::future` to get the return value of an asynchronous task

```
#include <future>
#include <iostream>
int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
    std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
    do_other_stuff();
    std::cout<<"The answer is "<<the_answer.get()<<std::endl;
}
```

`std::async` allows you to pass additional arguments to the function by adding extra arguments to the call, in the same way that `std::thread` does. If the first argument is a pointer to a member function, the second argument provides the object on which to apply the

¹ In *The Hitchhiker's Guide to the Galaxy*, the computer Deep Thought is built to determine “the answer to Life, the Universe and Everything.” The answer is 42.

member function (either directly, or via a pointer, or wrapped in `std::ref`), and the remaining arguments are passed as arguments to the member function. Otherwise, the second and subsequent arguments are passed as arguments to the function or callable object specified as the first argument. Just as with `std::thread`, if the arguments are rvalues, the copies are created by *moving* the originals. This allows the use of move-only types as both the function object and the arguments. See the following listing.

Listing 4.7 Passing arguments to a function with `std::async`

```
#include <string>
#include <future>
struct X
{
    void foo(int,std::string const&);  

    std::string bar(std::string const&);  

};  

X x;  

auto f1=std::async(&X::foo,&x,42,"hello");      #A  

auto f2=std::async(&X::bar,x,"goodbye");        #B  

struct Y
{
    double operator()(double);  

};  

Y y;  

auto f3=std::async(Y(),3.141);                  #C  

auto f4=std::async(std::ref(y),2.718);          #D  

X baz(x);  

std::async(baz,std::ref(x));                    #E  

class move_only
{
public:
    move_only();
    move_only(move_only&&)
    move_only(move_only const&) = delete;
    move_only& operator=(move_only&&);
    move_only& operator=(move_only const&) = delete;
    void operator()();  

};  

auto f5=std::async(move_only());                 #F  
  

#A Calls p->foo(42,"hello") where p is &x  

#B Calls tmpx.bar("goodbye") where tmpx is a copy of x  

#C Calls tmpy(3.141) where tmpy is move-constructed from Y()  

#D Calls y(2.718)  

#E Calls baz(x)  

#F Calls tmp() where tmp is constructed from std::move(move_only())
```

By default, it's up to the implementation whether `std::async` starts a new thread, or whether the task runs synchronously when the future is waited for. In most cases this is what you want, but you can specify which to use with an additional parameter to `std::async` before the function to call. This parameter is of the type `std::launch`, and can either be `std::launch::deferred` to indicate that the function call is to be deferred until either `wait()` or `get()` is called on the future, `std::launch::async` to indicate that the function must be

run on its own thread, or `std::launch::deferred` | `std::launch::async` to indicate that the implementation may choose. This last option is the default. If the function call is deferred, it may never actually run. For example:

```
auto f6=std::async(std::launch::async,Y(),1.2);          #A
auto f7=std::async(std::launch::deferred,baz,std::ref(x));    #B
auto f8=std::async(                                     #C
    std::launch::deferred | std::launch::async,
    baz,std::ref(x));
auto f9=std::async(baz,std::ref(x));
f7.wait();                                         #D

#A Run in new thread
#B Run in wait() or get()
#C Implementation chooses
#D Invoke deferred function
```

As you'll see later in this chapter and again in chapter 8, the use of `std::async` makes it easy to divide algorithms into tasks that can be run concurrently. However, it's not the only way to associate a `std::future` with a task; you can also do it by wrapping the task in an instance of the `std::packaged_task<>` class template or by writing code to explicitly set the values using the `std::promise<>` class template. `std::packaged_task` is a higher-level abstraction than `std::promise`, so I'll start with that.

4.2.2 Associating a task with a future

`std::packaged_task<>` ties a future to a function or callable object. When the `std::packaged_task<>` object is invoked, it calls the associated function or callable object and makes the future *ready*, with the return value stored as the associated data. This can be used as a building block for thread pools (see chapter 9) or other task management schemes, such as running each task on its own thread, or running them all sequentially on a particular background thread. If a large operation can be divided into self-contained sub-tasks, each of these can be wrapped in a `std::packaged_task<>` instance, and then that instance passed to the task scheduler or thread pool. This abstracts out the details of the tasks; the scheduler just deals with `std::packaged_task<>` instances rather than individual functions.

The template parameter for the `std::packaged_task<>` class template is a function signature, like `void()` for a function taking no parameters with no return value, or `int(std::string&,double*)` for a function that takes a non-const reference to a `std::string` and a pointer to a `double` and returns an `int`. When you construct an instance of `std::packaged_task`, you must pass in a function or callable object that can accept the specified parameters and that returns a type convertible to the specified return type. The types don't have to match exactly; you can construct a `std::packaged_task<double(double)>` from a function that takes an `int` and returns a `float` because the types are implicitly convertible.

The return type of the specified function signature identifies the type of the `std::future<>` returned from the `get_future()` member function, whereas the argument list of the function

signature is used to specify the signature of the packaged task's function call operator. For example, a partial class definition for `std::packaged_task<std::string(std::vector<char>*,int)>` would be as shown in the following listing.

Listing 4.8 Partial class definition for a specialization of `std::packaged_task<>`

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};
```

The `std::packaged_task` object is thus a callable object, and it can be wrapped in a `std::function` object, passed to a `std::thread` as the thread function, passed to another function that requires a callable object, or even invoked directly. When the `std::packaged_task` is invoked as a function object, the arguments supplied to the function call operator are passed on to the contained function, and the return value is stored as the asynchronous result in the `std::future` obtained from `get_future()`. You can thus wrap a task in a `std::packaged_task` and retrieve the future before passing the `std::packaged_task` object elsewhere to be invoked in due course. When you need the result, you can wait for the future to become ready. The following example shows this in action.

PASSING TASKS BETWEEN THREADS

Many GUI frameworks require that updates to the GUI be done from specific threads, so if another thread needs to update the GUI, it must send a message to the right thread in order to do so. `std::packaged_task` provides one way of doing this without requiring a custom message for each and every GUI-related activity, as shown here.

Listing 4.9 Running code on a GUI thread using `std::packaged_task`

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>
std::mutex m;
std::deque<std::packaged_task<void()>> tasks;
bool gui_shutdown_message_received();
void get_and_process_gui_message();
void gui_thread()      #1
{
    while(!gui_shutdown_message_received())    #2
    {
        get_and_process_gui_message();          #3
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
```

```

        if(tasks.empty())          #4
            continue;
        task=std::move(tasks.front());    #5
        tasks.pop_front();
    }
    task();    #6
}
std::thread gui_bg_thread(gui_thread);
template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f);      #7
    std::future<void> res=task.get_future();      #8
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));      #9
    return res;                          #10
}

```

This code is very simple: the GUI thread #1 loops until a message has been received telling the GUI to shut down #2, repeatedly polling for GUI messages to handle #3, such as user clicks, and for tasks on the task queue. If there are no tasks on the queue #4, it loops again; otherwise, it extracts the task from the queue #5, releases the lock on the queue, and then runs the task #6. The future associated with the task will then be made ready when the task completes.

Posting a task on the queue is equally simple: a new packaged task is created from the supplied function #7, the future is obtained from that task #8 by calling the `get_future()` member function, and the task is put on the list #9 before the future is returned to the caller #10. The code that posted the message to the GUI thread can then wait for the future if it needs to know that the task has been completed, or it can discard the future if it doesn't need to know.

This example uses `std::packaged_task<void()>` for the tasks, which wraps a function or other callable object that takes no parameters and returns `void` (if it returns anything else, the return value is discarded). This is the simplest possible task, but as you saw earlier, `std::packaged_task` can also be used in more complex situations—by specifying a different function signature as the template parameter, you can change the return type (and thus the type of data stored in the future's associated state) and also the argument types of the function call operator. This example could easily be extended to allow for tasks that are to be run on the GUI thread to accept arguments and return a value in the `std::future` rather than just a completion indicator.

What about those tasks that can't be expressed as a simple function call or those tasks where the result may come from more than one place? These cases are dealt with by the third way of creating a future: using a `std::promise` to set the value explicitly.

4.2.3 Making (`std::promise`) promises

When you have an application that needs to handle a lot of network connections, it's often tempting to handle each connection on a separate thread, because this can make the network

communication easier to think about and easier to program. This works well for low numbers of connections (and thus low numbers of threads). Unfortunately, as the number of connections rises, this becomes less suitable; the large numbers of threads consequently consume large numbers of operating system resources and potentially cause a lot of context switching (when the number of threads exceeds the available hardware concurrency), impacting performance. In the extreme case, the operating system may run out of resources for running new threads before its capacity for network connections is exhausted. In applications with very large numbers of network connections, it's therefore common to have a small number of threads (possibly only one) handling the connections, each thread dealing with multiple connections at once.

Consider one of these threads handling the connections. Data packets will come in from the various connections being handled in essentially random order, and likewise data packets will be queued to be sent in random order. In many cases, other parts of the application will be waiting either for data to be successfully sent or for a new batch of data to be successfully received via a specific network connection.

`std::promise<T>` provides a means of setting a value (of type `T`), which can later be read through an associated `std::future<T>` object. A `std::promise/std::future` pair would provide one possible mechanism for this facility; the waiting thread could block on the future, while the thread providing the data could use the promise half of the pairing to set the associated value and make the future *ready*.

You can obtain the `std::future` object associated with a given `std::promise` by calling the `get_future()` member function, just like with `std::packaged_task`. When the value of the promise is set (using the `set_value()` member function), the future becomes *ready* and can be used to retrieve the stored value. If you destroy the `std::promise` without setting a value, an exception is stored instead. Section 4.2.4 describes how exceptions are transferred across threads.

Listing 4.10 shows some example code for a thread processing connections as just described. In this example, you use a `std::promise<bool>/std::future<bool>` pair to identify the successful transmission of a block of outgoing data; the value associated with the future is a simple success/failure flag. For incoming packets, the data associated with the future is the payload of the data packet.

Listing 4.10 Handling multiple connections from a single thread using promises

```
#include <future>
void process_connections(connection_set& connections)
{
    while(!done(connections))    #1
    {
        for(connection_iterator      #2
            connection=connections.begin(),end=connections.end();
            connection!=end;
            ++connection)
        {
            if(connection->has_incoming_data())    #3
            {
```

```

        data_packet data=connection->incoming();
        std::promise<payload_type>& p=
            connection->get_promise(data.id);    #4
        p.set_value(data.payload);
    }
    if(connection->has_outgoing_data())    #5
    {
        outgoing_packet data=
            connection->top_of_outgoing_queue();
        connection->send(data.payload);
        data.promise.set_value(true);      #6
    }
}
}
}

```

The function `process_connections()` loops until `done()` returns `true` #1. Every time through the loop, it checks each connection in turn #2, retrieving incoming data if there is any #3 or sending any queued outgoing data #5. This assumes that an incoming packet has some ID and a payload with the actual data in it. The ID is mapped to a `std::promise` (perhaps by a lookup in an associative container) #4, and the value is set to the packet's payload. For outgoing packets, the packet is retrieved from the outgoing queue and actually sent through the connection. Once the send has completed, the promise associated with the outgoing data is set to `true` to indicate successful transmission #6. Whether this maps nicely to the actual network protocol depends on the protocol; this promise/future style structure may not work for a particular scenario, although it does have a similar structure to the asynchronous I/O support of some operating systems.

All the code up to now has completely disregarded exceptions. Although it might be nice to imagine a world in which everything worked all the time, this isn't actually the case. Sometimes disks fill up, sometimes what you're looking for just isn't there, sometimes the network fails, and sometimes the database goes down. If you were performing the operation in the thread that needed the result, the code could just report an error with an exception, so it would be unnecessarily restrictive to require that everything go well just because you wanted to use a `std::packaged_task` or a `std::promise`. The C++ Standard Library therefore provides a clean way to deal with exceptions in such a scenario and allows them to be saved as part of the associated result.

4.2.4 Saving an exception for the future

Consider the following short snippet of code. If you pass in `-1` to the `square_root()` function, it throws an exception, and this gets seen by the caller:

```

double square_root(double x)
{
    if(x<0)
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}

```

Now suppose that instead of just invoking `square_root()` from the current thread,

```
double y=square_root(-1);
```

you run the call as an asynchronous call:

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

It would be ideal if the behavior was exactly the same; just as `y` gets the result of the function call in either case, it would be great if the thread that called `f.get()` could see the exception too, just as it would in the single-threaded case.

Well, that's exactly what happens: if the function call invoked as part of `std::async` throws an exception, that exception is stored in the future in place of a stored value, the future becomes *ready*, and a call to `get()` rethrows that stored exception. (Note: the standard leaves it unspecified whether it is the original exception object that's rethrown or a copy; different compilers and libraries make different choices on this matter.) The same happens if you wrap the function in a `std::packaged_task`—when the task is invoked, if the wrapped function throws an exception, that exception is stored in the future in place of the result, ready to be thrown on a call to `get()`.

Naturally, `std::promise` provides the same facility, with an explicit function call. If you wish to store an exception rather than a value, you call the `set_exception()` member function rather than `set_value()`. This would typically be used in a catch block for an exception thrown as part of the algorithm, to populate the promise with that exception:

```
extern std::promise<double> some_promise;
try
{
    some_promise.set_value(calculate_value());
}
catch(...)
{
    some_promise.set_exception(std::current_exception());
}
```

This uses `std::current_exception()` to retrieve the thrown exception; the alternative here would be to use `std::make_exception_ptr()` to store a new exception directly without throwing:

```
some_promise.set_exception(std::make_exception_ptr(std::logic_error("foo ")));
```

This is much cleaner than using a try/catch block if the type of the exception is known, and it should be used in preference; not only does it simplify the code, but it also provides the compiler with greater opportunity to optimize the code.

Another way to store an exception in a future is to destroy the `std::promise` or `std::packaged_task` associated with the future without calling either of the set functions on the promise or invoking the packaged task. In either case, the destructor of the `std::promise` or `std::packaged_task` will store a `std::future_error` exception with an error code of

`std::future_errc::broken_promise` in the associated state if the future isn't already *ready*; by creating a future you make a promise to provide a value or exception, and by destroying the source of that value or exception without providing one, you break that promise. If the compiler didn't store anything in the future in this case, waiting threads could potentially wait forever.

Up until now all the examples have used `std::future`. However, `std::future` has its limitations, not the least of which being that only one thread can wait for the result. If you need to wait for the same event from more than one thread, you need to use `std::shared_future` instead.

4.2.5 Waiting from multiple threads

Although `std::future` handles all the synchronization necessary to transfer data from one thread to another, calls to the member functions of a particular `std::future` instance are not synchronized with each other. If you access a single `std::future` object from multiple threads without additional synchronization, you have a *data race* and undefined behavior. This is by design: `std::future` models unique ownership of the asynchronous result, and the one-shot nature of `get()` makes such concurrent access pointless anyway—only one thread can retrieve the value, because after the first call to `get()` there's no value left to retrieve.

If your fabulous design for your concurrent code requires that multiple threads can wait for the same event, don't despair just yet; `std::shared_future` allows exactly that. Whereas `std::future` is only *moveable*, so ownership can be transferred between instances, but only one instance refers to a particular asynchronous result at a time, `std::shared_future` instances are *copyable*, so you can have multiple objects referring to the same associated state.

Now, with `std::shared_future`, member functions on an individual object are still unsynchronized, so to avoid data races when accessing a single object from multiple threads, you must protect accesses with a lock. The preferred way to use it would be pass a copy of the `shared_future` object to each thread, so each thread can access its own local `shared_future` object safely, as the internals are now correctly synchronized by the library.. Accesses to the shared asynchronous state from multiple threads are safe if each thread accesses that state through its own `std::shared_future` object. See figure 4.1.

One potential use of `std::shared_future` is for implementing parallel execution of something akin to a complex spreadsheet; each cell has a single final value, which may be used by the formulas in multiple other cells. The formulas for calculating the results of the dependent cells can then use a `std::shared_future` to reference the first cell. If all the formulas for the individual cells are then executed in parallel, those tasks that can proceed to completion will do so, whereas those that depend on others will block until their dependencies are ready. This will thus allow the system to make maximum use of the available hardware concurrency.

Instances of `std::shared_future` that reference some asynchronous state are constructed from instances of `std::future` that reference that state. Since `std::future` objects don't

share ownership of the asynchronous state with any other object, the ownership must be transferred into the `std::shared_future` using `std::move`, leaving the `std::future` in an empty state, as if it was default constructed:

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());                                #1
std::shared_future<int> sf(std::move(f));
assert(!f.valid());                               #2
assert(sf.valid());    #3
```

#1 The future f is valid
#2 f is no longer valid
#3 sf is now valid

Here, the future `f` is initially valid #1 because it refers to the asynchronous state of the promise `p`, but after transferring the state to `sf`, `f` is no longer valid #2, whereas `sf` is #3.

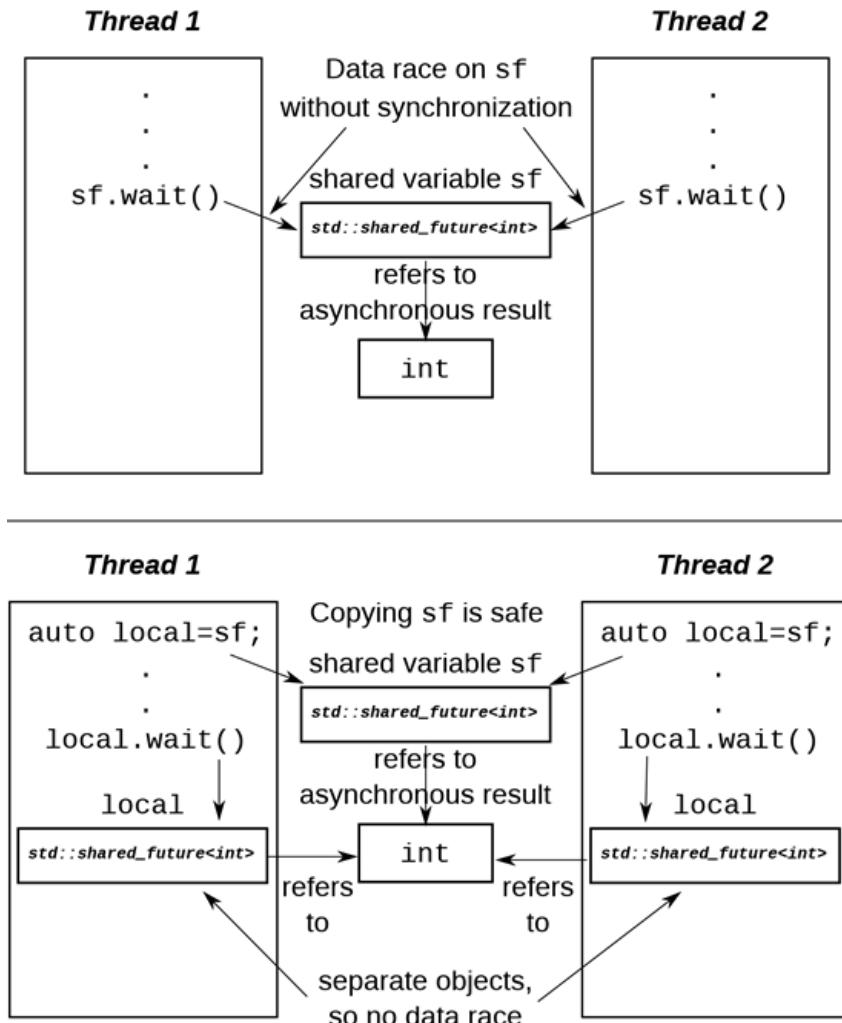


Figure 4.1 Using multiple `std::shared_future` objects to avoid data races

Just as with other movable objects, the transfer of ownership is implicit for rvalues, so you can construct a `std::shared_future` directly from the return value of the `get_future()` member function of a `std::promise` object, for example:

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future()); #1
```

#1 Implicit transfer of ownership

Here, the transfer of ownership is implicit; the `std::shared_future` is constructed from an rvalue of type `std::future<std::string> #1`.

`std::future` also has an additional feature to facilitate the use of `std::shared_future` with the new facility for automatically deducing the type of a variable from its initializer (see appendix A, section A.6). `std::future` has a `share()` member function that creates a new `std::shared_future` and transfers ownership to it directly. This can save a lot of typing and makes code easier to change:

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator,
    SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

In this case, the type of `sf` is deduced to be `std::shared_future< std::map< SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`, which is rather a mouthful. If the comparator or allocator is changed, you only need to change the type of the promise; the type of the future is automatically updated to match.

Sometimes you want to limit the amount of time you're waiting for an event, either because you have a hard time limit on how long a particular section of code may take, or because there's other useful work that the thread can be doing if the event isn't going to happen soon. To handle this facility, many of the waiting functions have variants that allow a timeout to be specified.

4.3 Waiting with a time limit

All the blocking calls introduced previously will block for an indefinite period of time, suspending the thread until the event being waited for occurs. In many cases this is fine, but in some cases you want to put a limit on how long you wait. This might be to allow you to send some form of "I'm still alive" message either to an interactive user or another process or indeed to allow you to abort the wait if the user has given up waiting and pressed Cancel.

There are two sorts of timeouts you may wish to specify: a *duration-based* timeout, where you wait for a specific amount of time (for example, 30 milliseconds), or an *absolute* timeout, where you wait until a specific point in time (for example, 17:30:15.045987023 UTC on November 30, 2011). Most of the waiting functions provide variants that handle both forms of timeouts. The variants that handle the duration-based timeouts have a `_for` suffix, and those that handle the absolute timeouts have a `_until` suffix.

So, for example, `std::condition_variable` has two overloads of the `wait_for()` member function and two overloads of the `wait_until()` member function that correspond to the two overloads of `wait()`—one overload that just waits until signaled, or the timeout expires, or a spurious wakeup occurs, and another that will check the supplied predicate when woken and will return only when the supplied predicate is `true` (and the condition variable has been signaled) or the timeout expires.

Before we look at the details of the functions that use the timeouts, let's examine the way that times are specified in C++, starting with clocks.

4.3.1 Clocks

As far as the C++ Standard Library is concerned, a clock is a source of time information. In particular, a clock is a class that provides four distinct pieces of information:

- The time *now*
- The type of the value used to represent the times obtained from the clock
- The tick period of the clock
- Whether or not the clock ticks at a uniform rate and is thus considered to be a *steady* clock

The current time of a clock can be obtained by calling the static member function `now()` for that clock class; for example, `std::chrono::system_clock::now()` will return the current time of the system clock. The type of the time points for a particular clock is specified by the `time_point` member `typedef`, so the return type of `some_clock::now()` is `some_clock::time_point`.

The tick period of the clock is specified as a fractional number of seconds, which is given by the `period` member `typedef` of the clock—a clock that ticks 25 times per second thus has a period of `std::ratio<1,25>`, whereas a clock that ticks every 2.5 seconds has a period of `std::ratio<5,2>`. If the tick period of a clock can't be known until runtime, or it may vary during a given run of the application, the `period` may be specified as the average tick period, smallest possible tick period, or some other value that the library writer deems appropriate. There's no guarantee that the observed tick period in a given run of the program matches the specified period for that clock.

If a clock *ticks at a uniform rate* (whether or not that rate matches the `period`) and *can't be adjusted*, the clock is said to be a *steady* clock. The `is_steady` static data member of the `clock` class is `true` if the clock is steady and `false` otherwise. Typically, `std::chrono::system_clock` will *not* be steady, because the clock can be adjusted, even if such adjustment is done automatically to take account of local clock drift. Such an adjustment may cause a call to `now()` to return a value earlier than that returned by a prior call to `now()`, which is in violation of the requirement for a uniform tick rate. Steady clocks are important for timeout calculations, as you'll see shortly, so the C++ Standard Library provides one in the form of `std::chrono::steady_clock`. The other clocks provided by the C++ Standard Library are `std::chrono::system_clock` (mentioned above), which represents the “real time” clock of the system and which provides functions for converting its time points to and from `time_t` values, and `std::chrono::high_resolution_clock`, which provides the smallest possible tick period (and thus the highest possible resolution) of all the library-supplied clocks. It may actually be a `typedef` to one of the other clocks. These clocks are defined in the `<chrono>` library header, along with the other time facilities.

We'll look at the representation of time points shortly, but first let's look at how durations are represented.

4.3.2 Durations

Durations are the simplest part of the time support; they're handled by the `std::chrono::duration<>` class template (all the C++ time-handling facilities used by the Thread Library are in the `std::chrono` namespace). The first template parameter is the type of the representation (such as `int`, `long`, or `double`), and the second is a fraction specifying how many seconds each unit of the duration represents. For example, a number of minutes stored in a `short` is `std::chrono::duration<short, std::ratio<60,1>>`, because there are 60 seconds in a minute. On the other hand, a count of milliseconds stored in a `double` is `std::chrono::duration<double, std::ratio <1,1000>>`, because each millisecond is 1/1000 of a second.

The Standard Library provides a set of predefined `typedefs` in the `std::chrono` namespace for various durations: `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes`, and `hours`. They all use a sufficiently large integral type for the representation chosen such that you can represent a duration of over 500 years in the appropriate units if you so desire. There are also `typedefs` for all the SI ratios from `std::atto` (10^{-18}) to `std::exa` (10^{18}) (and beyond, if your platform has 128-bit integer types) for use when specifying custom durations such as `std::duration<double, std::centi>` for a count of 1/100 of a second represented in a `double`.

For convenience, there are a number of predefined literal suffix operators for durations in the `std::chrono_literals` namespace, introduced with C++14. This can simplify code that uses hard-coded duration values. e.g.

```
using namespace std::chrono_literals;
auto one_day=24h;
auto half_an_hour=30min;
auto max_time_between_messages=30ms;
```

When used with integer literals, these suffixes are equivalent to using the predefined duration `typedefs`, so `15ns` and `std::chrono::nanoseconds(15)` are identical values. However, when used with floating-point literals, these suffixes create a suitably-scaled floating-point duration with an unspecified representation type. Thus `2.5min` will be `std::chrono::duration<some-floating-point-type, std::ratio<60,1>>`. If you are concerned about the range or precision of the implementation's chosen floating point type, then you will need to construct an object with suitable representation yourself, rather than using the convenience of the literal suffixes.

Conversion between durations is implicit where it does not require truncation of the value (so converting hours to seconds is OK, but converting seconds to hours is not). Explicit conversions can be done with `std::chrono::duration_cast<>`:

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
    std::chrono::duration_cast<std::chrono::seconds>(ms);
```

The result is truncated rather than rounded, so `s` will have a value of 54 in this example.

Durations support arithmetic, so you can add and subtract durations to get new durations or multiply or divide by a constant of the underlying representation type (the first template parameter). Thus `5*seconds(1)` is the same as `seconds(5)` or `minutes(1) - seconds(55)`. The count of the number of units in the duration can be obtained with the `count()` member function. Thus `std::chrono::milliseconds(1234).count()` is 1234.

Duration-based waits are done with instances of `std::chrono::duration<>`. For example, you can wait for up to 35 milliseconds for a future to be ready:

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready)
    do_something_with(f.get());
```

The wait functions all return a status to indicate whether the wait timed out or the waited-for event occurred. In this case, you're waiting for a future, so the function returns `std::future_status::timeout` if the wait times out, `std::future_status::ready` if the future is ready, or `std::future_status::deferred` if the future's task is deferred. The time for a duration-based wait is measured using a steady clock internal to the library, so 35 milliseconds means 35 milliseconds of elapsed time, even if the system clock was adjusted (forward or back) during the wait. Of course, the vagaries of system scheduling and the varying precisions of OS clocks means that the actual time between the thread issuing the call and returning from it may be much longer than 35 ms.

With durations under our belt, we can now move on to time points.

4.3.3 Time points

The time point for a clock is represented by an instance of the `std::chrono::time_point<>` class template, which specifies which clock it refers to as the first template parameter and the units of measurement (a specialization of `std::chrono::duration<>`) as the second template parameter. The value of a time point is the length of time (in multiples of the specified duration) since a specific point in time called the *epoch* of the clock. The epoch of a clock is a basic property but not something that's directly available to query or specified by the C++ Standard. Typical epochs include 00:00 on January 1, 1970 and the instant when the computer running the application booted up. Clocks may share an epoch or have independent epochs. If two clocks share an epoch, the `time_point` `typedef` in one class may specify the other as the clock type associated with the `time_point`. Although you can't find out when the epoch is, you *can* get the `time_since_epoch()` for a given `time_point`. This member function returns a duration value specifying the length of time since the clock epoch to that particular time point.

For example, you might specify a time point as `std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>`. This would hold the time relative to the system clock but measured in minutes as opposed to the native precision of the system clock (which is typically seconds or less).

You can add durations and subtract durations from instances of `std::chrono::time_point<>` to produce new time points, so `std::chrono::high_resolution_clock::`

`now() + std::chrono::nanoseconds(500)` will give you a time 500 nanoseconds in the future. This is good for calculating an absolute timeout when you know the maximum duration of a block of code, but there are multiple calls to waiting functions within it or nonwaiting functions that precede a waiting function but take up some of the time budget.

You can also subtract one time point from another that shares the same clock. The result is a duration specifying the length of time between the two time points. This is useful for timing blocks of code, for example:

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<"do_something() took "
    <<std::chrono::duration<double, std::chrono::seconds>(stop-start).count()
    <<" seconds"<<std::endl;
```

The clock parameter of a `std::chrono::time_point` instance does more than just specify the epoch, though. When you pass the time point to a wait function that takes an absolute timeout, the clock parameter of the time point is used to measure the time. This has important consequences when the clock is changed, because the wait tracks the clock change and won't return until the clock's `now()` function returns a value later than the specified timeout. If the clock is adjusted forward, this may reduce the total length of the wait (as measured by a steady clock), and if it's adjusted backward, this may increase the total length of the wait.

As you may expect, time points are used with the `_until` variants of the wait functions. The typical use case is as an offset from `some_clock::now()` at a fixed point in the program, although time points associated with the system clock can be obtained by converting from a `time_t` using the `std::chrono::system_clock::to_time_point()` static member function for scheduling operations at a user-visible time. For example, if you have a maximum of 500 milliseconds to wait for an event associated with a condition variable, you might do something like in the following listing.

Listing 4.11 Waiting for a condition variable with a timeout

```
#include <condition_variable>
#include <mutex>
#include <chrono>
std::condition_variable cv;
bool done;
std::mutex m;
bool wait_loop()
{
    auto const timeout= std::chrono::steady_clock::now()+
        std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done)
    {
        if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
            break;
    }
    return done;
}
```

```
}
```

This is the recommended way to wait for condition variables with a time limit, if you're not passing a predicate to the wait. This way, the overall length of the loop is bounded. As you saw in section 4.1.1, you need to loop when using condition variables if you don't pass in the predicate, in order to handle spurious wakeups. If you use `wait_for()` in a loop, you might end up waiting almost the full length of time before a spurious wake, and the next time through the wait time starts again. This may repeat any number of times, making the total wait time unbounded.

With the basics of specifying timeouts under your belt, let's look at the functions that you can use the timeout with.

4.3.4 Functions that accept timeouts

The simplest use for a timeout is to add a delay to the processing of a particular thread, so that it doesn't take processing time away from other threads when it has nothing to do. You saw an example of this in section 4.1, where you polled a "done" flag in a loop. The two functions that handle this are `std::this_thread::sleep_for()` and `std::this_thread::sleep_until()`. They work like a basic alarm clock: the thread goes to sleep either for the specified duration (with `sleep_for()`) or until the specified point in time (with `sleep_until()`). `sleep_for()` makes sense for examples like that from section 4.1, where something must be done periodically, and the elapsed time is what matters. On the other hand, `sleep_until()` allows you to schedule the thread to wake at a particular point in time. This could be used to trigger the backups at midnight, or the payroll print run at 6:00 a.m., or to suspend the thread until the next frame refresh when doing a video playback.

Of course, sleeping isn't the only facility that takes a timeout; you already saw that you can use timeouts with condition variables and futures. You can even use timeouts when trying to acquire a lock on a mutex if the mutex supports it. Plain `std::mutex` and `std::recursive_mutex` don't support timeouts on locking, but `std::timed_mutex` does, as does `std::recursive_timed_mutex`. Both these types support `try_lock_for()` and `try_lock_until()` member functions that try to obtain the lock within a specified time period or before a specified time point. Table 4.1 shows the functions from the C++ Standard Library that can accept timeouts, their parameters, and their return values. Parameters listed as `duration` must be an instance of `std::duration<>`, and those listed as `time_point` must be an instance of `std::time_point<>`.

:σηαρεδ_τιμεδ_μυτεξtry
_lock_forstd::timed_mutex
op ,

(δυρατιον)

Class / Namespace	Functions	Return Values
std::this_thread namespace	sleep_for(<i>duration</i>) sleep_until(<i>time_point</i>)	N/A
std::condition_variable or std::condition_variable_anywait_for(<i>lock</i> , <i>duration</i>) wait_until(<i>lock</i> , <i>time_point</i>)	std::cv_status::timeout or std::cv_status::no_timeout wait_for(<i>lock</i> , <i>duration</i> , <i>predicate</i>) wait_until(<i>lock</i> , <i>time_point</i> , <i>predicate</i>)	bool — the return value of the <i>predicate</i> when woken
std::timed_mutex, std::recursive_timed_mutex or std::shared_timed_mutextry_lock_for(<i>duration</i>) try_lock_until(<i>time_point</i>)	bool — true if the lock was acquired, false otherwise	
std::shared_timed_mutex	try_lock_shared_for(<i>duration</i>) try_lock_shared_until(<i>time_point</i>)	bool — true if the lock was acquired, false otherwise
std::unique_lock< TimedLockable>unique_lock(<i>lockable</i> , <i>duration</i>)	N/A — owns_lock() on the newly- constructed object returns true if the lock was acquired, false otherwise	
unique_lock(<i>lockable</i> , <i>time_point</i>)	try_lock_for(<i>duration</i>) try_lock_until(<i>time_point</i>)	bool — true if the lock was acquired, false otherwise

```

std::shared_lock< N/A — owns_lock() on the newly-
SharedTimedLockab constructed object returns true if the
le>shared_lock(lo lock was acquired, false otherwise
ckable,duration)
shared_lock(locka try_lock_for(duration) bool — true if the lock was
ble,time_point) try_lock_until(time_point) acquired, false otherwise

std::future<Value std::future_status::timeout if
Type> or the wait timed out,
std::shared_futur std::future_status::ready if the
e<ValueType>wait_ future is ready, or
for(duration) std::future_status::deferred if
wait_until(time_p the future holds a deferred function that
oint) hasn't yet started

```

Now that I've covered the mechanics of condition variables, futures, promises, and packaged tasks, it's time to look at the wider picture and how they can be used to simplify the synchronization of operations between threads.

4.4 Using synchronization of operations to simplify code

Using the synchronization facilities described so far in this chapter as building blocks allows you to focus on the operations that need synchronizing rather than the mechanics. One way this can help simplify your code is that it accommodates a much more *functional* (in the sense of *functional programming*) approach to programming concurrency. Rather than sharing data directly between threads, each task can be provided with the data it needs, and the result can be disseminated to any other threads that need it through the use of futures.

4.4.1 Functional programming with futures

The term *functional programming* (FP) refers to a style of programming where the result of a function call depends solely on the parameters to that function and doesn't depend on any external state. This is related to the mathematical concept of a function, and it means that if you invoke a function twice with the same parameters, the result is exactly the same. This is a property of many of the mathematical functions in the C++ Standard Library, such as `sin`, `cos`, and `sqrt`, and simple operations on basic types, such as `3+3`, `6*9`, or `1.3/4.7`. A *pure* function doesn't *modify* any external state either; the effects of the function are entirely limited to the return value.

This makes things easy to think about, especially when concurrency is involved, because many of the problems associated with shared memory discussed in chapter 3 disappear. If there are no modifications to shared data, there can be no race conditions and thus no need to protect shared data with mutexes either. This is such a powerful simplification that

programming languages such as Haskell,² where all functions are pure by default, are becoming increasingly popular for programming concurrent systems. Because most things are pure, the *impure* functions that actually do modify the shared state stand out all the more, and it's therefore easier to reason about how they fit into the overall structure of the application.

The benefits of functional programming aren't limited to those languages where it's the default paradigm, however. C++ is a multiparadigm language, and it's entirely possible to write programs in the FP style. This is even easier in C++11 than it was in C++98, with the advent of lambda functions (see appendix A, section A.6), the incorporation of `std::bind` from Boost and TR1, and the introduction of automatic type deduction for variables (see appendix A, section A.7). Futures are the final piece of the puzzle that makes FP-style concurrency viable in C++; a future can be passed around between threads to allow the result of one computation to depend on the result of another, *without any explicit access to shared data*.

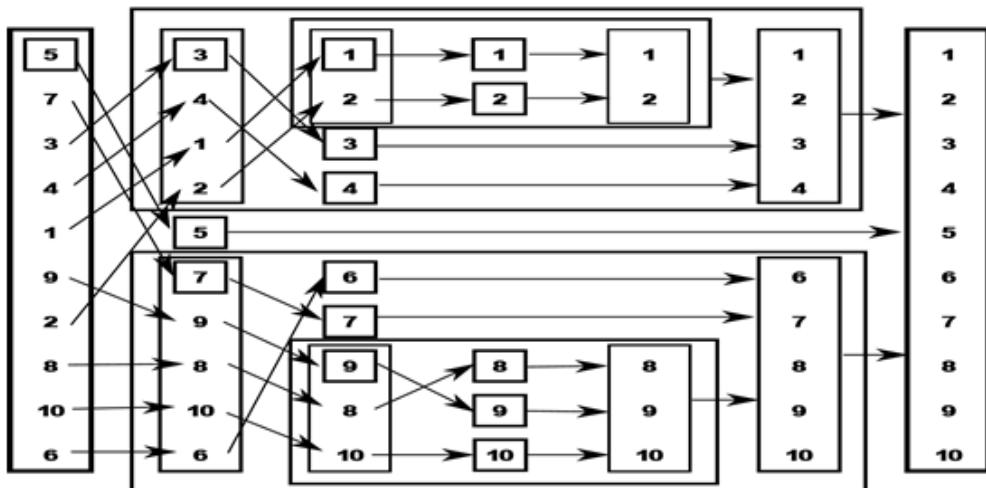


Figure 4.2 FP-style recursive sorting

FP-STYLE QUICKSORT

To illustrate the use of futures for FP-style concurrency, let's look at a simple implementation of the Quicksort algorithm. The basic idea of the algorithm is simple: given a list of values, take an element to be the pivot element, and then partition the list into two sets—those less than the pivot and those greater than or equal to the pivot. A sorted copy of the list is obtained by sorting the two sets and returning the sorted list of values less than the pivot, followed by the pivot, followed by the sorted list of values greater than or equal to the

² See <http://www.haskell.org/>.

pivot. Figure 4.2 shows how a list of 10 integers is sorted under this scheme. An FP-style sequential implementation is shown in the following listing; it takes and returns a list by value rather than sorting in place like `std::sort()` does.

Listing 4.12 A sequential implementation of Quicksort

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());          #1
    T const& pivot=*result.begin();                            #2

    auto divide_point=std::partition(input.begin(),input.end(),
        [&](T const& t){return t<pivot;});                      #3
    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);                                         #4
    auto new_lower(
        sequential_quick_sort(std::move(lower_part)));       #5
    auto new_higher(
        sequential_quick_sort(std::move(input)));            #6
    result.splice(result.end(),new_higher);                  #7
    result.splice(result.begin(),new_lower);                #8
    return result;
}
```

Although the interface is FP-style, if you used FP-style throughout you'd do a lot of copying, so you use "normal" imperative style for the internals. You take the first element as the pivot by slicing it off the front of the list using `splice()` #1. Although this can potentially result in a suboptimal sort (in terms of numbers of comparisons and exchanges), doing anything else with a `std::list` can add quite a bit of time because of the list traversal. You know you're going to want it in the result, so you can splice it directly into the list you'll be using for that. Now, you're also going to want to use it for comparisons, so let's take a reference to it to avoid copying #2. You can then use `std::partition` to divide the sequence into those values *less than* the pivot and those *not less* than the pivot #3. The easiest way to specify the partition criteria is to use a lambda function; you use a reference capture to avoid copying the `pivot` value (see appendix A, section A.5 for more on lambda functions).

`std::partition()` rearranges the list in place and returns an iterator marking the first element that's *not less* than the pivot value. The full type for an iterator can be quite long-winded, so you just use the `auto` type specifier to force the compiler to work it out for you (see appendix A, section A.7).

Now, you've opted for an FP-style interface, so if you're going to use recursion to sort the two "halves," you'll need to create two lists. You can do this by using `splice()` again to move the values from `input` up to the `divide_point` into a new list: `lower_part` #4. This leaves the remaining values alone in `input`. You can then sort the two lists with recursive calls #5, #6.

By using `std::move()` to pass the lists in, you can avoid copying here too—the result is implicitly moved out anyway. Finally, you can use `splice()` yet again to piece the result together in the right order. The `new_higher` values go on the end #7, after the pivot, and the `new_lower` values go at the beginning, before the pivot #8.

FP-STYLE PARALLEL QUICKSORT

Because this uses a functional style already, it's now easy to convert this to a parallel version using futures, as shown in the next listing. The set of operations is the same as before, except that some of them now run in parallel. This version uses an implementation of the Quicksort algorithm using futures and a functional style.

Listing 4.13 Parallel Quicksort using futures

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());
    T const& pivot=*result.begin();
    auto divide_point=std::partition(input.begin(),input.end(),
        [&](T const &t){return t<pivot;});
    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);
    std::future<std::list<T> > new_lower(                      #1
        std::async(&parallel_quick_sort<T>,std::move(lower_part)));
    auto new_higher(
        parallel_quick_sort(std::move(input)));    #2
    result.splice(result.end(),new_higher);          #3
    result.splice(result.begin(),new_lower.get());    #4
    return result;
}
```

The big change here is that rather than sorting the lower portion on the current thread, you sort it on another thread using `std::async()` #1. The upper portion of the list is sorted with direct recursion as before #2. By recursively calling `parallel_quick_sort()`, you can take advantage of the available hardware concurrency. If `std::async()` starts a new thread every time, then if you recurse down three times, you'll have eight threads running; if you recurse down 10 times (for ~1000 elements), you'll have 1024 threads running if the hardware can handle it. If the library decides there are too many spawned tasks (perhaps because the number of tasks has exceeded the available hardware concurrency), it may switch to spawning the new tasks synchronously. They will run in the thread that calls `get()` rather than on a new thread, thus avoiding the overhead of passing the task to another thread when this won't help the performance. It's worth noting that it's perfectly conforming for an implementation of `std::async` to start a new thread for each task (even in the face of massive

oversubscription) unless `std::launch::deferred` is explicitly specified or to run all tasks synchronously unless `std::launch::async` is explicitly specified. If you're relying on the library for automatic scaling, you're advised to check the documentation for your implementation to see what behavior it exhibits.

Rather than using `std::async()`, you could write your own `spawn_task()` function as a simple wrapper around `std::packaged_task` and `std::thread`, as shown in listing 4.14; you'd create a `std::packaged_task` for the result of the function call, get the future from it, run it on a thread, and return the future. This wouldn't itself offer much advantage (and indeed would likely lead to massive oversubscription), but it would pave the way to migrate to a more sophisticated implementation that adds the task to a queue to be run by a pool of worker threads. We'll look at thread pools in chapter 9. It's probably worth going this way in preference to using `std::async` only if you really know what you're doing and want complete control over the way the thread pool is built and executes tasks.

Anyway, back to `parallel_quick_sort`. Because you just used direct recursion to get `new_higher`, you can just splice it into place as before #3. But `new_lower` is now a `std::future<std::list<T>>` rather than just a list, so you need to call `get()` to retrieve the value before you can call `splice()` #4. This then waits for the background task to complete and moves the result into the `splice()` call; `get()` returns an rvalue reference to the contained result, so it can be moved out (see appendix A, section A.1.1 for more on rvalue references and move semantics).

Even assuming that `std::async()` makes optimal use of the available hardware concurrency, this still isn't an ideal parallel implementation of Quicksort. For one thing, `std::partition` does a lot of the work, and that's still a sequential call, but it's good enough for now. If you're interested in the fastest possible parallel implementation, check the academic literature. Alternatively, you could just use the parallel overload from the C++17 Standard Library (see chapter 10).

Listing 4.14 A sample implementation of `spawn_task`

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
spawn_task(F&& f,A&& a)
{
    typedef std::result_of<F(A&&)>::type result_type;
    std::packaged_task<result_type(A&&)>
        task(std::move(f));
    std::future<result_type> res(task.get_future());
    std::thread t(std::move(task),std::move(a));
    t.detach();
    return res;
}
```

Functional programming isn't the only concurrent programming paradigm that eschews shared mutable data; another paradigm is CSP (Communicating Sequential Processes),³ where

³ *Communicating Sequential Processes*, C.A.R. Hoare, Prentice Hall, 1985. Available free online at <http://www.usingcsp.com/cspbook.pdf>.

threads are conceptually entirely separate, with no shared data but with communication channels that allow messages to be passed between them. This is the paradigm adopted by the programming language Erlang (<http://www.erlang.org/>) and by the MPI (Message Passing Interface) (<http://www.mpi-forum.org/>) environment commonly used for high-performance computing in C and C++. I'm sure that by now you'll be unsurprised to learn that this can also be supported in C++ with a bit of discipline; the following section discusses one way to achieve this.

4.4.2 Synchronizing operations with message passing

The idea of CSP is simple: if there's no shared data, each thread can be reasoned about entirely independently, purely on the basis of how it behaves in response to the messages that it received. Each thread is therefore effectively a state machine: when it receives a message, it updates its state in some manner and maybe sends one or more messages to other threads, with the processing performed depending on the initial state. One way to write such threads would be to formalize this and implement a Finite State Machine model, but this isn't the only way; the state machine can be implicit in the structure of the application. Which method works better in any given scenario depends on the exact behavioral requirements of the situation and the expertise of the programming team. However you choose to implement each thread, the separation into independent processes has the potential to remove much of the complication from shared-data concurrency and therefore make programming easier, lowering the bug rate.

True communicating sequential processes have no shared data, with all communication passed through the message queues, but because C++ threads share an address space, it's not possible to enforce this requirement. This is where the discipline comes in: as application or library authors, it's our responsibility to ensure that we don't share data between the threads. Of course, the message queues must be shared in order for the threads to communicate, but the details can be wrapped in the library.

Imagine for a moment that you're implementing the code for an ATM. This code needs to handle interaction with the person trying to withdraw money and interaction with the relevant bank, as well as control the physical machinery to accept the person's card, display appropriate messages, handle key presses, issue money, and return their card.

One way to handle everything would be to split the code into three independent threads: one to handle the physical machinery, one to handle the ATM logic, and one to communicate with the bank. These threads could communicate purely by passing messages rather than sharing any data. For example, the thread handling the machinery would send a message to the logic thread when the person at the machine entered their card or pressed a button, and the logic thread would send a message to the machinery thread indicating how much money to dispense, and so forth.

One way to model the ATM logic would be as a state machine. In each state the thread waits for an acceptable message, which it then processes. This may result in transitioning to a new state, and the cycle continues. The states involved in a simple implementation are shown

in figure 4.3. In this simplified implementation, the system waits for a card to be inserted. Once the card is inserted, it then waits for the user to enter their PIN, one digit at a time. They can delete the last digit entered. Once enough digits have been entered, the PIN is verified. If the PIN is not OK, you're finished, so you return the card to the customer and resume waiting for someone to enter their card. If the PIN is OK, you wait for them to either cancel the transaction or select an amount to withdraw. If they cancel, you're finished, and you return their card. If they select an amount, you wait for confirmation from the bank before issuing the cash and returning the card or displaying an "insufficient funds" message and returning their card. Obviously, a real ATM is considerably more complex, but this is enough to illustrate the idea.

Having designed a state machine for your ATM logic, you can implement it with a class that has a member function to represent each state. Each member function can then wait for specific sets of incoming messages and handle them when they arrive, possibly triggering a switch to another state. Each distinct message type is represented by a separate struct. Listing 4.15 shows part of a simple implementation of the ATM logic in such a system, with the main loop and the implementation of the first state, waiting for the card to be inserted.

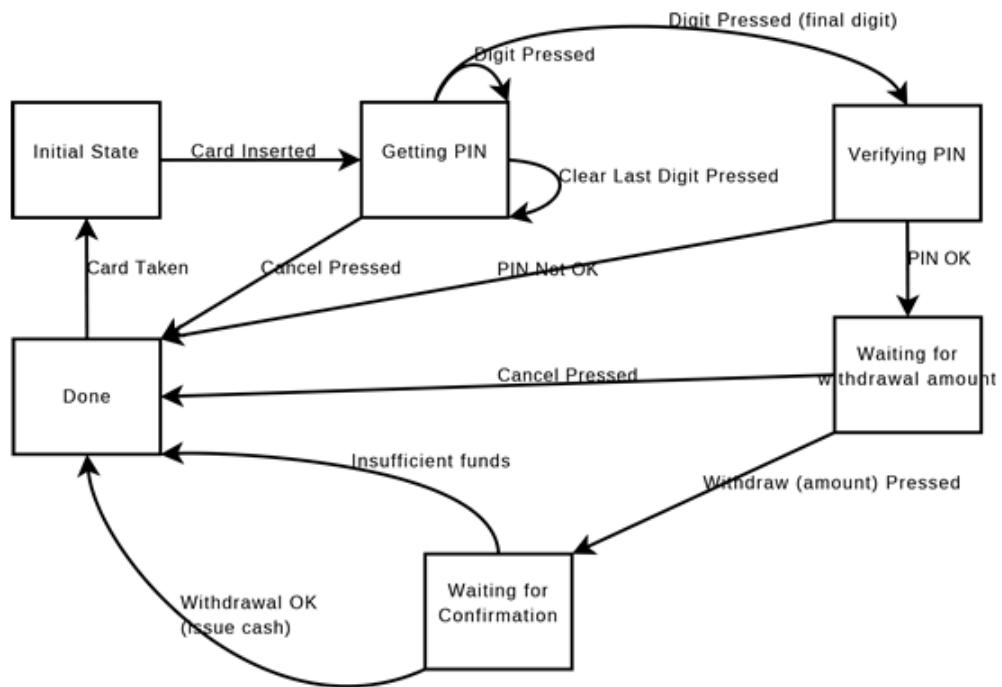


Figure 4.13 A simple state machine model for an ATM

As you can see, all the necessary synchronization for the message passing is entirely hidden inside the message-passing library (a basic implementation of which is given in appendix C, along with the full code for this example).

Listing 4.15 A simple implementation of an ATM logic class

```
struct card_inserted
{
    std::string account;
};

class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface_hardware;
    void (*atm::*state)();
    std::string account;
    std::string pin;
    void waiting_for_card() #1
    {
        interface_hardware.send(display_enter_card()); #2
        incoming.wait() #3
            .handle<card_inserted>(
                [&](card_inserted const& msg) #4
                {
                    account=msg.account;
                    pin="";
                    interface_hardware.send(display_enter_pin());
                    state=&atm::getting_pin;
                }
            );
    }
    void getting_pin();
public:
    void run() #5
    {
        state=&atm::waiting_for_card; #6
        try
        {
            for(;;)
            {
                (*this->*state)(); #7
            }
        }
        catch(messaging::close_queue const&)
        {
        }
    }
};
```

As already mentioned, the implementation described here is grossly simplified from the real logic that would be required in an ATM, but it does give you a feel for the message-passing style of programming. There's no need to think about synchronization and concurrency issues, just which messages may be received at any given point and which messages to send. The state machine for this ATM logic runs on a single thread, with other

parts of the system such as the interface to the bank and the terminal interface running on separate threads. This style of program design is called the *Actor model*—there are several discrete *actors* in the system (each running on a separate thread), which send messages to each other to perform the task at hand, and there's no shared state except that directly passed via messages.

Execution starts with the `run()` member function #5, which sets the initial state to `waiting_for_card` #6 and then repeatedly executes the member function representing the current state (whatever it is) #7. The state functions are simple member functions of the `atm` class. The `waiting_for_card` state function #1 is also simple: it sends a message to the interface to display a “waiting for card” message #2 and then waits for a message to handle #3. The only type of message that can be handled here is a `card_inserted` message, which you handle with a lambda function #4. You could pass any function or function object to the `handle()` function, but for a simple case like this, it's easiest to use a lambda. Note that the `handle()` function call is chained onto the `wait()` function; if a message is received that doesn't match the specified type, it's discarded, and the thread continues to wait until a matching message is received.

The lambda function itself just caches the account number from the card in a member variable, clears the current PIN, sends a message to the interface hardware to display something asking the user to enter their PIN, and changes to the “getting PIN” state. Once the message handler has completed, the state function returns, and the main loop then calls the new state function #7.

The `getting_pin` state function is a bit more complex in that it can handle three distinct types of message, as in figure 4.3. This is shown in the following listing.

Listing 4.16 The `getting_pin` state function for the simple ATM implementation

```
void atm::getting_pin()
{
    incoming.wait()
        .handle<digit_pressed>(#1
            [&](digit_pressed const& msg)
        {
            unsigned const pin_length=4;
            pin+=msg.digit;
            if(pin.length()==pin_length)
            {
                bank.send(verify_pin(account,pin,incoming));
                state=&atm::verifying_pin;
            }
        })
        .handle<clear_last_pressed>(#2
            [&](clear_last_pressed const& msg)
        {
            if(!pin.empty())
            {
                pin.resize(pin.length()-1);
            }
        })
}
```

```

        )
    .handle<cancel_pressed>(#3
        [&](cancel_pressed const& msg)
    {
        state=&atm::done_processing;
    });
}

```

This time, there are three message types you can process, so the `wait()` function has three `handle()` calls chained on the end #1, #2, #3. Each call to `handle()` specifies the message type as the template parameter and then passes in a lambda function that takes that particular message type as a parameter. Because the calls are chained together in this way, the `wait()` implementation knows that it's waiting for a `digit_pressed` message, a `clear_last_pressed` message, or a `cancel_pressed` message. Messages of any other type are again discarded.

This time, you don't necessarily change state when you get a message. For example, if you get a `digit_pressed` message, you just add it to the `pin` unless it's the final digit. The main loop #7 in listing 4.15) will then call `getting_pin()` again to wait for the next digit (or clear or cancel).

This corresponds to the behavior shown in figure 4.3. Each state box is implemented by a distinct member function, which waits for the relevant messages and updates the state as appropriate.

As you can see, this style of programming can greatly simplify the task of designing a concurrent system, because each thread can be treated entirely independently. It is thus an example of using multiple threads to separate concerns and as such requires you to explicitly decide how to divide the tasks between threads.

Back in section 4.2, I mentioned that the Concurrency TS provides extended versions of futures. The core part of the extensions is the ability to specify *continuations* — additional functions that are run automatically when the future becomes *ready*. Let us now take the opportunity to explore how this can simplify our code.

4.5 Continuation-style concurrency with the Concurrency TS

The Concurrency TS provides new versions of `std::promise` and `std::packaged_task` in the `std::experimental` namespace that all differ from their `std` originals in the same way: they return instances of `std::experimental::future` rather than `std::future`. This enables users to take advantage of the key new feature in `std::experimental::future` — *continuations*.

Suppose we have a task running that will produce some result, and a future that will hold the result when it becomes available. We then have some code that needs to run in order to process that result. With `std::future` we would have to wait for the future to become ready, either with the fully-blocking `wait()` member function or either of the `wait_for()` or `wait_until()` member functions to allow a wait with a timeout. This can be inconvenient, and complicate the code. What we want is a means of saying “when the data is ready, *then* do this

processing". This is exactly what continuations give us; unsurprisingly, the member function to add a continuation to a future is called `then()`. Given a future `fut`, a continuation is thus added with the call `fut.then(continuation)`.

Just like `std::future`, `std::experimental::future` only allows the stored value to be retrieved once. If that value is being consumed by a continuation, this means it cannot be accessed by other code. Consequently, when a continuation is added with `fut.then()`, the original future `fut` becomes *invalid*. Instead, the call to `fut.then()` returns a new future to hold the result of the continuation call. This is shown in the code below:

```
std::experimental::future<int> find_the_answer();
auto fut=find_the_answer();
auto fut2=fut.then(find_the_question);
assert(!fut.valid());
assert(fut2.valid());
```

The continuation function `find_the_question` is scheduled to run "on an unspecified thread" when the original future is *ready*. This gives the implementation freedom to run it on a thread pool, or other library-managed thread. As it stands, this gives the implementation a lot of freedom; this is deliberate, with the intention that when continuations are added to a future C++ Standard, the implementors will be able to draw on their experience to better specify the choice of threads, and provide users with suitable mechanisms for controlling the choice of threads.

Unlike direct calls to `std::async`, or `std::thread`, you cannot pass arguments to a continuation function, because the argument is already defined by the library — the continuation is passed a *ready* future that holds the result that triggered the continuation. Assuming our `find_the_answer` function returns an `int`, the `find_the_question` function referenced in the previous example must take a `std::experimental::future<int>` as its sole parameter. e.g.

```
std::string find_the_question(std::experimental::future<int> the_answer);
```

The reason for this is that the future on which the continuation was chained may end up holding a value or an exception. If the future was implicitly dereferenced to pass the value directly to the continuation, then the library would have to decide how to handle the exception, whereas by passing the future to the continuation, the continuation can handle the exception. In simple cases, this may be done just by calling `fut.get()`, and allowing the re-thrown exception to propagate out of the continuation function. Just as for functions passed to `std::async`, exceptions that escape a continuation are stored in the future that holds the continuation result.

Note that the Concurrency TS doesn't specify that there is an equivalent to `std::async`, though implementations may of course provide one as an extension. Writing such a function is fairly straightforward: use a `std::experimental::promise` to obtain a future, and then spawn a new thread running a lambda that sets the promise's value to the return value of the supplied function, as in listing 4.17.

Listing 4.17 A simple equivalent to std::async for Concurrency TS futures

```
template<typename Func>
std::experimental::future<decltype(std::declval<Func>()())>
spawn_async(Func&& func){
    std::experimental::promise<
        decltype(std::declval<Func>()())> p;
    auto res=p.get_future();
    std::thread t(
        [p=std::move(p),f=std::decay_t<Func>(func)]()
        mutable{
            try{
                p.set_value_at_thread_exit(f());
            } catch(...){
                p.set_exception_at_thread_exit(std::current_exception());
            }
        });
    t.detach();
    return res;
}
```

This stores the result of the function in the future, or catches the exception thrown from the function, and stores that in the future, just as `std::async` does. Also, it uses `set_value_at_thread_exit` and `set_exception_at_thread_exit` to ensure that `thread_local` variables have been properly cleaned up before the future becomes *ready*.

The value returned from a `then()` call is a fully-fledged future itself. This of course means that you can chain continuations.

4.5.1 Chaining continuations

Suppose you have a series of time-consuming tasks to do, and you want to do them asynchronously, in order to free up the “main” thread for other tasks. For example, when the user logs in to your application, you might need to send the credentials to the backend for authentication, then, when the details have been authenticated, make a further request to the backend for information about the user’s account, and then, finally, when that information has been retrieved, update the display with the relevant information. As sequential code, you might write something like listing 4.18.

Listing 4.181 A simple sequential function to process user login

```
void process_login(std::string const& username,std::string const& password)
{
    try {
        user_id const id=backend.authenticate_user(username,password);
        user_data const info_to_display=backend.request_current_info(id);
        update_display(info_to_display);
    } catch(std::exception& e){
        display_error(e);
    }
}
```

However, we don’t want sequential code; we want asynchronous code, so we’re not blocking the UI thread. With plain `std::async`, we could punt it all to a background thread like

listing 4.19, but that would still block that thread, consuming resources while waiting for the tasks to complete. If we have many such tasks, then we can end up with a large number of threads that are doing nothing except waiting.

Listing 4.192 Processing user login with a single async task

```
std::future<void> process_login(
    std::string const& username, std::string const& password)
{
    return std::async(std::launch::async, [=](){
        try {
            user_id const id=backend.authenticate_user(username,password);
            user_data const info_to_display=
                backend.request_current_info(id);
            update_display(info_to_display);
        } catch(std::exception& e){
            display_error(e);
        }
    });
}
```

In order to avoid all these blocked threads, we need some mechanism for chaining tasks as they each complete: continuations. Listing 4.20 shows the same overall process, but this time split into a series of tasks, each of which is then chained on the previous one as a continuation.

Listing 4.203 A function to process user login with continuations

```
std::experimental::future<void> process_login(
    std::string const& username, std::string const& password)
{
    return spawn_async([=](){
        return backend.authenticate_user(username,password);
    }).then([](std::experimental::future<user_id> id){
        return backend.request_current_info(id.get());
    }).then([](std::experimental::future<user_data> info_to_display){
        try{
            update_display(info_to_display.get());
        } catch(std::exception& e){
            display_error(e);
        }
    });
}
```

Note how each continuation takes a `std::experimental::future` as the sole parameter, and then uses `.get()` to retrieve the contained value. This means that exceptions get propagated all the way down the chain, so the call to `info_to_display.get()` in the final continuation will throw if any of the functions in the chain threw an exception, and the catch block here can handle all the exceptions, just like the catch block in listing 4.18 did.

If the function calls to the backend block internally, because they're waiting for messages to cross the network, or for a database operation to complete, then we're not done yet. We may have split the task into its individual parts, but they're still blocking calls, so we still get

blocked threads. What we need is for the backend calls to return futures themselves, which become ready when the data is ready, without blocking any threads. In this case, then `backend.async_authenticate_user(username,password)` will now return a `std::experimental::future<user_id>` rather than a plain `user_id`.

You might think this would complicate the code, since returning a future from a continuation would give us a `future<future<some_value>>`, or else we'd have to put the `.then` calls inside the continuations. Thankfully, if you thought that, then you'd be mistaken: the continuation support has a nifty feature called future-unwrapping. If the continuation function you passed to a `.then()` call returns a `future<some_type>`, then the `.then()` call will just return a `future<some_type>` in turn. This means our final code looks like listing 4.21, and there is no blocking in our asynchronous function chain.

Listing 4.214 A function to process user login with fully asynchronous operations

```
std::experimental::future<void> process_login(
    std::string const& username, std::string const& password)
{
    return backend.async_authenticate_user(username,password).then(
        [](std::experimental::future<user_id> id){
            return backend.async_request_current_info(id.get());
        }).then([](std::experimental::future<user_data> info_to_display){
            try{
                update_display(info_to_display.get());
            } catch(std::exception& e){
                display_error(e);
            }
        });
}
```

This is almost as straight-forward as the sequential code from listing 4.18, with just a little bit more boilerplate around the `.then` calls and the lambda declarations. If your compiler supports C++14 generic lambdas, then the types of the futures in the lambda parameters can be replaced with `auto`, which simplifies the code even further. e.g.

```
return backend.async_authenticate_user(username,password).then(
    [](auto id){
        return backend.async_request_current_info(id.get());
    });
}
```

If you need anything more complex than simple linear control flow, then you can implement this by putting the logic in one of the lambdas; for truly complex control flow you probably need to write a separate function.

So far, we've focused on the continuation support in `std::experimental::future`. As you might expect, `std::experimental::shared_future` also supports continuations. The difference here is that `std::experimental::shared_future` objects can have more than one continuation, and the continuation parameter is a `std::experimental::shared_future` rather than a `std::experimental::future`. This naturally falls out of the shared nature of `std::experimental::shared_future` — since multiple objects can refer to the same shared state, if only one continuation was allowed, there would be a race condition between two

threads that were each trying to add continuations to their own `std::experimental::shared_future` objects. This is obviously undesirable, so multiple continuations are permitted. Once you have multiple continuations permitted, you may as well allow them to be added via the same `std::experimental::shared_future` instance, rather than only allowing one continuation per object. In addition, you cannot package the shared state in a one-shot `std::experimental::future` passed to the first continuation, when you're still going to want to also pass it to the second continuation. Thus the parameter passed to the continuation function must also be a `std::experimental::shared_future`. e.g.

```
auto fut=spawn_async(some_function).share();
auto fut2=fut.then([](std::experimental::shared_future<some_data> data){
    do_stuff(data);
});
auto fut3=fut.then([](std::experimental::shared_future<some_data> data){
    return do_other_stuff(data);
});
```

`fut` is a `std::experimental::shared_future` due to the `share()` call, so the continuation function must take a `std::experimental::shared_future` as its parameter. However, the return value **from** the continuation is just a plain `std::experimental::future` — that value isn't currently shared until you do something to share it — so both `fut2` and `fut3` are just `std::experimental::futures`.

Continuations aren't the only enhancement to futures in the Concurrency TS, though they are probably the most important. Also provided are two overloaded functions which allow us to wait for either **any one** of a bunch of futures to become ready, or **all** of a bunch of futures to become ready.

4.5.2 Waiting for more than one future

Suppose you've got a large volume of data to process, and each item can be processed independently. This is a prime opportunity to make use of the available hardware by spawning a set of asynchronous tasks to process the data items, each of them returning the processed data via a future. However, if you need to wait for all the tasks to finish and then gather all the results for some final processing, then this can be inconvenient — you have to wait for each future in turn, and then gather the results. If you wanted to do the result gathering with another asynchronous task, then you either have to spawn it up front, so it is occupying a thread just waiting, or you have to keep polling the futures, and spawn the new task when all the futures are *ready*. An example of such code is shown in listing 4.22.

Listing 4.22 Gathering results from futures using `std::async`

```
std::future<FinalResult> process_data(std::vector<MyData>& vec)
{
    size_t const chunk_size=whatever;
    std::vector<std::future<ChunkResult>> results;
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-beg;
```

```

        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            std::async(process_chunk,begin,begin+this_chunk_size));
        begin+=this_chunk_size;
    }
    return std::async([all_results=std::move(results)](){
        std::vector<ChunkResult> v;
        v.reserve(all_results.size());
        for(auto& f: all_results)
        {
            v.push_back(f.get());                      #1
        }
        return gather_results(v);
    });
}

```

This code spawns a new asynchronous task to wait for the results, and then processes them when they are all available. However, since it waits for each task individually, it will repeatedly be woken by the scheduler at #1 as each result becomes available, and then go back to sleep again when it finds another result that is not yet ready. Not only does this occupy a thread doing the waiting, but it adds additional context switches as each future becomes ready, which adds additional overhead.

With `std::experimental::when_all`, this waiting and switching can be avoided. You pass the set of futures to be waited on to `when_all`, and it returns a new future that becomes *ready* when all the futures in the set are *ready*. This future can then be used with continuations to schedule additional work when the all the futures are ready. See for example listing 4.23.

Listing 4.23 Gathering results from futures using `std::experimental::when_all`

```

std::experimental::future<FinalResult> process_data(
    std::vector<MyData>& vec)
{
    size_t const chunk_size=whatever;
    std::vector<std::experimental::future<ChunkResult>> results;
    for(auto begin=vec.begin(),end=vec.end();beg!=end;){
        size_t const remaining_size=end-begin;
        size_t const this_chunk_size=std::min(remaining_size,chunk_size);
        results.push_back(
            spawn_async(
                process_chunk,begin,begin+this_chunk_size));
        begin+=this_chunk_size;
    }
    return std::experimental::when_all(
        results.begin(),results.end()).then(          #1
        [](std::future<std::vector<
            std::experimental::future<ChunkResult>>> ready_results)
    {
        std::vector<std::experimental::future<ChunkResult>>
            all_results=ready_results .get();
        std::vector<ChunkResult> v;
        v.reserve(all_results.size());
        for(auto& f: all_results)
        {
            v.push_back(f.get());                  #2
        }
    });
}

```

```

        return gather_results(v);
    });
}

```

In this case, we use `when_all` to wait for all the futures to become ready, and then schedule the function using `.then` rather than `async` (#1). Though the lambda is superficially the same, it takes the results vector as a parameter (wrapped in a future) rather than as a capture, and the calls to get on the futures at #2 do not block, as all the values are *ready* by the time execution gets there. This has the potential to reduce the load on the system for very little change in the code.

To complement `when_all`, we also have `when_any`. This creates a future that becomes *ready* when *any* of the supplied futures becomes *ready*. This works well for scenarios where you've spawned multiple tasks to take advantage of the available concurrency, but need to do something when the first one becomes *ready*.

4.5.3 Waiting for the first future in a set with `when_any`

Suppose you are searching a large data set for a value that meets particular criteria, but if there are multiple such values then *any* will do. This is a prime target for parallelism — you can spawn multiple threads, each of which checks a subset of the data; if a given thread finds a suitable value then it sets a flag indicating that the other threads should stop their search, and then sets the final return value. In this case, we want to do the further processing when the first task completes its search, even if the other tasks haven't finished cleaning up yet.

Here, we can use `std::experimental::when_any` to gather the futures together, and give us a new future that is *ready* when at least one of the original set is *ready*. Whereas `when_all` gave us a future that just wrapped the collection of futures we passed in, `when_any` adds a further layer, combining the collection with an index value that indicates which future was the one that triggered the combined future to be *ready* into an instance of the `std::experimental::when_any_result` class template.

An example of using `when_any` as described here is shown in listing 4.24.

Listing 4.24 Using `std::experimental::when_any` to process the first value found

```

std::experimental::future<FinalResult>
find_and_process_value(std::vector<MyData> &data)
{
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_tasks = (concurrency > 0) ? concurrency : 2;
    std::vector<std::experimental::future<MyData *>> results;
    auto const chunk_size = (data.size() + num_tasks - 1) / num_tasks;
    auto chunk_begin = data.begin();
    std::shared_ptr<std::atomic<bool>> done_flag =
        std::make_shared<std::atomic<bool>>(false);
    for (unsigned i = 0; i < num_tasks; ++i) {           #1
        auto chunk_end =
            (i < (num_tasks - 1)) ? chunk_begin + chunk_size : data.end();
        results.push_back(spawn_async([=] { #2
            for (auto entry = chunk_begin;
                  !*done_flag && (entry != chunk_end);

```

```

        ++entry) {
            if (matches_find_criteria(*entry)) {
                *done_flag = true;
                return &*entry;
            }
        }
        return (MyData *)nullptr;
    });
    chunk_begin = chunk_end;
}
std::shared_ptr<std::experimental::promise<FinalResult>> final_result =
    std::make_shared<std::experimental::promise<FinalResult>>();
struct DoneCheck {
    std::shared_ptr<std::experimental::promise<FinalResult>>
        final_result;
    DoneCheck(
        std::shared_ptr<std::experimental::promise<FinalResult>>
            final_result_)
        : final_result(std::move(final_result_)) {}

    void operator()#4
        std::experimental::future<std::experimental::when_any_result<
            std::vector<std::experimental::future<MyData *>>>>
            results_param) {
        auto results = results_param.get();
        MyData *const ready_result =
            results.futures[results.index].get();#5
        if (ready_result)
            final_result->set_value(#6
                process_found_value(*ready_result));
        else {
            results.futures.erase(
                results.futures.begin() + results.index);#7
            if (!results.futures.empty()) {
                std::experimental::when_any(#8
                    results.futures.begin(), results.futures.end())
                    .then(std::move(*this));
            } else {
                final_result->set_exception(
                    std::make_exception_ptr(#9
                        std::runtime_error("Not found")));
            }
        }
    };
    std::experimental::when_any(results.begin(), results.end())
        .then(DoneCheck(final_result));#3
    return final_result->get_future();#10
}

```

The initial loop (#1) spawns off `num_tasks` asynchronous tasks, each running the lambda function from #2. This lambda captures by copying, so each task will have its own values for `chunk_begin` and `chunk_end`, as well as a copy of the shared pointer `done_flag`. This avoids any concerns over lifetime issues.

Once all the tasks have been spawned, then we want to handle the case that a task returned. This is done by chaining a continuation on the `when_any` call #3. This time we write the continuation as a class, because we want to reuse it recursively. When one of the initial tasks is *ready*, the `DoneCheck` function call operator is invoked (#4). First, it extracts the value from the future that is *ready* (#5), and then if the value was found we process it, and set the final result (#6). Otherwise, we drop the *ready* future from the collection (#7), and if there are still more futures to check, issue a new call to `when_any` (#8) that will trigger its continuation when the next future is *ready*. If there are no futures left, then none of them found the value, so store an exception instead (#9). The return value of the function is the future for the final result (#10). There are of course alternative ways to solve this problem, but I hope this shows how one might use `when_any`.

Both these examples of using `when_all` and `when_any` have used the iterator-range overloads, that take a pair of iterators denoting the beginning and end of a set of futures to wait for. Both functions also come in variadic forms, where they accept a number of futures directly as parameters to the function. In this case, the result is a future holding a tuple (or a `when_any_result` holding a tuple) rather than a vector. e.g.

```
std::experimental::future<int> f1=spawn_async(func1);
std::experimental::future<std::string> f2=spawn_async(func2);
std::experimental::future<double> f3=spawn_async(func3);
std::experimental::future<
    std::tuple<
        std::experimental::future<int>,
        std::experimental::future<std::string>,
        std::experimental::future<double>>> result=
    std::experimental::when_all(std::move(f1),std::move(f2),std::move(f3));
```

This example actually highlights something important about all the uses of `when_any` and `when_all` — they always *move* from any `std::experimental::futures` passed in via a container, and they take their parameters by value, so **you** have to explicitly *move* the futures in, or pass temporaries.

Sometimes the event that you're waiting for is for a set of threads to reach a particular point in the code, or to have processed a certain number of data items between them. In these cases, you might be better served using a *latch* or a *barrier* rather than a *future*. Let's look at the latches and barriers that are provided by the Concurrency TS.

4.6 Latches and Barriers in the Concurrency TS

First off, let us consider what is meant when we talk of a *latch* or a *barrier*. A *latch* is a synchronization object which becomes *ready* when its counter is decremented to zero. Its name comes from the fact that it *latches* the output — once it is *ready*, it stays *ready* until it is destroyed. A latch is thus a light-weight facility for waiting for a series of events to occur.

On the other hand, a *barrier* is a reusable synchronization component used for internal synchronization between a set of threads. Whereas a latch doesn't care which threads decrement the counter — the same thread can decrement the counter multiple times, or

multiple threads can each decrement the counter once, or some combination of the two — with barriers, each thread can only *arrive* at the barrier once per cycle. When threads arrive at the barrier, they block until all of the threads involved have *arrived* at the barrier, at which point they are all released. The barrier can then be reused — the threads can then *arrive* at the barrier again to wait for all the threads for the next cycle.

Latches are inherently simpler than barriers, so let's start with the latch type from the Concurrency TS: `std::experimental::latch`.

4.6.1 A basic latch type: `std::experimental::latch`

`std::experimental::latch` comes from the `<experimental/latch>` header. When you construct a `std::experimental::latch`, you specify the initial counter value as the one and only argument to the constructor. Then, as the events that you are waiting for occur, you call `count_down` on your latch object, and the latch becomes *ready* when that count reaches zero. If you need to wait for the latch to become *ready*, then you can call `wait` on the latch; if you just need to check if it is ready then you can call `is_ready`. Finally, if you need to both count down the counter, and then wait for the counter to reach zero you can call `count_down_and_wait`. A basic example is shown in listing 4.25.

Listing 4.25 Waiting for events with `std::experimental::latch`

```
void foo(){
    unsigned const thread_count=...
    latch done(thread_count);                                #1
    my_data data[thread_count];
    std::vector<std::future<void>> threads;
    for(unsigned i=0;i<thread_count;++i)
        threads.push_back(std::async(std::launch::async,[&,i]{
            #2
            data[i]=make_data(i);
            done.count_down();                                #3
            do_more_stuff();                                #4
        }));
    done.wait();                                         #5
    process_data(data,thread_count);                      #6
}                                                       #7
```

This constructs `done` with the number of events that we need to wait for (#1), and then spawns the appropriate number of threads using `std::async` (#2). Each thread then counts down the latch when it has generated the relevant chunk of data (#3), before continuing on with further processing (#4). The main thread can wait for all the data to be ready by waiting on the latch (#5), before processing the generated data (#6). The data processing at #6 will potentially run concurrently with the final processing steps of each thread (#4) — there is no guarantee that the threads have all completed until the `std::future` destructors run at the end of the function (#7).

One thing to note is that the lambda passed to `std::async` at #2 captures everything by reference except `i`, which is captured by value. This is because `i` is the loop counter, and capturing that by reference would cause a data race and undefined behaviour, whereas `data` and `done` are things we really do need to share access to. Also, we only need a latch at all in

this scenario because the threads have additional processing to do after the data is ready, otherwise we could just wait for all the futures to ensure the tasks were complete before processing the data.

It is safe to access data in the `process_data` call (#6), even though it is stored by tasks running in other threads, because the latch is a synchronization object, so changes visible to a thread that `call count_down` are guaranteed to be visible to a thread that returns from a call to `wait` on the same latch object. Formally, the call to `count_down` *synchronizes-with* the call to `wait` — we'll see what that really means when we look at the low level memory ordering and synchronization constraints in chapter 5.

Alongside latches, the Concurrency TS gives us barriers — reusable synchronization objects for synchronizing a group of threads. Let us look at those next.

4.6.2 std::experimental::barrier: a basic barrier

The Concurrency TS provides two types of barrier in the `<experimental/barrier>` header: `std::experimental::barrier` and `std::experimental::flex_barrier`. The former is more basic, and potentially therefore has lower overhead, whereas the latter is more flexible, but potentially has more overhead.

Suppose you have a group of threads that are operating on some data. Each thread can do its processing on the data independently of the others, so no synchronization is needed during the processing, but all the threads must have completed their processing before the next data item can be processed, or before the subsequent processing can be done. `std::experimental::barrier` is targeted at precisely this scenario. You construct a barrier with a count specifying the number of threads involved in the synchronization group. As each thread is done with its processing, it *arrives* at the barrier, and waits for the rest of the group, by calling `arrive_and_wait` on the barrier object. When the last thread in the group arrives, all the threads are released, and the barrier reset. The threads in the group can then resume their processing, and either process the next data item, or proceed with the next stage of processing, as appropriate.

Whereas latches *latch*, so once they are *ready* they stay *ready*, barriers do not — barriers release the waiting threads, and then reset so they can be used again. They also only synchronize *within* a group of threads — a thread cannot wait for a barrier to be *ready* unless it is one of the threads in the synchronization group. Threads can explicitly drop out of the group by calling `arrive_and_drop` on the barrier, in which case that thread cannot wait for the barrier to be *ready* any more, and the count of threads that must *arrive* in the next cycle is one less.

Listing 4.26 Using a `std::experimental::barrier`

```
result_chunk process(data_chunk);
std::vector<data_chunk>
divide_into_chunks(data_block data, unsigned num_threads);

void process_data(data_source &source, data_sink &sink) {
    unsigned const concurrency = std::thread::hardware_concurrency();
```

```

unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

std::experimental::barrier sync(num_threads);
std::vector<joining_thread> threads(num_threads);

std::vector<data_chunk> chunks;
result_block result;

for (unsigned i = 0; i < num_threads; ++i) {
    threads[i] = joining_thread([&, i] {
        while (!source.done()) {                                #6
            if (!i) {                                         #1
                data_block current_block =
                    source.get_next_data_block();
                chunks = divide_into_chunks(
                    current_block, num_threads);
            }
            sync.arrive_and_wait();                            #2
            result.set_chunk(i, num_threads, process(chunks[i]));  #3
            sync.arrive_and_wait();                            #4
            if (!i) {                                         #5
                sink.write_data(std::move(result));
            }
        }
    });
}
}                                                               #7

```

Listing 4.26 shows an example of using a barrier to synchronize a group of threads. We have data coming from `source`, and we're writing the output to `sink`, but in order to make use of the available concurrency in the system, we're splitting each block of data into `num_threads` chunks. This has to be done serially, so we have an initial block (#1) that only runs on the thread for which `i==0`. All threads then wait on the barrier for that serial code to complete (#2), before we reach the parallel region, where each thread processes its individual chunk and updates the `result` with that (#3), before synchronizing again (#4). We then have a second serial region where only thread 0 writes the result out to the `sink` (#5). All threads then keep looping until the `source` reports that everything is done (#6). Note, that as each thread loops round, the serial section at the bottom of the loop combines with the section at the top; since only thread 0 has anything to do in either of these sections, this is OK, and all the threads will synchronize together at the first use of the barrier (#2). When all the processing is done, then all the threads will exit the loop, and the destructors for the `joining_thread` objects will wait for them all to finish at the end of the outer function (#7) (`joining_thread` was introduced in chapter 2, listing 2.7).

The key thing to note here is that the calls to `arrive_and_wait` are at the points in the code where it is important that no threads proceed until all threads are ready. At the first synchronization point, all the threads are really waiting for thread 0 to arrive, but the use of the barrier provides us with a clean line in the sand. At the second synchronization point, we have the reverse situation: really it is thread 0 that is waiting for all the other threads to arrive before it can write out the completed `result` to the `sink`.

The Concurrency TS doesn't just give us one barrier type; as well as `std::experimental::barrier`, we also get `std::experimental::flex_barrier`, which is more flexible. One of the ways that it is more flexible is that it allows for a final serial region to be run when all threads have *arrived* at the barrier, before they are all released again.

4.6.3 `std::experimental::flex_barrier` – `std::experimental::barrier's flexible friend`

The interface to `std::experimental::flex_barrier` differs from that of `std::experimental::barrier` in only one way: there is an additional constructor that takes a completion function, as well as thread count. This function is run on exactly one of the threads that *arrived* at the barrier, once all the threads have *arrived* at the barrier. Not only does it provide a means of specifying a chunk of code that must be run serially, it also provides a means of changing the number of threads that must *arrive* at the barrier for the next cycle. The thread count can be changed to any number, whether higher or lower than the previous count; it is up to the programmer who uses this facility to ensure that the correct number of threads will *arrive* at the barrier the next time round.

Listing 4.27 shows how listing 4.26 could be rewritten to use `std::experimental::flex_barrier` to manage the serial region.

Listing 4.27 Using `std::flex_barrier` to provide a serial region

```
void process_data(data_source &source, data_sink &sink) {
    unsigned const concurrency = std::thread::hardware_concurrency();
    unsigned const num_threads = (concurrency > 0) ? concurrency : 2;

    std::vector<data_chunk> chunks;

    auto split_source = [&] {                                     #1
        if (!source.done()) {
            data_block current_block = source.get_next_data_block();
            chunks = divide_into_chunks(current_block, num_threads);
        }
    };

    split_source();                                         #2

    result_block result;

    std::experimental::flex_barrier sync(num_threads, [&] {      #3
        sink.write_data(std::move(result));
        split_source();                                         #4
        return -1;                                              #5
    });
    std::vector<joining_thread> threads(num_threads);

    for (unsigned i = 0; i < num_threads; ++i) {
        threads[i] = joining_thread([&, i] {
            while (!source.done()) {                            #6
                result.set_chunk(i, num_threads, process(chunks[i]));
                sync.arrive_and_wait();                         #7
            }
        });
    }
}
```

```
        }
    });
}
```

The first difference between this code and listing 4.26 is that we've extracted a lambda that splits the next data block into chunks (#1). This is called before we start (#2), and encapsulates the code that was run on thread 0 at the start of each iteration.

The second difference is that our sync object is now a `std::experimental::flex_barrier`, and we are passing a completion function as well as a thread count (#3). This completion function is run on one thread after each thread has arrived, and so can encapsulate the code that was to be run on thread 0 at the end of each iteration, and then a call to our newly-extracted `split_source` lambda that would have been called at the start of the next iteration (#4). The return value of -1 (#5) indicates that the number of participating threads is to remain unchanged; a return value of zero or more would specify the number of participating threads in the next cycle.

The main loop (#6) is now simplified: it only contains the parallel portion of the code, and thus only needs a single synchronization point (#7). The use of `std::experimental::flex_barrier` has thus simplified the code.

The use of the completion function to provide a serial section is quite powerful, as is the ability to change the number of participating threads. For example, this could be used by pipeline style code where the number of threads is less during the initial priming of the pipeline and the final draining of the pipeline than it is during the main processing when all the stages of the pipeline are operating.

4.7 Summary

Synchronizing operations between threads is an important part of writing an application that uses concurrency: if there's no synchronization, the threads are essentially independent and might as well be written as separate applications that are run as a group because of their related activities. In this chapter, I've covered various ways of synchronizing operations from the basic condition variables, through futures, promises, packaged tasks, latches and barriers. I've also discussed ways of approaching the synchronization issues: functional-style programming where each task produces a result entirely dependent on its input rather than on the external environment, message passing where communication between threads is via asynchronous messages sent through a messaging subsystem that acts as an intermediary, and continuation style, where the follow-on tasks for each operation are specified, and the system takes care of the scheduling.

Having discussed many of the high-level facilities available in C++, it's now time to look at the low-level facilities that make it all work: the C++ memory model and atomic operations.

5

The C++ memory model and operations on atomic types

This chapter covers

- The details of the C++ memory model
- The atomic types provided by the C++ Standard Library
- The operations that are available on those types
- How those operations can be used to provide synchronization between threads

One of the most important features of the C++ Standard is something most programmers won't even notice. It's not the new syntax features, nor is it the new library facilities, but the new multithreading-aware memory model. Without the memory model to define exactly how the fundamental building blocks work, none of the facilities I've covered could be relied on to work. Of course, there's a reason that most programmers won't notice: if you use mutexes to protect your data and condition variables, futures, latches or barriers to signal events, the details of *why* they work aren't important. It's only when you start trying to get "close to the machine" that the precise details of the memory model matter.

Whatever else it is, C++ is a systems programming language. One of the goals of the Standards Committee is that there shall be no need for a lower-level language than C++. Programmers should be provided with enough flexibility within C++ to do whatever they need without the language getting in the way, allowing them to get "close to the machine" when the need arises. The atomic types and operations allow just that, providing facilities for low-level synchronization operations that will commonly reduce to one or two CPU instructions.

In this chapter, I'll start by covering the basics of the memory model, then move on to the atomic types and operations, and finally cover the various types of synchronization available with the operations on atomic types. This is quite complex: unless you're planning on writing code that uses the atomic operations for synchronization (such as the lock-free data structures in chapter 7), you won't need to know these details.

Let's ease into things with a look at the basics of the memory model.

5.1 Memory model basics

There are two aspects to the memory model: the basic *structural* aspects, which relate to how things are laid out in memory, and then the *concurrency* aspects. The structural aspects are important for concurrency, especially when you're looking at low-level atomic operations, so I'll start with those. In C++, it's all about objects and memory locations.

5.1.1 Objects and memory locations

All data in a C++ program is made up of *objects*. This is not to say that you can create a new class derived from `int`, or that the fundamental types have member functions, or any of the other consequences often implied when people say "everything is an object" when discussing a language like Smalltalk or Ruby. It's just a statement about the building blocks of data in C++. The C++ Standard defines an object as "a region of storage," although it goes on to assign properties to these objects, such as their type and lifetime.

Some of these objects are simple values of a fundamental type such as `int` or `float`, whereas others are instances of user-defined classes. Some objects (such as arrays, instances of derived classes, and instances of classes with non-static data members) have subobjects, but others don't.

Whatever its type, an object is stored in one or more *memory locations*. Each such memory location is either an object (or subobject) of a scalar type such as `unsigned short` or `my_class*` or a sequence of adjacent bit fields. If you use bit fields, this is an important point to note: though adjacent bit fields are distinct objects, they're still counted as the same memory location. Figure 5.1 shows how a `struct` divides into objects and memory locations.

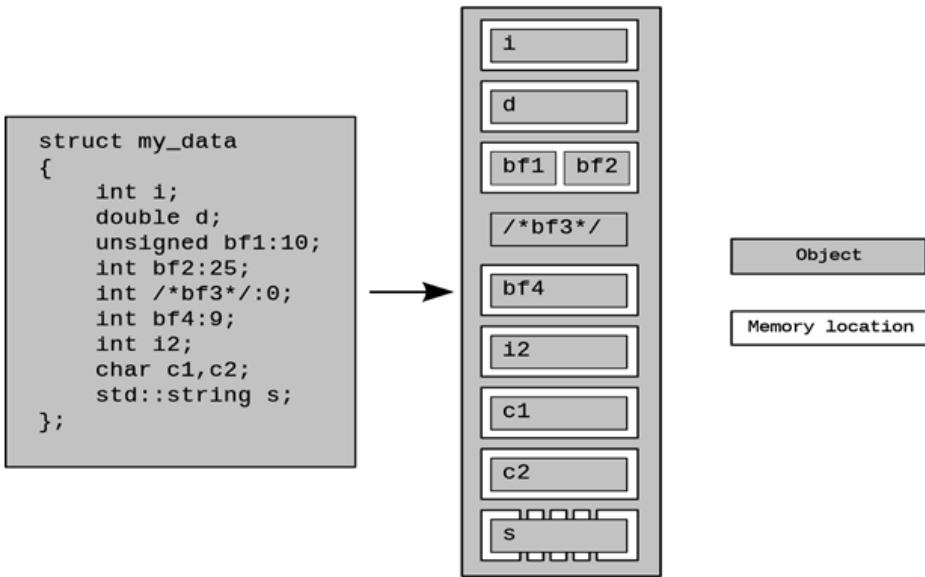


Figure 5.1 The division of a struct into objects and memory locations

First, the entire `struct` is one object, which consists of several subobjects, one for each data member. The bit fields `bf1` and `bf2` share a memory location, and the `std::string` object `s` consists of several memory locations internally, but otherwise each member has its own memory location. Note how the zero-length bit field marked `/*bf3*/` (the name is commented out because zero-length bitfields must be unnamed) separates `bf4` into its own memory location, but doesn't have a memory location itself..

There are four important things to take away from this:

- Every variable is an object, including those that are members of other objects.
- Every object occupies *at least one* memory location.
- Variables of fundamental type such as `int` or `char` are *exactly one* memory location, whatever their size, even if they're adjacent or part of an array.
- Adjacent bit fields are part of the same memory location.

I'm sure you're wondering what this has to do with concurrency, so let's take a look.

5.1.2 Objects, memory locations, and concurrency

Now, here's the part that's crucial for multithreaded applications in C++: everything hinges on those memory locations. If two threads access *separate* memory locations, there's no problem: everything works fine. On the other hand, if two threads access the *same* memory location, then you have to be careful. If neither thread is updating the memory location, you're

fine; read-only data doesn't need protection or synchronization. If either thread is modifying the data, there's a potential for a race condition, as described in chapter 3.

In order to avoid the race condition, there has to be an enforced ordering between the accesses in the two threads. This could be a fixed ordering such that one access is always before the other, or it could be an ordering that varies between runs of the application, but guarantees that there is *some* defined ordering. One way to ensure there's a defined ordering is to use mutexes as described in chapter 3; if the same mutex is locked prior to both accesses, only one thread can access the memory location at a time, so one must happen before the other (though you can't in general know in advance which will be first). The other way is to use the synchronization properties of *atomic* operations (see section 5.2 for the definition of atomic operations) either on the same or other memory locations to enforce an ordering between the accesses in the two threads. The use of atomic operations to enforce an ordering is described in section 5.3. If more than two threads access the same memory location, each pair of accesses must have a defined ordering.

If there's no enforced ordering between two accesses to a single memory location from separate threads, one or both of those accesses is not atomic, and one or both is a write, then this is a data race and causes undefined behavior.

This statement is crucially important: undefined behavior is one of the nastiest corners of C++. According to the language standard, once an application contains any undefined behavior, all bets are off; the behavior of the complete application is now undefined, and it may do anything at all. I know of one case where a particular instance of undefined behavior caused someone's monitor to catch on fire. Although this is rather unlikely to happen to you, a data race is definitely a serious bug and should be avoided at all costs.

There's another important point in that statement: you can also avoid the undefined behavior by using atomic operations to access the memory location involved in the race. This doesn't prevent the race itself—which of the atomic operations touches the memory location first is still not specified—but it does bring the program back into the realm of defined behavior.

Before we look at atomic operations, there's one more concept that's important to understand about objects and memory locations: modification orders.

5.1.3 Modification orders

Every object in a C++ program has a *modification order* composed of all the writes to that object from all threads in the program, starting with the object's initialization. In most cases this order will vary between runs, but in any given execution of the program all threads in the system must agree on the order. If the object in question isn't one of the atomic types described in section 5.2, you're responsible for making certain that there's sufficient synchronization to ensure that threads agree on the modification order of each variable. If different threads see distinct sequences of values for a single variable, you have a data race and undefined behavior (see section 5.1.2). If you do use atomic operations, the compiler is responsible for ensuring that the necessary synchronization is in place.

This requirement means that certain kinds of speculative execution aren't permitted, because once a thread has seen a particular entry in the modification order, subsequent reads from that thread must return later values, and subsequent writes from that thread to that object must occur later in the modification order. Also, a read of an object that follows a write to that object in the same thread must either return the value written or another value that occurs later in the modification order of that object. Although all threads must agree on the modification orders of each individual object in a program, they don't necessarily have to agree on the relative order of operations on separate objects. See section 5.3.3 for more on the ordering of operations between threads.

So, what constitutes an atomic operation, and how can these be used to enforce ordering?

5.2 Atomic operations and types in C++

An *atomic operation* is an indivisible operation. You can't observe such an operation half-done from any thread in the system; it's either done or not done. If the load operation that reads the value of an object is *atomic*, and all modifications to that object are also *atomic*, that load will retrieve either the initial value of the object or the value stored by one of the modifications.

The flip side of this is that a non-atomic operation might be seen as half-done by another thread. If the non-atomic operation is composed of atomic operations (e.g. assignment to a struct with atomic members), then other threads may observe some subset of the constituent atomic operations as complete, but others as not yet started, so you might observe or end up with a value which is a mixed-up combination of the various values stored. In any case, unsynchronized accesses to non-atomic variables form a simple problematic race condition, as described in chapter 3, but at this level it may constitute a *data race* (see section 5.1) and thus cause undefined behavior.

In C++, you need to use an atomic type to get an atomic operation in most cases, so let's look at those.

5.2.1 The standard atomic types

The standard *atomic* types can be found in the `<atomic>` header. All operations on such types are atomic, and only operations on these types are atomic in the sense of the language definition, although you can use mutexes to make other operations *appear* atomic. In actual fact, the standard atomic types themselves might use such emulation: they (almost — see below) all have an `is_lock_free()` member function, which allows the user to determine whether operations on a given type are done directly with atomic instructions (`x.is_lock_free() returns true`) or done by using a lock internal to the compiler and library (`x.is_lock_free() returns false`).

This is important to know in many cases — the key use case for atomic operations is as a replacement for an operation that would otherwise use a mutex for synchronization; if the atomic operations themselves use an internal mutex then the hoped-for performance gains will probably not materialize, and you might be better off using the easier-to-get-right mutex-

based implementation instead. This is especially the case with lock-free data structures such as those discussed in chapter 7.

In fact, this is so important that the library provides a set of macros to identify at compile time whether the atomic types for the various integral types are lock-free, and since C++17 all atomic types have a static `constexpr` member variable `x::is_always_lock_free`, which is true if and only if the atomic type `x` is lock-free for all supported hardware that the output of the current compilation might run on. For example, for a given target platform, `std::atomic<int>` might always be lock-free, so `std::atomic<int>::is_always_lock_free` will be true, but `std::atomic<uintmax_t>` might only be lock-free if the hardware the program ends up running on supports the necessary instructions, so this is thus a run-time property, and `std::atomic<uintmax_t>::is_always_lock_free` would be false when compiling for that platform.

The macros are `ATOMIC_BOOL_LOCK_FREE`, `ATOMIC_CHAR_LOCK_FREE`, `ATOMIC_CHAR16_T_LOCK_FREE`, `ATOMIC_CHAR32_T_LOCK_FREE`, `ATOMIC_WCHAR_T_LOCK_FREE`, `ATOMIC_SHORT_LOCK_FREE`, `ATOMIC_INT_LOCK_FREE`, `ATOMIC_LONG_LOCK_FREE`, `ATOMIC_LLONG_LOCK_FREE`, and `ATOMIC_POINTER_LOCK_FREE`. They specify the lock-free-ness of the corresponding atomic types for the specified built-in types and their unsigned counterparts (`LLONG` refers to `long long`, and `POINTER` refers to all pointer types). They evaluate to the value 0 if the atomic type is *never* lock-free, to the value 2 if the atomic type is *always* lock-free, and to the value 1 if the lock-free-ness of the corresponding atomic type is a runtime property as described above.

The only type that doesn't provide an `is_lock_free()` member function is `std::atomic_flag`. This type is a really simple Boolean flag, and operations on this type are *required* to be lock-free; once you have a simple lock-free Boolean flag, you can use that to implement a simple lock and thus implement all the other atomic types using that as a basis. When I said *really simple*, I meant it: objects of type `std::atomic_flag` are initialized to clear, and they can then either be queried and set (with the `test_and_set()` member function) or cleared (with the `clear()` member function). That's it: no assignment, no copy construction, no test and clear, no other operations at all.

The remaining atomic types are all accessed through specializations of the `std::atomic<>` class template and are a bit more full-featured but may not be lock-free (as explained previously). On most popular platforms it's expected that the atomic variants of all the built-in types (such as `std::atomic<int>` and `std::atomic<void*>`) are indeed lock-free, but it isn't required. As you'll see shortly, the interface of each specialization reflects the properties of the type; bitwise operations such as `&=` aren't defined for plain pointers, so they aren't defined for atomic pointers either, for example.

In addition to using the `std::atomic<>` class template directly, you can use the set of names shown in table 5.1 to refer to the implementation-supplied atomic types. Because of the history of how atomic types were added to the C++ Standard, if you have an older compiler, these alternative type names may refer either to the corresponding `std::atomic<>` specialization or to a base class of that specialization, whereas in a compiler that fully supports

C++17, these are always aliases for the corresponding `std::atomic<>` specializations. Mixing these alternative names with direct naming of `std::atomic<>` specializations in the same program can therefore lead to nonportable code.

Table 5.1 The alternative names for the standard atomic types and their corresponding `std::atomic<>` specializations

Atomic type	Corresponding specialization
<code>atomic_bool</code>	<code>std::atomic<bool></code>
<code>atomic_char</code>	<code>std::atomic<char></code>
<code>atomic_schar</code>	<code>std::atomic<signed char></code>
<code>atomic_uchar</code>	<code>std::atomic<unsigned char></code>
<code>atomic_int</code>	<code>std::atomic<int></code>
<code>atomic_uint</code>	<code>std::atomic<unsigned></code>
<code>atomic_short</code>	<code>std::atomic<short></code>
<code>atomic_ushort</code>	<code>std::atomic<unsigned short></code>
<code>atomic_long</code>	<code>std::atomic<long></code>
<code>atomic_ulong</code>	<code>std::atomic<unsigned long></code>
<code>atomic_llong</code>	<code>std::atomic<long long></code>
<code>atomic_ullong</code>	<code>std::atomic<unsigned long long></code>

<code>atomic_char16_t</code>	<code>std::atomic<char16_t></code>
<code>atomic_char32_t</code>	<code>std::atomic<char32_t></code>
<code>atomic_wchar_t</code>	<code>std::atomic<wchar_t></code>

As well as the basic atomic types, the C++ Standard Library also provides a set of `typedefs` for the atomic types corresponding to the various nonatomic Standard Library `typedefs` such as `std::size_t`. These are shown in table 5.2.

Table 5.2 The standard atomic `typedefs` and their corresponding built-in `typedefs`

Atomic <code>typedef</code>	Corresponding Standard Library <code>typedef</code>
<code>atomic_int_least8_t</code>	<code>int_least8_t</code>
<code>atomic_uint_least8_t</code>	<code>uint_least8_t</code>
<code>atomic_int_least16_t</code>	<code>int_least16_t</code>
<code>atomic_uint_least16_t</code>	<code>uint_least16_t</code>
<code>atomic_int_least32_t</code>	<code>int_least32_t</code>
<code>atomic_uint_least32_t</code>	<code>uint_least32_t</code>
<code>atomic_int_least64_t</code>	<code>int_least64_t</code>
<code>atomic_uint_least64_t</code>	<code>uint_least64_t</code>
<code>atomic_int_fast8_t</code>	<code>int_fast8_t</code>
<code>atomic_uint_fast8_t</code>	<code>uint_fast8_t</code>
<code>atomic_int_fast16_t</code>	<code>int_fast16_t</code>

atomic_uint_fast16_t	uint_fast16_t
atomic_int_fast32_t	int_fast32_t
atomic_uint_fast32_t	uint_fast32_t
atomic_int_fast64_t	int_fast64_t
atomic_uint_fast64_t	uint_fast64_t
atomic_intptr_t	intptr_t
atomic_uintptr_t	uintptr_t
atomic_size_t	size_t
atomic_ptrdiff_t	ptrdiff_t
atomic_intmax_t	intmax_t
atomic_uintmax_t	uintmax_t

That's a lot of types! There's a rather simple pattern to it; for a standard `typedef T`, the corresponding atomic type is the same name with an `atomic_` prefix: `atomic_T`. The same applies to the built-in types, except that `signed` is abbreviated as just `s`, `unsigned` as just `u`, and `long long` as `llong`. It's generally just simpler to say `std::atomic<T>` for whichever `T` you wish to work with, rather than use the alternative names.

The standard atomic types are not copyable or assignable in the conventional sense, in that they have no copy constructors or copy assignment operators. They *do*, however, support assignment from and implicit conversion to the corresponding built-in types as well as direct `load()` and `store()` member functions, `exchange()`, `compare_exchange_weak()`, and `compare_exchange_strong()`. They also support the compound assignment operators where appropriate: `+=`, `-=`, `*=`, `|=`, and so on, and the integral types and `std::atomic<>` specializations for pointers support `++` and `--`. These operators also have corresponding named member functions with the same functionality: `fetch_add()`, `fetch_or()`, and so on. The return value from the assignment operators and member functions is either the value stored (in the case of the assignment operators) or the value prior to the operation (in the case of the named functions). This avoids the potential problems that could stem from the usual habit of such assignment operators returning a reference to the object being assigned to. In order to get the stored value from such a reference, the code would have to perform a

separate read, thus allowing another thread to modify the value between the assignment and the read and opening the door for a race condition.

The `std::atomic<T>` class template isn't just a set of specializations, though. It does have a primary template that can be used to create an atomic variant of a user-defined type. Because it's a generic class template, the operations are limited to `load()`, `store()` (and assignment from and conversion to the user-defined type), `exchange()`, `compare_exchange_weak()`, and `compare_exchange_strong()`.

Each of the operations on the atomic types has an optional memory-ordering argument which is one of the values of the `std::memory_order` enumeration. This argument is used to specify the required memory-ordering semantics. The `std::memory_order` enumeration has 6 possible values: `std::memory_order_relaxed`, `std::memory_order_acquire`, `std::memory_order_consume`, `std::memory_order_acq_rel`, `std::memory_order_release`, and `std::memory_order_seq_cst`.

The permitted values for the memory ordering depend on the operation category. If you don't specify an ordering value, then the default ordering is used, which is the strongest ordering: `std::memory_order_seq_cst`. The precise semantics of the memory-ordering options are covered in section 5.3. For now, it suffices to know that the operations are divided into three categories:

- *Store* operations, which can have `memory_order_relaxed`, `memory_order_release`, or `memory_order_seq_cst` ordering
- *Load* operations, which can have `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, or `memory_order_seq_cst` ordering
- *Read-modify-write* operations, which can have `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, or `memory_order_seq_cst` ordering

Let's now look at the operations you can actually do on each of the standard atomic types, starting with `std::atomic_flag`.

5.2.2 Operations on `std::atomic_flag`

`std::atomic_flag` is the simplest standard atomic type, which represents a Boolean flag. Objects of this type can be in one of two states: set or clear. It's deliberately basic and is intended as a building block only. As such, I'd never expect to see it in use, except under very special circumstances. Even so, it will serve as a starting point for discussing the other atomic types, because it shows some of the general policies that apply to the atomic types.

Objects of type `std::atomic_flag` *must* be initialized with `ATOMIC_FLAG_INIT`. This initializes the flag to a *clear* state. There's no choice in the matter; the flag always starts clear:

```
std::atomic_flag f=ATOMIC_FLAG_INIT;
```

This applies wherever the object is declared and whatever scope it has. It's the only atomic type to require such special treatment for initialization, but it's also the only type guaranteed to be lock-free. If the `std::atomic_flag` object has static storage duration, it's guaranteed to

be statically initialized, which means that there are no initialization-order issues; it will always be initialized by the time of the first operation on the flag.

Once you have your flag object initialized, there are only three things you can do with it: destroy it, clear it, or set it and query the previous value. These correspond to the destructor, the `clear()` member function, and the `test_and_set()` member function, respectively. Both the `clear()` and `test_and_set()` member functions can have a memory order specified. `clear()` is a `store` operation and so can't have `memory_order_acquire` or `memory_order_acq_rel` semantics, but `test_and_set()` is a read-modify-write operation and so can have any of the memory-ordering tags applied. As with every atomic operation, the default for both is `memory_order_seq_cst`. For example:

```
f.clear(std::memory_order_release);      #1
bool x=f.test_and_set();                 #2
```

Here, the call to `clear()` #1 explicitly requests that the flag is cleared with release semantics, while the call to `test_and_set()` #2 uses the default memory ordering for setting the flag and retrieving the old value.

You can't copy-construct another `std::atomic_flag` object from the first, and you can't assign one `std::atomic_flag` to another. This isn't something peculiar to `std::atomic_flag` but something common with all the atomic types. All operations on an atomic type are defined as atomic, and assignment and copy-construction involve two objects. A single operation on two distinct objects can't be atomic. In the case of copy-construction or copy-assignment, the value must first be read from one object and then written to the other. These are two separate operations on two separate objects, and the combination can't be atomic. Therefore, these operations aren't permitted.

The limited feature set makes `std::atomic_flag` ideally suited to use as a spin-lock mutex. Initially the flag is clear and the mutex is unlocked. To lock the mutex, loop on `test_and_set()` until the old value is `false`, indicating that *this* thread set the value to `true`. Unlocking the mutex is simply a matter of clearing the flag. Such an implementation is shown in the following listing.

Listing 5.1 Implementation of a spinlock mutex using `std::atomic_flag`

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
}
```

```
};
```

Such a mutex is very basic, but it's enough to use with `std::lock_guard<>` (see chapter 3). By its very nature it does a busy-wait in `lock()`, so it's a poor choice if you expect there to be any degree of contention, but it's enough to ensure mutual exclusion. When we look at the memory-ordering semantics, you'll see how this guarantees the necessary enforced ordering that goes with a mutex lock. This example is covered in section 5.3.6.

`std::atomic_flag` is so limited that it can't even be used as a general Boolean flag, because it doesn't have a simple nonmodifying query operation. For that you're better off using `std::atomic<bool>`, so I'll cover that next.

5.2.3 Operations on `std::atomic<bool>`

The most basic of the atomic integral types is `std::atomic<bool>`. This is a more full-featured Boolean flag than `std::atomic_flag`, as you might expect. Although it's still not copy-constructible or copy-assignable, you can construct it from a nonatomic `bool`, so it can be initially `true` or `false`, and you can also assign to instances of `std::atomic<bool>` from a nonatomic `bool`:

```
std::atomic<bool> b(true);
b=false;
```

One other thing to note about the assignment operator from a nonatomic `bool` is that it differs from the general convention of returning a reference to the object it's assigned to: it returns a `bool` with the value assigned instead. This is another common pattern with the atomic types: the assignment operators they support return values (of the corresponding nonatomic type) rather than references. If a reference to the atomic variable was returned, any code that depended on the result of the assignment would then have to explicitly load the value, potentially getting the result of a modification by another thread. By returning the result of the assignment as a nonatomic value, you can avoid this additional load, and you know that the value obtained is the actual value stored.

Rather than using the restrictive `clear()` function of `std::atomic_flag`, writes (of either `true` or `false`) are done by calling `store()`, although the memory-order semantics can still be specified. Similarly, `test_and_set()` has been replaced with the more general `exchange()` member function that allows you to replace the stored value with a new one of your choosing and atomically retrieve the original value. `std::atomic<bool>` also supports a plain nonmodifying query of the value with an implicit conversion to plain `bool` or with an explicit call to `load()`. As you might expect, `store()` is a store operation, whereas `load()` is a load operation. `exchange()` is a read-modify-write operation:

```
std::atomic<bool> b;
bool x=b.load(std::memory_order_acquire);
b.store(true);
x=b.exchange(false,std::memory_order_acq_rel);
```

`exchange()` isn't the only read-modify-write operation supported by `std::atomic <bool>`; it also introduces an operation to store a new value if the current value is equal to an expected value.

STORING A NEW VALUE (OR NOT) DEPENDING ON THE CURRENT VALUE

This new operation is called compare-exchange, and it comes in the form of the `compare_exchange_weak()` and `compare_exchange_strong()` member functions. The compare-exchange operation is the cornerstone of programming with atomic types; it compares the value of the atomic variable with a supplied expected value and stores the supplied desired value if they're equal. If the values aren't equal, the expected value is updated with the actual value of the atomic variable. The return type of the compare-exchange functions is a `bool`, which is `true` if the store was performed and `false` otherwise. The operation is said to *succeed* if the store was done (because the values were equal), and *fail* otherwise; the return value is thus `true` for *success*, and `false` for *failure*.

For `compare_exchange_weak()`, the store might not be successful even if the original value was equal to the expected value, in which case the value of the variable is unchanged and the return value of `compare_exchange_weak()` is `false`. This is most likely to happen on machines that lack a single compare-and-exchange instruction, if the processor can't guarantee that the operation has been done atomically—possibly because the thread performing the operation was switched out in the middle of the necessary sequence of instructions and another thread scheduled in its place by the operating system where there are more threads than processors. This is called a *spurious failure*, because the reason for the failure is a function of timing rather than the values of the variables.

Because `compare_exchange_weak()` can fail spuriously, it must typically be used in a loop:

```
bool expected=false;
extern atomic<bool> b; // set somewhere else
while(!b.compare_exchange_weak(expected,true) && !expected);
```

In this case, you keep looping as long as `expected` is still `false`, indicating that the `compare_exchange_weak()` call failed spuriously.

On the other hand, `compare_exchange_strong()` is guaranteed to return `false` only if the actual value wasn't equal to the expected value. This can eliminate the need for loops like the one shown where you just want to know whether you successfully changed a variable or whether another thread got there first.

If you want to change the variable whatever the initial value is (perhaps with an updated value that depends on the current value), the update of `expected` becomes useful; each time through the loop, `expected` is reloaded, so if no other thread modifies the value in the meantime, the `compare_exchange_weak()` or `compare_exchange_strong()` call should be successful the next time around the loop. If the calculation of the value to be stored is simple, it may be beneficial to use `compare_exchange_weak()` in order to avoid a double loop on platforms where `compare_exchange_weak()` can fail spuriously (and so `compare_exchange_strong()` contains a loop). On the other hand, if the calculation of the

value to be stored is itself time consuming, it may make sense to use `compare_exchange_strong()` to avoid having to recalculate the value to store when the expected value hasn't changed. For `std::atomic<bool>` this isn't so important—there are only two possible values after all—but for the larger atomic types this can make a difference.

The compare-exchange functions are also unusual in that they can take *two* memory-ordering parameters. This allows for the memory-ordering semantics to differ in the case of success and failure; it might be desirable for a successful call to have `memory_order_acq_rel` semantics whereas a failed call has `memory_order_relaxed` semantics. A failed compare-exchange doesn't do a store, so it can't have `memory_order_release` or `memory_order_acq_rel` semantics. It's therefore not permitted to supply these values as the ordering for failure. You also can't supply stricter memory ordering for failure than for success; if you want `memory_order_acquire` or `memory_order_seq_cst` semantics for failure, you must specify those for success as well.

If you don't specify an ordering for failure, it's assumed to be the same as that for success, except that the `release` part of the ordering is stripped: `memory_order_release` becomes `memory_order_relaxed`, and `memory_order_acq_rel` becomes `memory_order_acquire`. If you specify neither, they default to `memory_order_seq_cst` as usual, which provides the full sequential ordering for both success and failure. The following two calls to `compare_exchange_weak()` are equivalent:

```
std::atomic<bool> b;
bool expected;
b.compare_exchange_weak(expected,true,
    memory_order_acq_rel,memory_order_acquire);
b.compare_exchange_weak(expected,true,memory_order_acq_rel);
```

I'll leave the consequences of the choice of memory ordering to section 5.3.

One further difference between `std::atomic<bool>` and `std::atomic_flag` is that `std::atomic<bool>` may not be lock-free; the implementation may have to acquire a mutex internally in order to ensure the atomicity of the operations. For the rare case when this matters, you can use the `is_lock_free()` member function to check whether operations on `std::atomic<bool>` are lock-free. This is another feature common to all atomic types other than `std::atomic_flag`.

The next-simplest of the atomic types are the atomic pointer specializations `std::atomic<T*>`, so we'll look at those next.

5.2.4 Operations on `std::atomic<T*>`: pointer arithmetic

The atomic form of a pointer to some type `T` is `std::atomic<T*>`, just as the atomic form of `bool` is `std::atomic<bool>`. The interface is essentially the same, although it operates on values of the corresponding pointer type rather than `bool` values. Just like `std::atomic<bool>`, it's neither copy-constructible nor copy-assignable, although it can be both constructed and assigned from the suitable pointer values. As well as the obligatory `is_lock_free()` member function, `std::atomic<T*>` also has `load()`, `store()`, `exchange()`,

`compare_exchange_weak()`, and `compare_exchange_strong()` member functions, with similar semantics to those of `std::atomic<bool>`, again taking and returning `T*` rather than `bool`.

The new operations provided by `std::atomic<T*>` are the pointer arithmetic operations. The basic operations are provided by the `fetch_add()` and `fetch_sub()` member functions, which do atomic addition and subtraction on the stored address, and the operators `+=` and `-=`, and both pre- and post-increment and decrement with `++` and `--`, which provide convenient wrappers. The operators work just as you'd expect from the built-in types: if `x` is `std::atomic<Foo*>` to the first entry of an array of `Foo` objects, then `x+=3` changes it to point to the fourth entry and returns a plain `Foo*` that also points to that fourth entry. `fetch_add()` and `fetch_sub()` are slightly different in that they return the original value (so `x.fetch_add(3)` will update `x` to point to the fourth value but return a pointer to the first value in the array). This operation is also known as *exchange-and-add*, and it's an atomic read-modify-write operation, like `exchange()` and `compare_exchange_weak()`/`compare_exchange_strong()`. Just as with the other operations, the return value is a plain `T*` value rather than a reference to the `std::atomic<T*>` object, so that the calling code can perform actions based on what the previous value was:

```
class Foo{};  
Foo some_array[5];  
std::atomic<Foo*> p(some_array);  
Foo* x=p.fetch_add(2);           #A  
assert(x==some_array);  
assert(p.load()==&some_array[2]);  
x=(p-=1);                      #B  
assert(x==&some_array[1]);  
assert(p.load()==&some_array[1]);  
#A Add 2 to p and return old value  
#B Subtract 1 from p and return new value
```

The function forms also allow the memory-ordering semantics to be specified as an additional function call argument:

```
p.fetch_add(3,std::memory_order_release);
```

Because both `fetch_add()` and `fetch_sub()` are read-modify-write operations, they can have any of the memory-ordering tags and can participate in a *release sequence*. Specifying the ordering semantics isn't possible for the operator forms, because there's no way of providing the information: these forms therefore always have `memory_order_seq_cst` semantics.

The remaining basic atomic types are essentially all the same: they're all atomic integral types and have the same interface as each other, except that the associated built-in type is different. We'll look at them as a group.

5.2.5 Operations on standard atomic integral types

As well as the usual set of operations (`load()`, `store()`, `exchange()`, `compare_exchange_weak()`, and `compare_exchange_strong()`), the atomic integral types such as

`std::atomic<int>` or `std::atomic<unsigned long long>` have quite a comprehensive set of operations available: `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, `fetch_xor()`, compound-assignment forms of these operations (`+=`, `-=`, `&=`, `|=`, and `^=`), and pre- and post-increment and decrement (`++x`, `x++`, `--x`, and `x--`). It's not quite the full set of compound-assignment operations you could do on a normal integral type, but it's close enough: only division, multiplication, and shift operators are missing. Because atomic integral values are typically used either as counters or as bitmasks, this isn't a particularly noticeable loss; additional operations can easily be done using `compare_exchange_weak()` in a loop, if required.

The semantics match closely to those of `fetch_add()` and `fetch_sub()` for `std::atomic<T*>`; the named functions atomically perform their operation and return the *old* value, whereas the compound-assignment operators return the *new* value. Pre- and post-increment and decrement work as usual: `++x` increments the variable and returns the new value, whereas `x++` increments the variable and returns the old value. As you'll be expecting by now, the result is a value of the associated integral type in both cases.

We've now looked at all the basic atomic types; all that remains is the generic `std::atomic<>` primary class template rather than the specializations, so let's look at that next.

5.2.6 The `std::atomic<>` primary class template

The presence of the primary template allows a user to create an atomic variant of a user-defined type, in addition to the standard atomic types. Given a user-defined type `UDT`, `std::atomic<UDT>` provides the same interface as `std::atomic<bool>` (as described in section 5.2.3), except that the `bool` parameters and return types that relate to the stored value (rather than the success/failure result of the compare-exchange operations) are `UDT` instead. You can't use just any user-defined type with `std::atomic<>`, though; the type has to fulfill certain criteria. In order to use `std::atomic<UDT>` for some user-defined type `UDT`, this type must have a *trivial* copy-assignment operator. This means that the type must not have any virtual functions or virtual base classes and must use the compiler-generated copy-assignment operator. Not only that, but every base class and non-static data member of a user-defined type must also have a trivial copy-assignment operator. This essentially permits the compiler to use `memcpy()` or an equivalent operation for assignment operations, because there's no user-written code to run.

Finally, it is worth noting that the compare-exchange operations do bitwise comparison as if using `memcmp`, rather than using any comparison operator that may be defined for `UDT`. If the type provides comparison operations that have different semantics, or the type has padding bits that do not participate in normal comparisons, then this can lead to a compare-exchange operation failing, even though the values compare equal.

The reasoning behind these restrictions goes back to one of the guidelines from chapter 3: don't pass pointers and references to protected data outside the scope of the lock by passing them as arguments to user-supplied functions. In general, the compiler isn't going to be able

to generate lock-free code for `std::atomic<UDT>`, so it will have to use an internal lock for all the operations. If user-supplied copy-assignment or comparison operators were permitted, this would require passing a reference to the protected data as an argument to a user-supplied function, thus violating the guideline. Also, the library is entirely at liberty to use a single lock for all atomic operations that need it, and allowing user-supplied functions to be called while holding that lock might cause deadlock or cause other threads to block because a comparison operation took a long time. Finally, these restrictions increase the chance that the compiler will be able to make use of atomic instructions directly for `std::atomic<UDT>` (and thus make a particular instantiation lock-free), because it can just treat the user-defined type as a set of raw bytes.

Note that although you can use `std::atomic<float>` or `std::atomic<double>`, because the built-in floating point types do satisfy the criteria for use with `memcpy` and `memcmp`, the behavior may be surprising in the case of `compare_exchange_strong` (`compare_exchange_weak` can always fail for arbitrary internal reasons, as described above). The operation may fail even though the old stored value was equal in value to the comparand, if the stored value had a different representation. Note that there are no atomic arithmetic operations on floating-point values. You'll get similar behavior with `compare_exchange_strong` if you use `std::atomic<>` with a user-defined type that has an equality-comparison operator defined, and that operator differs from the comparison using `memcmp`—the operation may fail because the otherwise-equal values have a different representation.

If your `UDT` is the same size as (or smaller than) an `int` or a `void*`, most common platforms will be able to use atomic instructions for `std::atomic<UDT>`. Some platforms will also be able to use atomic instructions for user-defined types that are twice the size of an `int` or `void*`. These platforms are typically those that support a so-called *double-word-compare-and-swap (DWCAS)* instruction corresponding to the `compare_exchange_xxx` functions. As you'll see in chapter 7, such support can be helpful when writing lock-free code.

These restrictions mean that you can't, for example, create a `std::atomic<std::vector<int>>` (since it has a non-trivial copy constructor and copy assignment operator), but you can instantiate `std::atomic<>` with classes containing counters or flags or pointers or even just arrays of simple data elements. This isn't particularly a problem; the more complex the data structure, the more likely you'll want to do operations on it other than simple assignment and comparison. If that's the case, you're better off using a `std::mutex` to ensure that the data is appropriately protected for the desired operations, as described in chapter 3.

As already mentioned, when instantiated with a user-defined type `T`, the interface of `std::atomic<T>` is limited to the set of operations available for `std::atomic<bool>`: `load()`, `store()`, `exchange()`, `compare_exchange_weak()`, `compare_exchange_strong()`, and assignment from and conversion to an instance of type `T`.

Table 5.3 shows the operations available on each atomic type.

Table 5.3 The operations available on atomic types

Operation	atomic_flag	atomic<bool>	atomic<T*>	atomic<integral-type>	atomic<other-type>
test_and_set		Y			
clear		Y			
is_lock_free		Y	Y	Y	Y
load		Y	Y	Y	Y
store		Y	Y	Y	Y
exchange		Y	Y	Y	Y
compare_exchange_weak, compare_exchange_strong		Y	Y	Y	Y
fetch_add, +=			Y	Y	
fetch_sub, -=			Y	Y	
fetch_or, =				Y	
fetch_and, &=				Y	
fetch_xor, ^=				Y	
++, --			Y	Y	

5.2.7 Free functions for atomic operations

Up until now I've limited myself to describing the member function forms of the operations on the atomic types. However, there are also equivalent nonmember functions for all the operations on the various atomic types. For the most part the nonmember functions are named after the corresponding member functions but with an `atomic_` prefix (for example, `std::atomic_load()`). These functions are then overloaded for each of the atomic types. Where there's opportunity for specifying a memory-ordering tag, they come in two varieties: one without the tag and one with an `_explicit` suffix and an additional parameter or parameters for the memory-ordering tag or tags (for example, `std::atomic_store(&atomic_var,new_value)` versus `std::atomic_store_explicit(&atomic_var,new_value,std::memory_order_release)`). Whereas the atomic object being referenced by the member functions is implicit, all the free functions take a pointer to the atomic object as the first parameter.

For example, `std::atomic_is_lock_free()` comes in just one variety (though overloaded for each type), and `std::atomic_is_lock_free(&a)` returns the same value as `a.is_lock_free()` for an object of atomic type `a`. Likewise, `std::atomic_load(&a)` is the same as `a.load()`, but the equivalent of `a.load(std::memory_order_acquire)` is `std::atomic_load_explicit(&a, std::memory_order_acquire)`.

The free functions are designed to be C-compatible, so they use pointers rather than references in all cases. For example, the first parameter of the `compare_exchange_weak()` and `compare_exchange_strong()` member functions (the expected value) is a reference, whereas the second parameter of `std::atomic_compare_exchange_weak()` (the first is the object pointer) is a pointer. `std::atomic_compare_exchange_weak_explicit()` also requires both the success and failure memory orders to be specified, whereas the compare-exchange member functions have both a single memory order form (with a default of `std::memory_order_seq_cst`) and an overload that takes the success and failure memory orders separately.

The operations on `std::atomic_flag` buck the trend, in that they spell out the "flag" part in the names: `std::atomic_flag_test_and_set()`, `std::atomic_flag_clear()`, although the additional variants that specify the memory ordering again have the `_explicit` suffix: `std::atomic_flag_test_and_set_explicit()` and `std::atomic_flag_clear_explicit()`.

The C++ Standard Library also provides free functions for accessing instances of `std::shared_ptr<>` in an atomic fashion. This is a break from the principle that only the atomic types support atomic operations, because `std::shared_ptr<>` is quite definitely *not* an atomic type (accessing the same `std::shared_ptr<T>` object from multiple threads without using the atomic access functions from all threads, or using suitable other external synchronization, is a data race and undefined behavior). However, the C++ Standards Committee felt it was sufficiently important to provide these extra functions. The atomic operations available are `load`, `store`, `exchange`, and `compare-exchange`, which are provided as overloads of the same operations on the standard atomic types, taking a `std::shared_ptr<>*` as the first argument:

```

std::shared_ptr<my_data> p;
void process_global_data()
{
    std::shared_ptr<my_data> local=std::atomic_load(&p);
    process_data(local);
}
void update_global_data()
{
    std::shared_ptr<my_data> local(new my_data);
    std::atomic_store(&p,local);
}

```

As with the atomic operations on other types, the `_explicit` variants are also provided to allow you to specify the desired memory ordering, and the `std::atomic_ is_lock_free()` function can be used to check whether the implementation uses locks to ensure the atomicity.

The Concurrency TS also provides `std::experimental::atomic_shared_ptr<T>`, which is an atomic type. To use it you must include the `<experimental/atomic>` header. It provides the same set of operations as `std::atomic<UDT>`: load, store, exchange, compare-exchange. It is provided as a separate type because that allows for a lock-free implementation that does not impose an additional cost on plain `std::shared_ptr` instances. However, just as with the `std::atomic` template, you still need to check whether it is lock-free on your platform, which can be tested with the `is_lock_free` member function. Even if it is not lock-free, `std::experimental::atomic_shared_ptr` is to be recommended over using the atomic free functions on a plain `std::shared_ptr`, as it is clearer in your code, and ensures that all accesses are atomic, and thus avoids the potential for data races due to forgetting to use the atomic free functions. As with all uses of atomic types and operations, if you are using them for a potential speed gain, it is important to profile, and compare with using alternative synchronization mechanisms.

As described in the introduction, the standard atomic types do more than just avoid the undefined behavior associated with a data race; they allow the user to enforce an ordering of operations between threads. This enforced ordering is the basis of the facilities for protecting data and synchronizing operations such as `std::mutex` and `std::future<>`. With that in mind, let's move on to the real meat of this chapter: the details of the concurrency aspects of the memory model and how atomic operations can be used to synchronize data and enforce ordering.

5.3 Synchronizing operations and enforcing ordering

Suppose you have two threads, one of which is populating a data structure to be read by the second. In order to avoid a problematic race condition, the first thread sets a flag to indicate that the data is ready, and the second thread doesn't read the data until the flag is set. The following listing shows such a scenario.

Listing 5.2 Reading and writing variables from different threads

```
#include <vector>
#include <atomic>
```

```

#include <iostream>
std::vector<int> data;
std::atomic<bool> data_ready(false);
void reader_thread()
{
    while(!data_ready.load())    #1
    {
        std::this_thread::sleep(std::chrono::milliseconds(1));
    }
    std::cout<<"The answer=<<data[0]<<"\n";    #2
}
void writer_thread()
{
    data.push_back(42);          #3
    data_ready=true;            #4
}

```

Leaving aside the inefficiency of the loop waiting for the data to be ready #1, you really need this to work, because otherwise sharing data between threads becomes impractical: every item of data is forced to be atomic. You've already learned that it's undefined behavior to have nonatomic reads #2 and writes #3 accessing the same data without an enforced ordering, so for this to work there must be an enforced ordering somewhere.

The required enforced ordering comes from the operations on the `std::atomic<bool>` variable `data_ready`; they provide the necessary ordering by virtue of the memory model relations *happens-before* and *synchronizes-with*. The write of the data #3 happens-before the write to the `data_ready` flag #4, and the read of the flag #1 happens-before the read of the data #2. When the value read from `data_ready` #1 is `true`, the write synchronizes-with that read, creating a *happens-before* relationship. Because *happens-before* is transitive, the write to the data #3 happens-before the write to the flag #4, which happens-before the read of the `true` value from the flag #1, which happens-before the read of the data #2, and you have an enforced ordering: the write of the data happens-before the read of the data and everything is OK. Figure 5.2 shows the important *happens-before* relationships in the two threads. I've added a couple of iterations of the `while` loop from the reader thread.

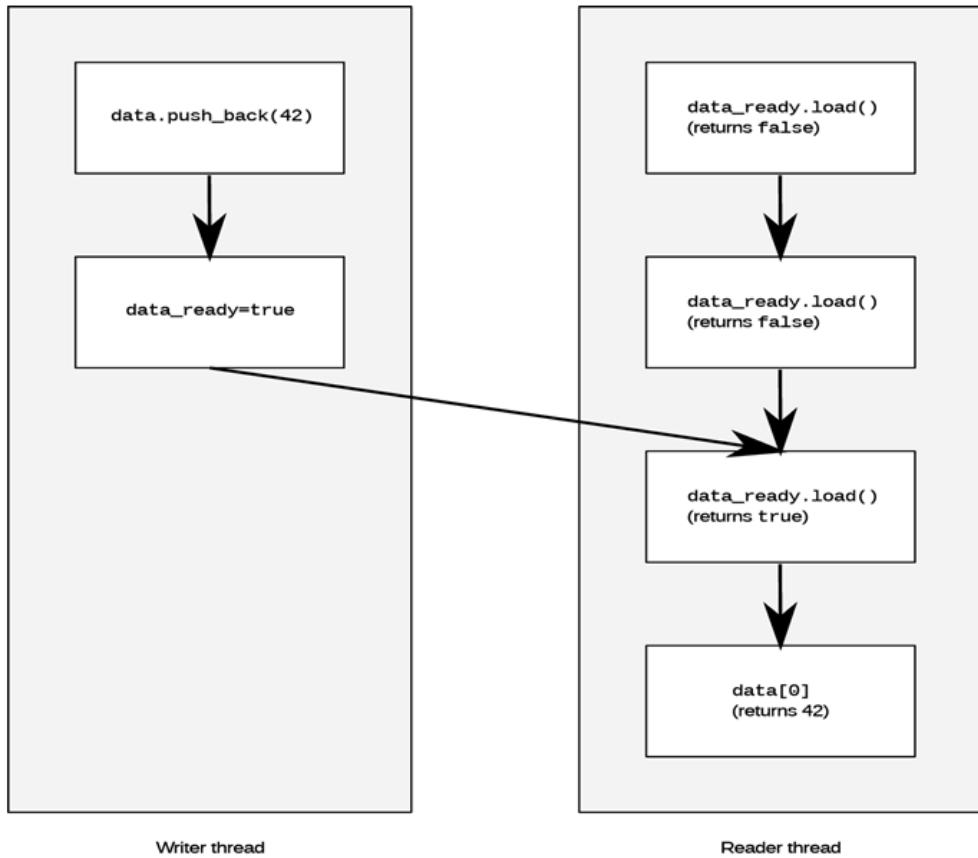


Figure 5.2 Enforcing an ordering between nonatomic operations using atomic operations

All this might seem fairly intuitive: of course the operation that writes a value happens before an operation that reads that value! With the default atomic operations, that's indeed true (which is why this is the default), but it does need spelling out: the atomic operations also have other options for the ordering requirements, which I'll come to shortly.

Now that you've seen happens-before and synchronizes-with in action, it's time to look at what they really mean. I'll start with synchronizes-with.

5.3.1 The synchronizes-with relationship

The synchronizes-with relationship is something that you can get only between operations on atomic types. Operations on a data structure (such as locking a mutex) might provide this relationship if the data structure contains atomic types and the operations on that data structure perform the appropriate atomic operations internally, but fundamentally it comes only from operations on atomic types.

The basic idea is this: a suitably tagged atomic write operation w on a variable x synchronizes-with a suitably tagged atomic read operation on x that reads the value stored by either that write (w), or a subsequent atomic write operation on x by the same thread that performed the initial write w , or a sequence of atomic read-modify-write operations on x (such as `fetch_add()` or `compare_exchange_weak()`) by any thread, where the value read by the first thread in the sequence is the value written by w (see section 5.3.4).

Leave the “suitably tagged” part aside for now, because all operations on atomic types are suitably tagged by default. This essentially means what you might expect: if thread A stores a value and thread B reads that value, there’s a synchronizes-with relationship between the store in thread A and the load in thread B, just as in listing 5.2.

As I’m sure you’ve guessed, the nuances are all in the “suitably tagged” part. The C++ memory model allows various ordering constraints to be applied to the operations on atomic types, and this is the tagging to which I refer. The various options for memory ordering and how they relate to the synchronizes-with relationship are covered in section 5.3.3. First, let’s step back and look at the happens-before relationship.

5.3.2 The happens-before relationship

The *happens-before* relationship is the basic building block of operation ordering in a program; it specifies which operations see the effects of which other operations. For a single thread, it’s largely straightforward: if one operation is sequenced before another, then it also happens-before it. This means that if one operation (A) occurs in a statement prior to another (B) in the source code, then A happens-before B. You saw that in listing 5.2: the write to `data #3` happens-before the write to `data_ready #4`. If the operations occur in the same statement, in general there’s no happens-before relationship between them, because they’re unordered. This is just another way of saying that the ordering is unspecified. You know that the program in the following listing will output “1,2” or “2,1”, but it’s unspecified which, because the order of the two calls to `get_num()` is unspecified.

Listing 5.3 Order of evaluation of arguments to a function call is unspecified

```
#include <iostream>
void foo(int a,int b)
{
    std::cout<<a<<","<<b<<std::endl;
}
int get_num()
{
    static int i=0;
    return ++i;
}
int main()
{
    foo(get_num(),get_num());    #A
}
#A Calls to get_num() are unordered
```

There are circumstances where operations within a single statement are sequenced such as where the built-in comma operator is used or where the result of one expression is used as an argument to another expression. But in general, operations within a single statement are nonsequenced, and there's no sequenced-before (and thus no happens-before) relationship between them. Of course, all operations in a statement happen before all of the operations in the next statement.

This is really just a restatement of the single-threaded sequencing rules you're used to, so what's new? The new part is the interaction between threads: if operation A on one thread inter-thread happens-before operation B on another thread, then A happens-before B. This doesn't really help much: you've just added a new relationship (inter-thread happens-before), but this is an important relationship when you're writing multithreaded code.

At the basic level, inter-thread happens-before is relatively simple and relies on the synchronizes-with relationship introduced in section 5.3.1: if operation A in one thread synchronizes-with operation B in another thread, then A inter-thread happens-before B. It's also a transitive relation: if A inter-thread happens-before B and B inter-thread happens-before C, then A inter-thread happens-before C. You saw this in listing 5.2 as well.

Inter-thread happens-before also combines with the sequenced-before relation: if operation A is sequenced before operation B, and operation B inter-thread happens-before operation C, then A inter-thread happens-before C. Similarly, if A synchronizes-with B and B is sequenced before C, then A inter-thread happens-before C. These two together mean that if you make a series of changes to data in a single thread, you need only one synchronizes-with relationship for the data to be visible to subsequent operations on the thread that executed C.

These are the crucial rules that enforce ordering of operations between threads and make everything in listing 5.2 work. There are some additional nuances with data dependency, as you'll see shortly. In order for you to understand this, I need to cover the memory-ordering tags used for atomic operations and how they relate to the synchronizes-with relation.

5.3.3 Memory ordering for atomic operations

There are six memory ordering options that can be applied to operations on atomic types: `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`. Unless you specify otherwise for a particular operation, the memory-ordering option for all operations on atomic types is `memory_order_seq_cst`, which is the most stringent of the available options. Although there are six ordering options, they represent three models: *sequentially consistent* ordering (`memory_order_seq_cst`), *acquire-release* ordering (`memory_order_consume`, `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel`), and *relaxed* ordering (`memory_order_relaxed`).

These distinct memory-ordering models can have varying costs on different CPU architectures. For example, on systems based on architectures with fine control over the visibility of operations by processors other than the one that made the change, additional synchronization instructions can be required for sequentially consistent ordering over acquire-

release ordering or relaxed ordering and for acquire-release ordering over relaxed ordering. If these systems have many processors, these additional synchronization instructions may take a significant amount of time, thus reducing the overall performance of the system. On the other hand, CPUs that use the x86 or x86-64 architectures (such as the Intel and AMD processors common in desktop PCs) don't require any additional instructions for acquire-release ordering beyond those necessary for ensuring atomicity, and even sequentially-consistent ordering doesn't require any special treatment for load operations, although there's a small additional cost on stores.

The availability of the distinct memory-ordering models allows experts to take advantage of the increased performance of the more fine-grained ordering relationships where they're advantageous while allowing the use of the default sequentially-consistent ordering (which is considerably easier to reason about than the others) for those cases that are less critical.

In order to choose which ordering model to use, or to understand the ordering relationships in code that uses the different models, it's important to know how the choices affect the program behavior. Let's therefore look at the consequences of each choice for operation ordering and synchronizes-with.

SEQUENTIALLY CONSISTENT ORDERING

The default ordering is named *sequentially consistent* because it implies that the behavior of the program is consistent with a simple sequential view of the world. If all operations on instances of atomic types are sequentially consistent, the behavior of a multithreaded program is as if all these operations were performed in some particular sequence by a single thread. This is by far the easiest memory ordering to understand, which is why it's the default: all threads must see the same order of operations. This makes it easy to reason about the behavior of code written with atomic variables. You can write down all the possible sequences of operations by different threads, eliminate those that are inconsistent, and verify that your code behaves as expected in the others. It also means that operations can't be reordered; if your code has one operation before another in one thread, that ordering must be seen by all other threads.

From the point of view of synchronization, a sequentially consistent store synchronizes-with a sequentially consistent load of the same variable that reads the value stored. This provides one ordering constraint on the operation of two (or more) threads, but sequential consistency is more powerful than that. Any sequentially consistent atomic operations done after that load must also appear after the store to other threads in the system using sequentially consistent atomic operations. The example in listing 5.4 demonstrates this ordering constraint in action. This constraint doesn't carry forward to threads that use atomic operations with relaxed memory orderings; they can still see the operations in a different order, so you must use sequentially consistent operations on all your threads in order to get the benefit.

This ease of understanding can come at a price, though. On a weakly ordered machine with many processors, it can impose a noticeable performance penalty, because the overall sequence of operations must be kept consistent between the processors, possibly requiring

extensive (and expensive!) synchronization operations between the processors. That said, some processor architectures (such as the common x86 and x86-64 architectures) offer sequential consistency relatively cheaply, so if you're concerned about the performance implications of using sequentially consistent ordering, check the documentation for your target processor architectures.

The following listing shows sequential consistency in action. The loads and stores to `x` and `y` are explicitly tagged with `memory_order_seq_cst`, although this tag could be omitted in this case because it's the default.

Listing 5.4 Sequential consistency implies a total ordering

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
    x.store(true,std::memory_order_seq_cst);    #1
}
void write_y()
{
    y.store(true,std::memory_order_seq_cst);    #2
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))        #3
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))        #4
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);    #5
}
```

The `assert #5` can never fire, because either the store to `x` #1 or the store to `y` #2 must happen first, even though it's not specified which. If the load of `y` in `read_x_then_y` #3

returns `false`, the store to `x` must occur before the store to `y`, in which case the load of `x` in `read_y_then_x` #4 must return `true`, because the `while` loop ensures that the `y` is `true` at this point. Because the semantics of `memory_order_seq_cst` require a single total ordering over all operations tagged `memory_order_seq_cst`, there's an implied ordering relationship between a load of `y` that returns `false` #3 and the store to `y` #1. For there to be a single total order, if one thread sees `x==true` and then subsequently sees `y==false`, this implies that the store to `x` occurs before the store to `y` in this total order.

Of course, because everything is symmetrical, it could also happen the other way around, with the load of `x` #4 returning `false`, forcing the load of `y` #3 to return `true`. In both cases, `z` is equal to 1. Both loads can return `true`, leading to `z` being 2, but under no circumstances can `z` be zero.

The operations and happens-before relationships for the case that `read_x_then_y` sees `x` as `true` and `y` as `false` are shown in figure 5.3. The dashed line from the load of `y` in `read_x_then_y` to the store to `y` in `write_y` shows the implied ordering relationship required in order to maintain sequential consistency: the load must occur before the store in the global order of `memory_order_seq_cst` operations in order to achieve the outcomes given here.

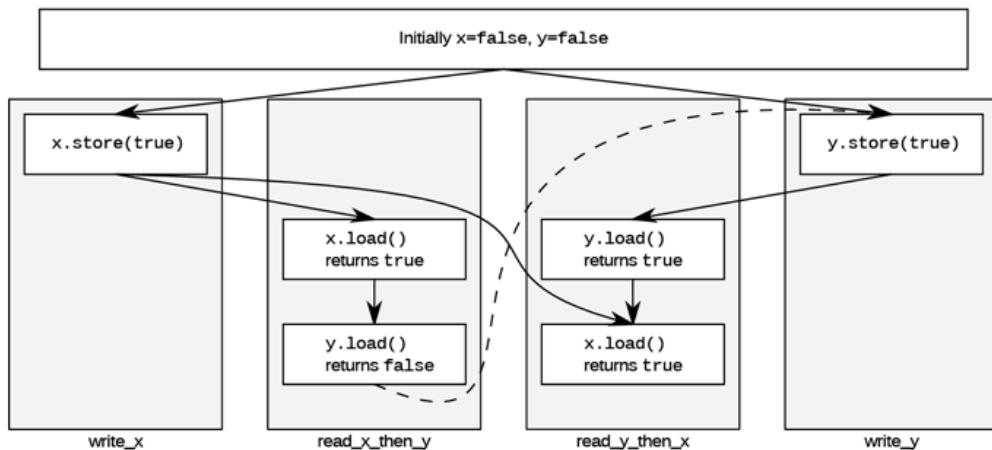


Figure 5.3 Sequential consistency and happens-before

Sequential consistency is the most straightforward and intuitive ordering, but it's also the most expensive memory ordering because it requires global synchronization between all threads. On a multiprocessor system this may require quite extensive and time-consuming communication between processors.

In order to avoid this synchronization cost, you need to step outside the world of sequential consistency and consider using other memory orderings.

NON-SEQUENTIALLY CONSISTENT MEMORY ORDERINGS

Once you step outside the nice sequentially consistent world, things start to get complicated. Probably the single biggest issue to come to grips with is the fact that *there's no longer a single global order of events*. This means that different threads can see different views of the same operations, and any mental model you have of operations from different threads neatly interleaved one after the other must be thrown away. Not only do you have to account for things happening truly concurrently, but *threads don't have to agree on the order of events*. In order to write (or even just to understand) any code that uses a memory ordering other than the default `memory_order_seq_cst`, it's absolutely vital to get your head around this. It's not just that the compiler can reorder the instructions. Even if the threads are running the same bit of code, they can disagree on the order of events because of operations in other threads in the absence of explicit ordering constraints, because the different CPU caches and internal buffers can hold different values for the same memory. It's so important I'll say it again: *threads don't have to agree on the order of events*.

Not only do you have to throw out mental models based on interleaving operations, you also have to throw out mental models based on the idea of the compiler or processor reordering the instructions. *In the absence of other ordering constraints, the only requirement is that all threads agree on the modification order of each individual variable*. Operations on distinct variables can appear in different orders on different threads, provided the values seen are consistent with any additional ordering constraints imposed.

This is best demonstrated by stepping completely outside the sequentially consistent world and using `memory_order_relaxed` for all operations. Once you've come to grips with that, you can move back to acquire-release ordering, which allows you to selectively introduce ordering relationships between operations and claw back some of your sanity.

RELAXED ORDERING

Operations on atomic types performed with relaxed ordering don't participate in synchronizes-with relationships. Operations on the same variable within a single thread still obey happens-before relationships, but there's almost no requirement on ordering relative to other threads. The only requirement is that accesses to a single atomic variable from the same thread can't be reordered; once a given thread has seen a particular value of an atomic variable, a subsequent read by that thread can't retrieve an earlier value of the variable. Without any additional synchronization, the modification order of each variable is the only thing shared between threads that are using `memory_order_relaxed`.

To demonstrate just how relaxed your relaxed operations can be, you need only two threads, as shown in the following listing.

Listing 5.5 Relaxed operations have very few ordering requirements

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
```

```

std::atomic<int> z;
void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);    #1
    y.store(true,std::memory_order_relaxed);    #2
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));    #3
    if(x.load(std::memory_order_relaxed))        #4
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);    #5
}

```

This time the assert #5 *can* fire, because the load of x #4 can read `false`, even though the load of y #3 reads `true` and the store of x #1 happens-before the store of y #2. x and y are different variables, so there are no ordering guarantees relating to the visibility of values arising from operations on each.

Relaxed operations on different variables can be freely reordered provided they obey any happens-before relationships they're bound by (for example, within the same thread). They don't introduce synchronizes-with relationships. The happens-before relationships from listing 5.5 are shown in figure 5.4, along with a possible outcome. Even though there's a happens-before relationship between the stores and between the loads, there isn't one between either store and either load, and so the loads can see the stores out of order.

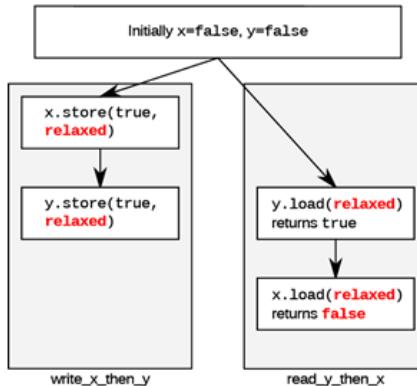


Figure 5.4 Relaxed atomics and happens-before

Let's look at the slightly more complex example with three variables and five threads in the next listing.

Listing 5.6 Relaxed operations on multiple threads

```
#include <thread>
#include <atomic>
#include <iostream>
std::atomic<int> x(0),y(0),z(0);      #1
std::atomic<bool> go(false);          #2
unsigned const loop_count=10;
struct read_values
{
    int x,y,z;
};
read_values values1[loop_count];
read_values values2[loop_count];
read_values values3[loop_count];
read_values values4[loop_count];
read_values values5[loop_count];
void increment(std::atomic<int>* var_to_inc,read_values* values)
{
    while(!go)                      #3
        std::this_thread::yield();
    for(unsigned i=0;i<loop_count;++i)
    {
        values[i].x=x.load(std::memory_order_relaxed);
        values[i].y=y.load(std::memory_order_relaxed);
        values[i].z=z.load(std::memory_order_relaxed);
        var_to_inc->store(i+1,std::memory_order_relaxed);   #4
        std::this_thread::yield();
    }
}
void read_vals(read_values* values)
{
    while(!go)                      #5
        std::this_thread::yield();
    for(unsigned i=0;i<loop_count;++i)
    {
        values[i].x=x.load(std::memory_order_relaxed);
        values[i].y=y.load(std::memory_order_relaxed);
        values[i].z=z.load(std::memory_order_relaxed);
        std::this_thread::yield();
    }
}
void print(read_values* v)
{
    for(unsigned i=0;i<loop_count;++i)
    {
        if(i)
            std::cout<<"("<<v[i].x<<","<<v[i].y<<","<<v[i].z<<")";
    }
    std::cout<<std::endl;
}
int main()
{
    std::thread t1(increment,&x,values1);
```

```

    std::thread t2(increment,&y,values2);
    std::thread t3(increment,&z,values3);
    std::thread t4(read_vals,values4);
    std::thread t5(read_vals,values5);
    go=true;                                #6
    t5.join();
    t4.join();
    t3.join();
    t2.join();
    t1.join();
    print(values1);                         #7
    print(values2);
    print(values3);
    print(values4);
    print(values5);
}

```

#3 Spin, waiting for the signal
#5 Spin, waiting for the signal
#6 Signal to start execution of main loop
#7 Print the final values

This is a really simple program in essence. You have three shared global atomic variables #1 and five threads. Each thread loops 10 times, reading the values of the three atomic variables using `memory_order_relaxed` and storing them in an array. Three of the threads each update one of the atomic variables each time through the loop #4, while the other two threads just read. Once all the threads have been joined, you print the values from the arrays stored by each thread #7.

The atomic variable `go` #2 is used to ensure that the threads all start the loop as near to the same time as possible. Launching a thread is an expensive operation, and without the explicit delay, the first thread may be finished before the last one has started. Each thread waits for `go` to become `true` before entering the main loop #3, #5, and `go` is set to `true` only once all the threads have started #6.

One possible output from this program is as follows:

```
(0,0,0),(1,0,0),(2,0,0),(3,0,0),(4,0,0),(5,7,0),(6,7,8),(7,9,8),(8,9,8),
(9,9,10)
(0,0,0),(0,1,0),(0,2,0),(1,3,5),(8,4,5),(8,5,5),(8,6,6),(8,7,9),(10,8,9),
(10,9,10)
(0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(0,0,5),(0,0,6),(0,0,7),(0,0,8),
(0,0,9)
(1,3,0),(2,3,0),(2,4,1),(3,6,4),(3,9,5),(5,10,6),(5,10,8),(5,10,10),
(9,10,10),(10,10,10)
(0,0,0),(0,0,0),(0,0,0),(6,3,7),(6,5,7),(7,7,7),(7,8,7),(8,8,7),(8,8,9),
(8,8,9)
```

The first three lines are the threads doing the updating, and the last two are the threads doing just reading. Each triplet is a set of the variables `x`, `y` and `z` in that order from one pass through the loop. There are a few things to notice from this output:

- The first set of values shows `x` increasing by one with each triplet, the second set has `y` increasing by one, and the third has `z` increasing by one.

- The x elements of each triplet only increase within a given set, as do the y and z elements, but the increments are uneven, and the relative orderings vary between all threads.
- Thread 3 doesn't see any of the updates to x or y ; it sees only the updates it makes to z . This doesn't stop the other threads from seeing the updates to z mixed in with the updates to x and y though.

This is a valid outcome for relaxed operations, but it's not the only valid outcome. Any set of values that's consistent with the three variables each holding the values 0 to 10 in turn and that has the thread incrementing a given variable printing the values 0 to 9 for that variable is valid.

UNDERSTANDING RELAXED ORDERING

To understand how this works, imagine that each variable is a man in a cubicle with a notepad. On his notepad is a list of values. You can phone him and ask him to give you a value, or you can tell him to write down a new value. If you tell him to write down a new value, he writes it at the bottom of the list. If you ask him for a value, he reads you a number from the list.

The first time you talk to this man, if you ask him for a value, he may give you *any* value from the list he has on his pad at the time. If you then ask him for another value, he may give you the same one again or a value from farther down the list. He'll never give you a value from farther up the list. If you tell him to write down a number and then subsequently ask him for a value, he'll give you either the number you told him to write down or a number below that on the list.

Imagine for a moment that his list starts with the values 5, 10, 23, 3, 1, 2. If you ask for a value, you could get any of those. If he gives you 10, then the next time you ask he could give you 10 again, or any of the later ones, but not 5. If you call him five times, he could say "10, 10, 1, 2, 2," for example. If you tell him to write down 42, he'll add it to the end of the list. If you ask him for a number again, he'll keep telling you "42" until he has another number on his list and he feels like telling it to you.

Now, imagine your friend Carl also has this man's number. Carl can also phone him and either ask him to write down a number or ask for one, and he applies the same rules to Carl as he does to you. He has only one phone, so he can only deal with one of you at a time, so the list on his pad is a nice straightforward list. However, just because you got him to write down a new number doesn't mean he has to tell it to Carl, and vice versa. If Carl asked him for a number and was told "23," then just because you asked the man to write down 42 doesn't mean he'll tell that to Carl next time. He may tell Carl any of the numbers 23, 3, 1, 2, 42, or even the 67 that Fred told him to write down after you called. He could very well tell Carl "23, 3, 3, 1, 67" without being inconsistent with what he told you. It's like he keeps track of which number he told to whom with a little movable sticky note for each person, like in figure 5.5.

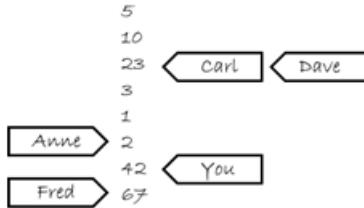


Figure 5.5 The notebook for the man in the cubicle

Now imagine that there's not just one man in a cubicle but a whole cubicle farm, with loads of men with phones and notepads. These are all our atomic variables. Each variable has its own modification order (the list of values on the pad), but there's no relationship between them at all. If each caller (you, Carl, Anne, Dave, and Fred) is a thread, then this is what you get when every operation uses `memory_order_relaxed`. There are a few additional things you can tell the man in the cubicle, such as "write down this number, and tell me what was at the bottom of the list" (`exchange`) and "write down *this* number if the number on the bottom of the list is *that*; otherwise tell me what I should have guessed" (`compare_exchange_strong`), but that doesn't affect the general principle.

If you think about the program logic from listing 5.5, then `write_x_then_y` is like some guy calling up the man in cubicle `x` and telling him to write `true` and then calling up the man in cubicle `y` and telling *him* to write `true`. The thread running `read_y_then_x` repeatedly calls up the man in cubicle `y` asking for a value until he says `true` and then calls the man in cubicle `x` to ask for a value. The man in cubicle `x` is under no obligation to tell you any specific value off his list and is quite within his rights to say `false`.

This makes relaxed atomic operations difficult to deal with. They must be used in combination with atomic operations that feature stronger ordering semantics in order to be useful for inter-thread synchronization. I strongly recommend avoiding relaxed atomic operations unless they're absolutely necessary and even then using them only with extreme caution. Given the unintuitive results that can be achieved with just two threads and two variables in listing 5.5, it's not hard to imagine the possible complexity when more threads and more variables are involved.

One way to achieve additional synchronization without the overhead of full-blown sequential consistency is to use *acquire-release ordering*.

ACQUIRE-RELEASE ORDERING

Acquire-release ordering is a step up from relaxed ordering; there's still no total order of operations, but it does introduce some synchronization. Under this ordering model, atomic loads are *acquire* operations (`memory_order_acquire`), atomic stores are *release* operations (`memory_order_release`), and atomic read-modify-write operations (such as `fetch_add()` or `exchange()`) are either *acquire*, *release*, or both (`memory_order_acq_rel`). Synchronization is pairwise, between the thread that does the release and the thread that does the acquire. A

release operation synchronizes-with an *acquire* operation that reads the value written. This means that different threads can still see different orderings, but these orderings are restricted. The following listing is a rework of listing 5.4 using acquire-release semantics rather than sequentially consistent ones.

Listing 5.7 Acquire-release doesn't imply a total ordering

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
    x.store(true,std::memory_order_release);
}
void write_y()
{
    y.store(true,std::memory_order_release);
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_acquire));
    if(y.load(std::memory_order_acquire))          #1
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));
    if(x.load(std::memory_order_acquire))          #2
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);   #3
}
```

In this case the assert #3 *can* fire (just like in the relaxed-ordering case), because it's possible for both the load of x #2 and the load of y #1 to read false. x and y are written by different threads, so the ordering from the release to the acquire in each case has no effect on the operations in the other threads.

Figure 5.6 shows the happens-before relationships from listing 5.7, along with a possible outcome where the two reading threads each have a different view of the world. This is

possible because there's no happens-before relationship to force an ordering, as described previously.

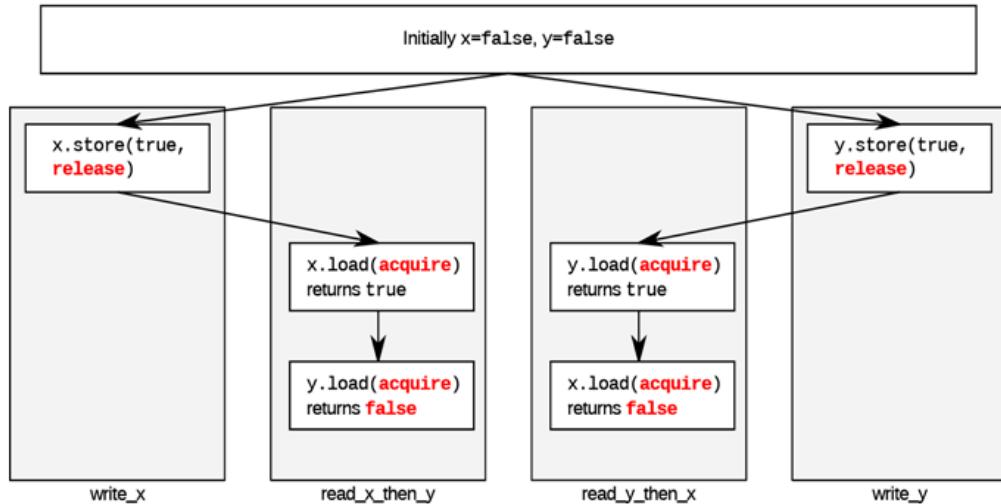


Figure 5.6 Acquire-release and happens-before

In order to see the benefit of acquire-release ordering, you need to consider two stores from the same thread, like in listing 5.5. If you change the store to `y` to use `memory_order_release` and the load from `y` to use `memory_order_acquire` like in the following listing, then you actually impose an ordering on the operations on `x`.

Listing 5.8 Acquire-release operations can impose ordering on relaxed operations

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);      #1
    y.store(true,std::memory_order_release);        #2
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));      #3
    if(x.load(std::memory_order_relaxed))           #4
        ++z;
}
int main()
{
    x=false;
    y=false;
```

```

z=0;
std::thread a(write_x_then_y);
std::thread b(read_y_then_x);
a.join();
b.join();
assert(z.load()!=0);    #5
}

```

#3 Spin, waiting for y to be set to true

Eventually, the load from `y` #3 will see `true` as written by the store #2. Because the store uses `memory_order_release` and the load uses `memory_order_acquire`, the store synchronizes-with the load. The store to `x` #1 happens-before the store to `y` #2, because they're in the same thread. Because the store to `y` synchronizes-with the load from `y`, the store to `x` also happens-before the load from `y` and by extension happens-before the load from `x` #4. Thus the load from `x` *must* read `true`, and the assert #5 *can't* fire. If the load from `y` wasn't in a while loop, this wouldn't necessarily be the case; the load from `y` might read `false`, in which case there'd be no requirement on the value read from `x`. In order to provide any synchronization, acquire and release operations must be paired up. The value stored by a release operation must be seen by an acquire operation for either to have any effect. If either the store at #2 or the load at #3 was a relaxed operation, there'd be no ordering on the accesses to `x`, so there'd be no guarantee that the load at #4 would read `true`, and the assert could fire.

You can still think about acquire-release ordering in terms of our men with notepads in their cubicles, but you have to add more to the model. First, imagine that every store that's done is part of some batch of updates, so when you call a man to tell him to write down a number, you also tell him which batch this update is part of: "Please write down 99, as part of batch 423." For the last store in a batch, you tell this to the man too: "Please write down 147, which is the last store in batch 423." The man in the cubicle will then duly write down this information, along with who gave him the value. This models a store-release operation. The next time you tell someone to write down a value, you increase the batch number: "Please write down 41, as part of batch 424."

When you ask for a value, you now have a choice: you can either just ask for a value (which is a relaxed load), in which case the man just gives you the number, or you can ask for a value and information about whether it's the last in a batch (which models a load-acquire). If you ask for the batch information, and the value wasn't the last in a batch, the man will tell you something like, "The number is 987, which is just a 'normal' value," whereas if it *was* the last in a batch, he'll tell you something like "The number is 987, which is the last number in batch 956 from Anne." Now, here's where the acquire-release semantics kick in: if you tell the man all the batches you know about when you ask for a value, he'll look down his list for the last value from any of the batches you know about and either give you that number or one further down the list.

How does this model acquire-release semantics? Let's look at our example and see. First off, thread `a` is running `write_x_then_y` and says to the man in cubicle `x`, "Please write `true`

as part of batch 1 from thread a," which he duly writes down. Thread a then says to the man in cubicle y, "Please write `true` as the last write of batch 1 from thread a," which he duly writes down. In the meantime, thread b is running `read_y_then_x`. Thread b keeps asking the man in box y for a value with batch information until he says "true." He may have to ask many times, but eventually the man will say "true." The man in box y doesn't *just* say "true" though; he also says, "This is the last write in batch 1 from thread a."

Now, thread b goes on to ask the man in box x for a value, but this time he says, "Please can I have a value, and by the way I know about batch 1 from thread a." So now, the man from cubicle x has to look down his list for the last mention of batch 1 from thread a. The only mention he has is the value `true`, which is also the last value on his list, so he *must* read out that value; otherwise, he's breaking the rules of the game.

If you look at the definition of *inter-thread happens-before* back in section 5.3.2, one of the important properties is that it's transitive: if A *inter-thread happens-before* B and B *inter-thread happens-before* C, then A *inter-thread happens-before* C. This means that acquire-release ordering can be used to synchronize data across several threads, even when the "intermediate" threads haven't actually touched the data.

TRANSITIVE SYNCHRONIZATION WITH ACQUIRE-RELEASE ORDERING

In order to think about transitive ordering, you need at least three threads. The first thread modifies some shared variables and does a store-release to one of them. A second thread then reads the variable subject to the store-release with a load-acquire and performs a store-release on a second shared variable. Finally, a third thread does a load-acquire on that second shared variable. Provided that the load-acquire operations see the values written by the store-release operations to ensure the synchronizes-with relationships, this third thread can read the values of the other variables stored by the first thread, even if the intermediate thread didn't touch any of them. This scenario is shown in the next listing.

Listing 5.9 Transitive synchronization using acquire and release ordering

```
std::atomic<int> data[5];
std::atomic<bool> sync1(false),sync2(false);
void thread_1()
{
    data[0].store(42,std::memory_order_relaxed);
    data[1].store(97,std::memory_order_relaxed);
    data[2].store(17,std::memory_order_relaxed);
    data[3].store(-141,std::memory_order_relaxed);
    data[4].store(2003,std::memory_order_relaxed);
    sync1.store(true,std::memory_order_release);      #1
}
void thread_2()
{
    while(!sync1.load(std::memory_order_acquire));  #2
    sync2.store(true,std::memory_order_release);      #3
}
void thread_3()
{
    while(!sync2.load(std::memory_order_acquire));      #4
```

```

        assert(data[0].load(std::memory_order_relaxed)==42);
        assert(data[1].load(std::memory_order_relaxed)==97);
        assert(data[2].load(std::memory_order_relaxed)==17);
        assert(data[3].load(std::memory_order_relaxed)==-141);
        assert(data[4].load(std::memory_order_relaxed)==2003);
    }

#1 Set sync1
#2 Loop until sync1 is set
#3 Set sync2
#4 Loop until sync2 is set

```

Even though `thread_2` only touches the variables `sync1` #2 and `sync2` #3, this is enough for synchronization between `thread_1` and `thread_3` to ensure that the asserts don't fire. First off, the stores to `data` from `thread_1` happens-before the store to `sync1` #1, because they're sequenced-before it in the same thread. Because the load from `sync1` #1 is in a while loop, it will eventually see the value stored from `thread_1` and thus form the second half of the release-acquire pair. Therefore, the store to `sync1` happens-before the final load from `sync2` #3, which forms a release-acquire pair with the final load from the while loop in `thread_3` #4. The store to `sync2` #3 thus happens-before the load #4, which happens-before the loads from `data`. Because of the transitive nature of happens-before, you can chain it all together: the stores to `data` happen-before the store to `sync1` #1, which happens-before the load from `sync1` #2, which happens-before the store to `sync2` #3, which happens-before the load from `sync2` #4, which happens-before the loads from `data`. Thus the stores to `data` in `thread_1` happen-before the loads from `data` in `thread_3`, and the asserts can't fire.

In this case, you could combine `sync1` and `sync2` into a single variable by using a read-modify-write operation with `memory_order_acq_rel` in `thread_2`. One option would be to use `compare_exchange_strong()` to ensure that the value is updated only once the store from `thread_1` has been seen:

```

std::atomic<int> sync(0);
void thread_1()
{
    // ...
    sync.store(1,std::memory_order_release);
}
void thread_2()
{
    int expected=1;
    while(!sync.compare_exchange_strong(expected,2,
                                         std::memory_order_acq_rel))
        expected=1;
}
void thread_3()
{
    while(sync.load(std::memory_order_acquire)<2);
    // ...
}

```

If you use read-modify-write operations, it's important to pick which semantics you desire. In this case, you want both acquire and release semantics, so `memory_order_acq_rel` is appropriate, but you can use other orderings too. A `fetch_sub` operation with `memory_order_acquire` semantics doesn't synchronize-with anything, even though it stores a value, because it isn't a release operation. Likewise, a store can't synchronize-with a `fetch_or` with `memory_order_release` semantics, because the read part of the `fetch_or` isn't an acquire operation. Read-modify-write operations with `memory_order_acq_rel` semantics behave as both an acquire and a release, so a prior store can synchronize-with such an operation, and it can synchronize-with a subsequent load, as is the case in this example.

If you mix acquire-release operations with sequentially consistent operations, the sequentially consistent loads behave like loads with acquire semantics, and sequentially consistent stores behave like stores with release semantics. Sequentially consistent read-modify-write operations behave as both acquire and release operations. Relaxed operations are still relaxed but are bound by the additional synchronizes-with and consequent happens-before relationships introduced through the use of acquire-release semantics.

Despite the potentially non-intuitive outcomes, anyone who's used locks has had to deal with the same ordering issues: locking a mutex is an acquire operation, and unlocking the mutex is a release operation. With mutexes, you learn that you must ensure that the same mutex is locked when you read a value as was locked when you wrote it, and the same applies here; your acquire and release operations have to be on the same variable to ensure an ordering. If data is protected with a mutex, the exclusive nature of the lock means that the result is indistinguishable from what it would have been had the lock and unlock been sequentially consistent operations. Similarly, if you use acquire and release orderings on atomic variables to build a simple lock, then from the point of view of code that uses the lock, the behavior will appear sequentially consistent, even though the internal operations are not.

If you don't need the stringency of sequentially consistent ordering for your atomic operations, the pair-wise synchronization of acquire-release ordering has the potential for a much lower synchronization cost than the global ordering required for sequentially consistent operations. The trade-off here is the mental cost required to ensure that the ordering works correctly and that the non-intuitive behavior across threads isn't problematic.

DATA DEPENDENCY WITH ACQUIRE-RELEASE ORDERING AND MEMORY_ORDER_CONSUME

In the introduction to this section I said that `memory_order_consume` was part of the acquire-release ordering model, but it was conspicuously absent from the preceding description. This is because `memory_order_consume` is special: it's all about data dependencies, and it introduces the data-dependency nuances to the inter-thread happens-before relationship mentioned in section 5.3.2. It is also special in that **the C++17 standard explicitly recommends that you do not use it**. It is therefore only covered here for completeness: **you should not use `memory_order_consume` in your code!**

The concept of a data dependency is relatively straightforward: there is a data dependency between two operations if the second one operates on the result of the first. There are two

new relations that deal with data dependencies: *dependency-ordered-before* and *carries-a-dependency-to*. Just like sequenced-before, carries-a-dependency-to applies strictly within a single thread and essentially models the data dependency between operations; if the result of an operation A is used as an operand for an operation B, then A carries-a-dependency-to B. If the result of operation A is a value of a scalar type such as an `int`, then the relationship still applies if the result of A is stored in a variable, and that variable is then used as an operand for operation B. This operation is also transitive, so if A carries-a-dependency-to B, and B carries-a-dependency-to C, then A carries-a-dependency-to C.

On the other hand, the dependency-ordered-before relationship can apply between threads. It's introduced by using atomic load operations tagged with `memory_order_consume`. This is a special case of `memory_order_acquire` that limits the synchronized data to direct dependencies; a store operation A tagged with `memory_order_release`, `memory_order_acq_rel`, or `memory_order_seq_cst` is dependency-ordered-before a load operation B tagged with `memory_order_consume` if the consume reads the value stored. This is as opposed to the synchronizes-with relationship you get if the load uses `memory_order_acquire`. If this operation B then carries-a-dependency-to some operation C, then A is also dependency-ordered-before C.

This wouldn't actually do you any good for synchronization purposes if it didn't affect the inter-thread happens-before relation, but it does: if A is dependency-ordered-before B, then A also inter-thread happens-before B.

One important use for this kind of memory ordering is where the atomic operation loads a pointer to some data. By using `memory_order_consume` on the load and `memory_order_release` on the prior store, you ensure that the pointed-to data is correctly synchronized, without imposing any synchronization requirements on any other nondependent data. The following listing shows an example of this scenario.

Listing 5.10 Using `std::memory_order_consume` to synchronize data

```
struct X
{
    int i;
    std::string s;
};

std::atomic<X*> p;
std::atomic<int> a;
void create_x()
{
    X* x=new X;
    x->i=42;
    x->s="hello";
    a.store(99,std::memory_order_relaxed);      #1
    p.store(x,std::memory_order_release);        #2
}
void use_x()
{
    X* x;
    while(!(x=p.load(std::memory_order_consume)))      #3
        std::this_thread::sleep(std::chrono::microseconds(1));
```

```

    assert(x->i==42);                                #4
    assert(x->s=="hello");                           #5
    assert(a.load(std::memory_order_relaxed)==99);     #6
}
int main()
{
    std::thread t1(create_x);
    std::thread t2(use_x);
    t1.join();
    t2.join();
}

```

Even though the store to `a` #1 is sequenced before the store to `p` #2, and the store to `p` is tagged `memory_order_release`, the load of `p` #3 is tagged `memory_order_consume`. This means that the store to `p` only happens-before those expressions that are dependent on the value loaded from `p`. This means that the asserts on the data members of the `x` structure #4, #5 are guaranteed not to fire, because the load of `p` carries a dependency to those expressions through the variable `x`. On the other hand, the assert on the value of `a` #6 may or may not fire; this operation isn't dependent on the value loaded from `p`, and so there's no guarantee on the value that's read. This is particularly apparent because it's tagged with `memory_order_relaxed`, as you'll see.

Sometimes, you don't want the overhead of carrying the dependency around. You want the compiler to be able to cache values in registers and reorder operations to optimize the code rather than fussing about the dependencies. In these scenarios, you can use `std::kill_dependency()` to explicitly break the dependency chain. `std::kill_dependency()` is a simple function template that copies the supplied argument to the return value but breaks the dependency chain in doing so. For example, if you have a global read-only array, and you use `std::memory_order_consume` when retrieving an index into that array from another thread, you can use `std::kill_dependency()` to let the compiler know that it doesn't need to reread the contents of the array entry, as in the following example:

```

int global_data[]={ ... };
std::atomic<int> index;
void f()
{
    int i=index.load(std::memory_order_consume);
    do_something_with(global_data[std::kill_dependency(i)]);
}

```

In real code, you should always use `memory_order_acquire` where you might be tempted to use `memory_order_consume`, and `std::kill_dependency` is thus unnecessary.

Now that I've covered the basics of the memory orderings, it's time to look at the more complex parts of the synchronizes-with relation, which manifest in the form of *release sequences*.

5.3.4 Release sequences and synchronizes-with

Back in section 5.3.1, I mentioned that you could get a synchronizes-with relationship between a store to an atomic variable and a load of that atomic variable from another thread,

even when there's a sequence of read-modify-write operations between the store and the load, provided all the operations are suitably tagged. Now that I've covered the possible memory-ordering "tags," I can elaborate on this. If the store is tagged with `memory_order_release`, `memory_order_acq_rel`, or `memory_order_seq_cst`, and the load is tagged with `memory_order_consume`, `memory_order_acquire`, or `memory_order_seq_cst`, and each operation in the chain loads the value written by the previous operation, then the chain of operations constitutes a *release sequence* and the initial store synchronizes-with (for `memory_order_acquire` or `memory_order_seq_cst`) or is dependency-ordered-before (for `memory_order_consume`) the final load. Any atomic read-modify-write operations in the chain can have *any* memory ordering (even `memory_order_relaxed`).

To see what this means and why it's important, consider an `atomic<int>` being used as a count of the number of items in a shared queue, as in the following listing.

Listing 5.11 Reading values from a queue with atomic operations

```
#include <atomic>
#include <thread>
std::vector<int> queue_data;
std::atomic<int> count;
void populate_queue()
{
    unsigned const number_of_items=20;
    queue_data.clear();
    for(unsigned i=0;i<number_of_items;++i)
    {
        queue_data.push_back(i);
    }

    count.store(number_of_items,std::memory_order_release);    #1
}
void consume_queue_items()
{
    while(true)
    {
        int item_index;
        if((item_index=count.fetch_sub(1,std::memory_order_acquire))<=0)  #2
        {
            wait_for_more_items();    #3
            continue;
        }
        process(queue_data[item_index-1]);    #4
    }
}
int main()
{
    std::thread a(populate_queue);
    std::thread b(consume_queue_items);
    std::thread c(consume_queue_items);
    a.join();
    b.join();
    c.join();
}
```

#1 The initial store

#2 An RMW operation
#3 Wait for more items
#4 Reading queue_data is safe

One way to handle things would be to have the thread that's producing the data store the items in a shared buffer and then do `count.store(number_of_items, memory_order_release)` #1 to let the other threads know that data is available. The threads consuming the queue items might then do `count.fetch_sub(1,memory_order_acquire)` #2 to claim an item from the queue, prior to actually reading the shared buffer #4. Once the count becomes zero, there are no more items, and the thread must wait #3.

If there's one consumer thread, this is fine; the `fetch_sub()` is a read, with `memory_order_acquire` semantics, and the store had `memory_order_release` semantics, so the store synchronizes-with the load and the thread can read the item from the buffer. If there are two threads reading, the second `fetch_sub()` will see the value written by the first and not the value written by the store. Without the rule about the release sequence, this second thread wouldn't have a happens-before relationship with the first thread, and it wouldn't be safe to read the shared buffer unless the first `fetch_sub()` also had `memory_order_release` semantics, which would introduce unnecessary synchronization between the two consumer threads. Without the release sequence rule or `memory_order_release` on the `fetch_sub` operations, there would be nothing to require that the stores to the `queue_data` were visible to the second consumer, and you would have a data race. Thankfully, the first `fetch_sub()` *does* participate in the release sequence, and so the `store()` synchronizes-with the second `fetch_sub()`. There's still no synchronizes-with relationship between the two consumer threads. This is shown in figure 5.7. The dotted lines in figure 5.7 show the release sequence, and the solid lines show the happens-before relationships.

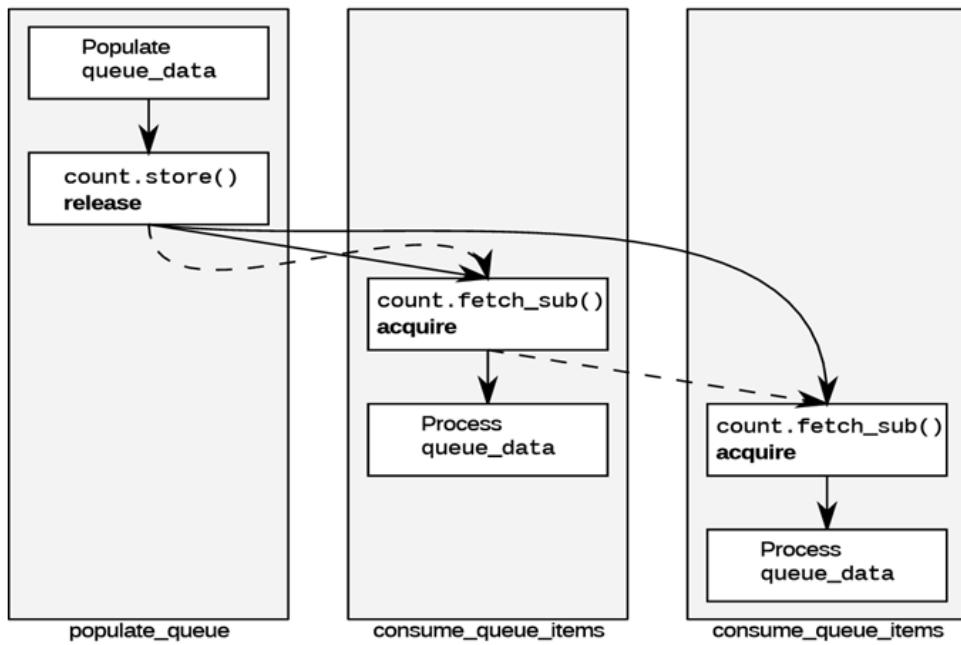


Figure 5.7 The release sequence for the queue operations from listing 5.11

There can be any number of links in the chain, but provided they're all read-modify-write operations such as `fetch_sub()`, the `store()` will still synchronize-with each one that's tagged `memory_order_acquire`. In this example, all the links are the same, and all are acquire operations, but they could be a mix of different operations with different memory-ordering semantics.

Although most of the synchronization relationships come from the memory-ordering semantics applied to operations on atomic variables, it's also possible to introduce additional ordering constraints by using *fences*.

5.3.5 Fences

An atomic operations library wouldn't be complete without a set of fences. These are operations that enforce memory-ordering constraints without modifying any data and are typically combined with atomic operations that use the `memory_order_relaxed` ordering constraints. Fences are global operations and affect the ordering of other atomic operations in the thread that executed the fence. Fences are also commonly called *memory barriers*, and they get their name because they put a line in the code that certain operations can't cross. As you may recall from section 5.3.3, relaxed operations on separate variables can usually be freely reordered by the compiler or the hardware. Fences restrict this freedom and introduce happens-before and synchronizes-with relationships that weren't present before.

Let's start by adding a fence between the two atomic operations on each thread in listing 5.5, as shown in the following listing.

Listing 5.12 Relaxed operations can be ordered with fences

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);      #1
    std::atomic_thread_fence(std::memory_order_release);    #2
    y.store(true,std::memory_order_relaxed);      #3
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));        #4
    std::atomic_thread_fence(std::memory_order_acquire);  #5
    if(x.load(std::memory_order_relaxed))            #6
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);   #7
}
```

The release fence #2 synchronizes-with the acquire fence #5, because the load from `y` at #4 reads the value stored at #3. This means that the store to `x` at #1 happens-before the load from `x` at #6, so the value read must be `true` and the assert at #7 won't fire. This is in contrast to the original case without the fences where the store to and load from `x` weren't ordered, and so the assert could fire. Note that both fences are necessary: you need a release in one thread and an acquire in another to get a synchronizes-with relationship.

In this case, the release fence #2 has the same effect as if the store to `y` #3 was tagged with `memory_order_release` rather than `memory_order_relaxed`. Likewise, the acquire fence #5 makes it as if the load from `y` #4 was tagged with `memory_order_acquire`. This is the general idea with fences: if an acquire operation sees the result of a store that takes place after a release fence, the fence synchronizes-with that acquire operation; and if a load that takes place before an acquire fence sees the result of a release operation, the release operation synchronizes-with the acquire fence. Of course, you can have fences on both sides, as in the example here, in which case if a load that takes place before the acquire fence sees a value written by a store that takes place after the release fence, the release fence synchronizes-with the acquire fence.

Although the fence synchronization depends on the values read or written by operations before or after the fence, it's important to note that the synchronization point is the fence itself. If you take `write_x_then_y` from listing 5.12 and move the write to `x` after the fence as follows, the condition in the assert is no longer guaranteed to be true, even though the write to `x` comes before the write to `y`:

```
void write_x_then_y()
{
    std::atomic_thread_fence(std::memory_order_release);
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_relaxed);
}
```

These two operations are no longer separated by the fence and so are no longer ordered. It's only when the fence comes *between* the store to `x` and the store to `y` that it imposes an ordering. Of course, the presence or absence of a fence doesn't affect any enforced orderings on happens-before relations that exist because of other atomic operations.

This example, and almost every other example so far in this chapter, is built entirely from variables with an atomic type. However, the real benefit to using atomic operations to enforce an ordering is that they can enforce an ordering on nonatomic operations and thus avoid the undefined behavior of a data race, as you saw back in listing 5.2.

5.3.6 Ordering nonatomic operations with atomics

If you replace `x` from listing 5.12 with an ordinary nonatomic `bool` (as in the following listing), the behavior is guaranteed to be the same.

Listing 5.13 Enforcing ordering on nonatomic operations

```
#include <atomic>
#include <thread>
#include <assert.h>
bool x=false;                      #A
std::atomic<bool> y;
std::atomic<int> z;
void write_x_then_y()
{
    x=true;                         #1
    std::atomic_thread_fence(std::memory_order_release);
    y.store(true,std::memory_order_relaxed);      #2
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));      #3
    std::atomic_thread_fence(std::memory_order_acquire);
    if(x)                                     #4
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
```

```

    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);           #5
}

```

```

#A x is now a plain nonatomic variable
#1 Store to x before the fence
#2 Store to y after the fence
#3 Wait until you see the write from #2
#4 This will read the value written by #1
#5 This assert won't fire

```

The fences still provide an enforced ordering of the store to `x` #1 and the store to `y` #2 and the load from `y` #3 and the load from `x` #4, and there's still a happens-before relationship between the store to `x` and the load from `x`, so the assert #5 still won't fire. The store to #2 and load from #3 `y` still have to be atomic; otherwise, there would be a data race on `y`, but the fences enforce an ordering on the operations on `x`, once the reading thread has seen the stored value of `y`. This enforced ordering means that there's no data race on `x`, even though it's modified by one thread and read by another.

It's not just fences that can order nonatomic operations. You saw the ordering effects back in listing 5.10 with a `memory_order_release/memory_order_consume` pair ordering nonatomic accesses to a dynamically allocated object, and many of the examples in this chapter could be rewritten with some of the `memory_order_relaxed` operations replaced with plain nonatomic operations instead.

5.4 Ordering non-atomic operations

Ordering of nonatomic operations through the use of atomic operations is where the sequenced-before part of happens-before becomes so important. If a nonatomic operation is sequenced-before an atomic operation, and that atomic operation happens-before an operation in another thread, the nonatomic operation also happens-before that operation in the other thread. This is where the ordering on the operations on `x` in listing 5.13 comes from and why the example in listing 5.2 works. This is also the basis for the higher-level synchronization facilities in the C++ Standard Library, such as mutexes and condition variables. To see how this works, consider the simple spinlock mutex from listing 5.1.

The `lock()` operation is a loop on `flag.test_and_set()` using `std::memory_order_acquire` ordering, and the `unlock()` is a call to `flag.clear()` with `std::memory_order_release` ordering. When the first thread calls `lock()`, the flag is initially clear, so the first call to `test_and_set()` will set the flag and return `false`, indicating that this thread now has the lock, and terminating the loop. The thread is then free to modify any data protected by the mutex. Any other thread that calls `lock()` at this time will find the flag already set and will be blocked in the `test_and_set()` loop.

When the thread with the lock has finished modifying the protected data, it calls `unlock()`, which calls `flag.clear()` with `std::memory_order_release` semantics. This then

synchronizes-with (see section 5.3.1) a subsequent call to `flag.test_` and `set()` from an invocation of `lock()` on another thread, because this call has `std::memory_order_acquire` semantics. Because the modification of the protected data is necessarily sequenced before the `unlock()` call, this modification happens-before the `unlock()` and thus happens-before the subsequent `lock()` call from the second thread (because of the synchronizes-with relationship between the `unlock()` and the `lock()`) and happens-before any accesses to that data from this second thread once it has acquired the lock.

Although other mutex implementations will have different internal operations, the basic principle is the same: `lock()` is an acquire operation on an internal memory location, and `unlock()` is a release operation on that same memory location.

Each of the synchronization mechanisms described in chapters 2, 3 and 4 will provide ordering guarantees in terms of the *synchronizes-with* relationship. This is what enables you to use them to synchronize your data, and provide ordering guarantees. Below are the synchronization relationships provided by these facilities.

STD:::THREAD

- The completion of the `std::thread` constructor *synchronizes-with* the invocation of the supplied function or callable object on the new thread.
- The completion of a thread *synchronizes-with* the return from a successful call to `join` on the `std::thread` object that owns that thread.

STD:::MUTEX, STD:::TIMED_MUTEX, STD:::RECURSIVE_MUTEX, STD:::RECURSIVE_TIMED_MUTEX

- All calls to `lock` and `unlock`, and successful calls to `try_lock`, `try_lock_for` or `try_lock_until`, on a given `mutex` object form a single total order: the *lock order* of the mutex.
- A call to `unlock` on a given `mutex` object *synchronizes-with* a subsequent call to `lock`, or a subsequent successful call to `try_lock`, `try_lock_for` or `try_lock_until`, on that object in the *lock order* of the mutex.
- Failed calls to `try_lock`, `try_lock_for`, or `try_lock_until` do not participate in any synchronization relationships.

STD:::SHARED_MUTEX, STD:::SHARED_TIMED_MUTEX

- All calls to `lock`, `unlock`, `lock_shared` and `unlock_shared` and successful calls to `try_lock`, `try_lock_for`, `try_lock_until`, `try_lock_shared`, `try_lock_shared_for` or `try_lock_shared_until`, on a given `mutex` object form a single total order: the *lock order* of the mutex.
- A call to `unlock` on a given `mutex` object *synchronizes-with* a subsequent call to `lock` or `shared_lock`, or a successful call to `try_lock`, `try_lock_for`, `try_lock_until`, `try_lock_shared`, `try_lock_shared_for` or `try_lock_shared_until`, on that object in the *lock order* of the mutex.
- Failed calls to `try_lock`, `try_lock_for`, `try_lock_until`, `try_lock_shared`,

`try_lock_shared_for` or `try_lock_shared_until` do not participate in any synchronization relationships.

STD:::PROMISE, STD:::FUTURE AND STD:::SHARED_FUTURE

- The successful completion of a call to `set_value` or `set_exception` on a given `std::promise` object *synchronizes-with* a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns `std::future_status::ready` on a future that shares the same asynchronous state as the promise.
- The destructor of a given `std::promise` object that stores a `std::future_error` exception in the shared asynchronous state associated with the promise *synchronizes-with* a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns `std::future_status::ready` on a future that shares the same asynchronous state as the promise.

STD:::PACKAGED_TASK, STD:::FUTURE AND STD:::SHARED_FUTURE

- The successful completion of a call to the function call operator of a given `std::packaged_task` object *synchronizes-with* a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns `std::future_status::ready` on a future that shares the same asynchronous state as the packaged task.
- The destructor of a given `std::packaged_task` object that stores a `std::future_error` exception in the shared asynchronous state associated with the packaged task *synchronizes-with* a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns `std::future_status::ready` on a future that shares the same asynchronous state as the packaged task.

STD:::ASYNC, STD:::FUTURE AND STD:::SHARED_FUTURE

- The completion of the thread running a task launched via a call to `std::async` with a policy of `std::launch::async` *synchronizes-with* a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns `std::future_status::ready` on a future that shares the same asynchronous state as the spawned task.
- The completion of a task launched via a call to `std::async` with a policy of `std::launch::deferred` *synchronizes-with* a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns `std::future_status::ready` on a future that shares the same asynchronous state as the promise.

STD:::EXPERIMENTAL::FUTURE, STD:::EXPERIMENTAL::SHARED_FUTURE AND CONTINUATIONS

- The event that causes an asynchronous shared state to become *ready* *synchronizes-with* the invocation of a continuation function scheduled on that shared state.
- The completion of a continuation function *synchronizes-with* the a successful return from a call to `wait` or `get`, or a call to `wait_for` or `wait_until` that returns

`std::future_status::ready` on a future that shares the same asynchronous state as the future returned from the call to `then` that scheduled the continuation, or the invocation of any continuation scheduled on that such a future.

`STD::EXPERIMENTAL::LATCH`

- The invocation of each call to `count_down` or `count_down_and_wait` on a given instance of `std::experimental::latch` *synchronizes-with* the completion of each successful call to `wait` or `count_down_and_wait` on that latch.

`STD::EXPERIMENTAL::BARRIER`

- The invocation of each call to `arrive_and_wait` or `arrive_and_drop` on a given instance of `std::experimental::barrier` *synchronizes-with* the completion of each subsequent successful call to `arrive_and_wait` on that barrier.

`STD::EXPERIMENTAL::FLEX_BARRIER`

- The invocation of each call to `arrive_and_wait` or `arrive_and_drop` on a given instance of `std::experimental::flex_barrier` *synchronizes-with* the completion of each subsequent successful call to `arrive_and_wait` on that barrier.
- The invocation of each call to `arrive_and_wait` or `arrive_and_drop` on a given instance of `std::experimental::flex_barrier` *synchronizes-with* the subsequent invocation of the completion function on that barrier.
- The return from the completion function on a given instance of `std::experimental::flex_barrier` *synchronizes-with* the completion of each call to `arrive_and_wait` on that barrier that was blocked waiting for that barrier when the completion function was invoked.

`STD::CONDITION_VARIABLE AND STD::CONDITION_VARIABLE_ANY`

- Condition variables do not provide any synchronization relationships. They are optimizations over busy-wait loops, and all the synchronization is provided by the operations on the associated mutex.

5.5 Summary

In this chapter I've covered the low-level details of the C++ memory model and the atomic operations that provide the basis for synchronization between threads. This includes the basic atomic types provided by specializations of the `std::atomic<>` class template as well as the generic atomic interface provided by the primary `std::atomic<>` template and the `std::experimental::atomic_shared_ptr<>` template, the operations on these types, and the complex details of the various memory-ordering options.

We've also looked at fences and how they can be paired with operations on atomic types to enforce an ordering. Finally, we've come back to the beginning with a look at how the atomic

operations can be used to enforce an ordering between nonatomic operations on separate threads, and the synchronization relationships provided by the higher-level facilities.

In the next chapter we'll look at using the high-level synchronization facilities alongside atomic operations to design efficient containers for concurrent access, and we'll write algorithms that process data in parallel.

6

Designing lock-based concurrent data structures

This chapter covers

- What it means to design data structures for concurrency
- Guidelines for doing so
- Example implementations of data structures designed for concurrency

In the last chapter we looked at the low-level details of atomic operations and the memory model. In this chapter we'll take a break from the low-level details (although we'll need them for chapter 7) and think about data structures.

The choice of data structure to use for a programming problem can be a key part of the overall solution, and parallel programming problems are no exception. If a data structure is to be accessed from multiple threads, either it must be completely immutable so the data never changes and no synchronization is necessary, or the program must be designed to ensure that changes are correctly synchronized between threads. One option is to use a separate mutex and external locking to protect the data, using the techniques we looked at in chapters 3 and 4, and another is to design the data structure itself for concurrent access.

When designing a data structure for concurrency, you can use the basic building blocks of multithreaded applications from earlier chapters, such as mutexes and condition variables. Indeed, you've already seen a couple of examples showing how to combine these building blocks to write data structures that are safe for concurrent access from multiple threads.

In this chapter we'll start by looking at some general guidelines for designing data structures for concurrency. We'll then take the basic building blocks of locks and condition variables and revisit the design of those basic data structures before moving on to more

complex data structures. In chapter 7 we'll look at how to go right back to basics and use the atomic operations described in chapter 5 to build data structures without locks.

So, without further ado, let's look at what's involved in designing a data structure for concurrency.

6.1 What does it mean to design for concurrency?

At the basic level, designing a data structure for concurrency means that multiple threads can access the data structure concurrently, either performing the same or distinct operations, and each thread will see a self-consistent view of the data structure. No data will be lost or corrupted, all invariants will be upheld, and there'll be no problematic race conditions. Such a data structure is said to be *thread-safe*. In general, a data structure will be safe only for particular types of concurrent access. It may be possible to have multiple threads performing one type of operation on the data structure concurrently, whereas another operation requires exclusive access by a single thread. Alternatively, it may be safe for multiple threads to access a data structure concurrently if they're performing *different* actions, whereas multiple threads performing the *same* action would be problematic.

Truly designing for concurrency means more than that, though: it means providing the *opportunity for concurrency* to threads accessing the data structure. By its very nature, a mutex provides *mutual exclusion*: only one thread can acquire a lock on the mutex at a time. A mutex protects a data structure by explicitly *preventing* true concurrent access to the data it protects.

This is called *serialization*: threads take turns accessing the data protected by the mutex; they must access it serially rather than concurrently. Consequently, you must put careful thought into the design of the data structure to enable true concurrent access. Some data structures have more scope for true concurrency than others, but in all cases the idea is the same: the smaller the protected region, the fewer operations are serialized, and the greater the potential for concurrency.

Before we look at some data structure designs, let's have a quick look at some simple guidelines for what to consider when designing for concurrency.

6.1.1 Guidelines for designing data structures for concurrency

As I just mentioned, you have two aspects to consider when designing data structures for concurrent access: ensuring that the accesses are *safe* and *enabling* genuine concurrent access. I covered the basics of how to make the data structure thread-safe back in chapter 3:

- Ensure that no thread can see a state where the invariants of the data structure have been broken by the actions of another thread.
- Take care to avoid race conditions inherent in the interface to the data structure by providing functions for complete operations rather than for operation steps.
- Pay attention to how the data structure behaves in the presence of exceptions to ensure that the invariants are not broken.
- Minimize the opportunities for deadlock when using the data structure by restricting the

scope of locks and avoiding nested locks where possible.

Before you think about any of these details, it's also important to think about what constraints you wish to put on the users of the data structure; if one thread is accessing the data structure through a particular function, which functions are safe to call from other threads?

This is actually quite a crucial question to consider. Generally constructors and destructors require exclusive access to the data structure, but it's up to the user to ensure that they're not accessed before construction is complete or after destruction has started. If the data structure supports assignment, `swap()`, or copy construction, then as the designer of the data structure, you need to decide whether these operations are safe to call concurrently with other operations or whether they require the user to ensure exclusive access even though the majority of functions for manipulating the data structure may be called from multiple threads concurrently without problem.

The second aspect to consider is that of enabling genuine concurrent access. I can't offer much in the way of guidelines here; instead, here's a list of questions to ask yourself as the data structure designer:

- Can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
- Can different parts of the data structure be protected with different mutexes?
- Do all operations require the same level of protection?
- Can a simple change to the data structure improve the opportunities for concurrency without affecting the operational semantics?

All these questions are guided by a single idea: how can you minimize the amount of serialization that must occur and enable the greatest amount of true concurrency? It's not uncommon for data structures to allow concurrent access from multiple threads that merely read the data structure, whereas a thread that can modify the data structure must have exclusive access. This is supported by using constructs like `std::shared_mutex`. Likewise, as you'll see shortly, it's quite common for a data structure to support concurrent access from threads performing different operations while serializing threads that try to perform the same operation.

The simplest thread-safe data structures typically use mutexes and locks to protect the data. Although there are issues with this, as you saw in chapter 3, it's relatively easy to ensure that only one thread is accessing the data structure at a time. To ease you into the design of thread-safe data structures, we'll stick to looking at such lock-based data structures in this chapter and leave the design of concurrent data structures without locks for chapter 7.

6.2 Lock-based concurrent data structures

The design of lock-based concurrent data structures is all about ensuring that the right mutex is locked when accessing the data and ensuring that the lock is held for a minimum amount of time. This is hard enough when there's just one mutex protecting a data structure.

You need to ensure that data can't be accessed outside the protection of the mutex lock and that there are no race conditions inherent in the interface, as you saw in chapter 3. If you use separate mutexes to protect separate parts of the data structure, these issues are compounded, and there's now also the possibility of deadlock if the operations on the data structure require more than one mutex to be locked. You therefore need to consider the design of a data structure with multiple mutexes even more carefully than the design of a data structure with a single mutex.

In this section you'll apply the guidelines from section 6.1.1 to the design of several simple data structures, using mutexes and locks to protect the data. In each case you'll seek out the opportunities for enabling greater concurrency while ensuring that the data structure remains thread-safe.

Let's start by looking at the stack implementation from chapter 3; it's one of the simplest data structures around, and it uses only a single mutex. Is it really thread-safe? How does it fare from the point of view of achieving true concurrency?

6.2.1 A thread-safe stack using locks

The thread-safe stack from chapter 3 is reproduced in the following listing. The intent is to write a thread-safe data structure akin to `std::stack<>`, which supports pushing data items onto the stack and popping them off again.

Listing 6.1 A class definition for a thread-safe stack

```
#include <exception>
struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));      #1
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();  #2
        std::shared_ptr<T> const res(
```

```

        std::make_shared<T>(std::move(data.top())));    #3
    data.pop();          #4
    return res;
}
void pop(T& value)
{
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();
    value=std::move(data.top());           #5
    data.pop();                      #6
}
bool empty() const
{
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
}
};

```

Let's look at each of the guidelines in turn, and see how they apply here.

First, as you can see, the basic thread safety is provided by protecting each member function with a lock on the mutex, `m`. This ensures that only one thread is actually accessing the data at any one time, so provided each member function maintains the invariants, no thread can see a broken invariant.

Second, there's a potential for a race condition between `empty()` and either of the `pop()` functions, but because the code explicitly checks for the contained stack being empty while holding the lock in `pop()`, this race condition isn't problematic. By returning the popped data item directly as part of the call to `pop()`, you avoid a potential race condition that would be present with separate `top()` and `pop()` member functions such as those in `std::stack<>`.

Next, there are a few potential sources of exceptions. Locking a mutex may throw an exception, but not only is this likely to be exceedingly rare (because it indicates a problem with the mutex or a lack of system resources), it's also the first operation in each member function. Because no data has been modified, this is safe. Unlocking a mutex can't fail, so that's always safe, and the use of `std::lock_guard<>` ensures that the mutex is never left locked.

The call to `data.push()` #1 may throw an exception if either copying/moving the data value throws an exception or not enough memory can be allocated to extend the underlying data structure. Either way, `std::stack<>` guarantees it will be safe, so that's not a problem either.

In the first overload of `pop()`, the code itself might throw an `empty_stack` exception #2, but nothing has been modified, so that's safe. The creation of `res` #3 might throw an exception though for a couple of reasons: the call to `std::make_shared` might throw because it can't allocate memory for the new object and the internal data required for reference counting, or the copy constructor or move constructor of the data item to be returned might throw when copying/moving into the freshly allocated memory. In both cases, the C++ runtime and Standard Library ensure that there are no memory leaks and the new object (if any) is correctly destroyed. Because you *still* haven't modified the underlying stack, you're still

OK. The call to `data.pop()` #4 is guaranteed not to throw, as is the return of the result, so this overload of `pop()` is exception-safe.

The second overload of `pop()` is similar, except this time it's the copy assignment or move assignment operator that can throw #5 rather than the construction of a new object and a `std::shared_ptr` instance. Again, you don't actually modify the data structure until the call to `data.pop()` #6, which is still guaranteed not to throw, so this overload is exception-safe too.

Finally, `empty()` doesn't modify any data, so that's exception-safe.

There are a couple of opportunities for deadlock here, because you call user code while holding a lock: the copy constructor or move constructor #1, #3 and copy assignment or move assignment operator #5 on the contained data items, as well as potentially a user-defined `operator new`. If these functions either call member functions on the stack that the item is being inserted into or removed from or require a lock of any kind and another lock was held when the stack member function was invoked, there's the possibility of deadlock. However, it's sensible to require that users of the stack be responsible for ensuring this: you can't reasonably expect to add an item onto a stack or remove it from a stack without copying it or allocating memory for it.

Because all the member functions use a `std::lock_guard<>` to protect the data, it's safe for any number of threads to call the stack member functions. The only member functions that aren't safe are the constructors and destructors, but this isn't a particular problem; the object can be constructed only once and destroyed only once. Calling member functions on an incompletely constructed object or a partially destructed object is never a good idea whether done concurrently or not. As a consequence, the user must ensure that other threads aren't able to access the stack until it's fully constructed and must ensure that all threads have ceased accessing the stack before it's destroyed.

Although it's safe for multiple threads to call the member functions concurrently, because of the use of locks, only one thread is ever actually doing any work in the stack data structure at a time. This *serialization* of threads can potentially limit the performance of an application where there's significant contention on the stack: while a thread is waiting for the lock, it isn't doing any useful work. Also, the stack doesn't provide any means for waiting for an item to be added, so if a thread needs to wait, it must periodically call `empty()` or just call `pop()` and catch the `empty_stack` exceptions. This makes this stack implementation a poor choice if such a scenario is required, because a waiting thread must either consume precious resources checking for data or the user must write external wait and notification code (for example, using condition variables), which might render the internal locking unnecessary and therefore wasteful. The queue from chapter 4 shows a way of incorporating such waiting into the data structure itself using a condition variable inside the data structure, so let's look at that next.

6.2.2 A thread-safe queue using locks and condition variables

The thread-safe queue from chapter 4 is reproduced in listing 6.2. Much like the stack was modeled after `std::stack<>`, this queue is modeled after `std::queue<>`. Again, the interface

differs from that of the standard container adaptor because of the constraints of writing a data structure that's safe for concurrent access from multiple threads.

Listing 6.2 The full class definition for a thread-safe queue using condition variables

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
        data_cond.notify_one();      #1
    }
    void wait_and_pop(T& value)    #2
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(data_queue.front());
        data_queue.pop();
    }
    std::shared_ptr<T> wait_and_pop()    #3
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});    #4
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(data_queue.front());
        data_queue.pop();
        return true;
    }
    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();    #5
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
    bool empty() const
    {
```

```

        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};

```

The structure of the queue implementation shown in listing 6.2 is similar to the stack from listing 6.1, except for the call to `data_cond.notify_one()` in `push()` #1 and the `wait_and_pop()` functions #2, #3. The two overloads of `try_pop()` are almost identical to the `pop()` functions from listing 6.1, except that they don't throw an exception if the queue is empty. Instead, they return either a `bool` value indicating whether a value was retrieved or a `NULL` pointer if no value could be retrieved by the pointer-returning overload #5. This would also have been a valid way of implementing the stack. So, if you exclude the `wait_and_pop()` functions, the analysis you did for the stack applies just as well here.

The new `wait_and_pop()` functions are a solution to the problem of waiting for a queue entry that you saw with the stack; rather than continuously calling `empty()`, the waiting thread can just call `wait_and_pop()` and the data structure will handle the waiting with a condition variable. The call to `data_cond.wait()` won't return until the underlying queue has at least one element, so you don't have to worry about the possibility of an empty queue at this point in the code, and the data is still protected with the lock on the mutex. These functions don't therefore add any new race conditions or possibilities for deadlock, and the invariants will be upheld.

There's a slight twist with regard to exception safety in that if more than one thread is waiting when an entry is pushed onto the queue, only one thread will be woken by the call to `data_cond.notify_one()`. However, if that thread then throws an exception in `wait_and_pop()`, such as when the new `std::shared_ptr<>` is constructed #4, none of the other threads will be woken. If this isn't acceptable, the call is readily replaced with `data_cond.notify_all()`, which will wake all the threads but at the cost of most of them then going back to sleep when they find that the queue is empty after all. A second alternative is to have `wait_and_pop()` call `notify_one()` if an exception is thrown, so that another thread can attempt to retrieve the stored value. A third alternative is to move the `std::shared_ptr<>` initialization to the `push()` call and store `std::shared_ptr<>` instances rather than direct data values. Copying the `std::shared_ptr<>` out of the internal `std::queue<>` then can't throw an exception, so `wait_and_pop()` is safe again. The following listing shows the queue implementation revised with this in mind.

Listing 6.3 A thread-safe queue holding `std::shared_ptr<>` instances

```

template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T> > data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}

```

```

void wait_and_pop(T& value)
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=std::move(*data_queue.front());           #1
    data_queue.pop();
}
bool try_pop(T& value)
{
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return false;
    value=std::move(*data_queue.front());           #2
    data_queue.pop();
    return true;
}
std::shared_ptr<T> wait_and_pop()
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T> res=data_queue.front();     #3
    data_queue.pop();
    return res;
}
std::shared_ptr<T> try_pop()
{
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return std::shared_ptr<T>();
    std::shared_ptr<T> res=data_queue.front();      #4
    data_queue.pop();
    return res;
}
void push(T new_value)
{
    std::shared_ptr<T> data(
        std::make_shared<T>(std::move(new_value)));   #5
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
}
bool empty() const
{
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
};

```

The basic consequences of holding the data by `std::shared_ptr<>` are straightforward: the pop functions that take a reference to a variable to receive the new value now have to dereference the stored pointer #1, #2, and the pop functions that return a `std::shared_ptr<>` instance can just retrieve it from the queue #3, #4 before returning it to the caller.

If the data is held by `std::shared_ptr<>`, there's an additional benefit: the allocation of the new instance can now be done outside the lock in `push()` #5, whereas in listing 6.2 it had to be done while holding the lock in `pop()`. Because memory allocation is typically quite an

expensive operation, this can be very beneficial for the performance of the queue, because it reduces the time the mutex is held, allowing other threads to perform operations on the queue in the meantime.

Just like in the stack example, the use of a mutex to protect the entire data structure limits the concurrency supported by this queue; although multiple threads might be blocked on the queue in various member functions, only one thread can be doing any work at a time. However, part of this restriction comes from the use of `std::queue<>` in the implementation; by using the standard container you now have essentially one data item that's either protected or not. By taking control of the detailed implementation of the data structure, you can provide more fine-grained locking and thus allow a higher level of concurrency.

6.2.3 A thread-safe queue using fine-grained locks and condition variables

In listings 6.2 and 6.3 you have one protected data item (`data_queue`) and thus one mutex. In order to use finer-grained locking, you need to look inside the queue at its constituent parts and associate one mutex with each distinct data item.

The simplest data structure for a queue is a singly linked list, as shown in figure 6.1. The queue contains a *head* pointer, which points to the first item in the list, and each item then points to the next item. Data items are removed from the queue by replacing the head pointer with the pointer to the next item and then returning the data from the old head.

Items are added to the queue at the other end. In order to do this, the queue also contains a *tail* pointer, which refers to the last item in the list. New nodes are added by changing the *next* pointer of the last item to point to the new node and then updating the tail pointer to refer to the new item. When the list is empty, both the head and tail pointers are `NULL`.

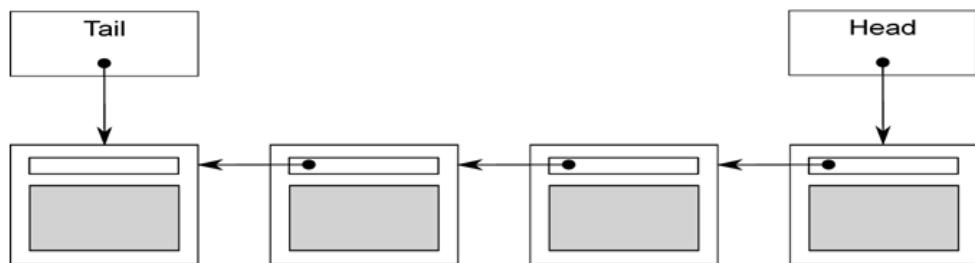


Figure 6.1 A queue represented using a single-linked list

The following listing shows a simple implementation of such a queue based on a cut-down version of the interface to the queue in listing 6.2; you have only one `try_pop()` function and no `wait_and_pop()` because this queue supports only single-threaded use.

Listing 6.4 A simple single-threaded queue implementation

```
template<typename T>
class queue
```

```

{
private:
    struct node
    {
        T data;
        std::unique_ptr<node> next;
        node(T data_):
            data(std::move(data_))
        {}
    };
    std::unique_ptr<node> head;      #1
    node* tail;                      #2
public:
    queue()
    {}
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        if(!head)
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(
            std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head=std::move(head);
        head=std::move(old_head->next);          #3
        if(!head)
            tail=nullptr;
        return res;
    }
    void push(T new_value)
    {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail=p.get();
        if(tail)
        {
            tail->next=std::move(p);    #4
        }
        else
        {
            head=std::move(p);          #5
        }
        tail=new_tail;           #6
    }
};

```

First off, note that listing 6.4 uses `std::unique_ptr<node>` to manage the nodes, because this ensures that they (and the data they refer to) get deleted when they're no longer needed, without having to write an explicit `delete`. This ownership chain is managed from `head`, with `tail` being a raw pointer to the last node, as it needs to refer to a node already owned by a `std::unique_ptr<node>`.

Although this implementation works fine in a single-threaded context, a couple of things will cause you problems if you try to use fine-grained locking in a multithreaded context. Given

that you have two data items (`head` #1 and `tail` #2), you could in principle use two mutexes, one to protect `head` and one to protect `tail`, but there are a couple of problems with that.

The most obvious problem is that `push()` can modify both `head` #5 and `tail` #6, so it would have to lock both mutexes. This isn't too much of a problem, although it's unfortunate, because locking both mutexes would be possible. The critical problem is that both `push()` and `pop()` access the `next` pointer of a node: `push()` updates `tail->next` #4, and `try_pop()` reads `head->next` #3. If there's a single item in the queue, then `head==tail`, so both `head->next` and `tail->next` are the same object, which therefore requires protection. Because you can't tell if it's the same object without reading both `head` and `tail`, you now have to lock the same mutex in both `push()` and `try_pop()`, so you're no better off than before. Is there a way out of this dilemma?

ENABLING CONCURRENCY BY SEPARATING DATA

You can solve this problem by preallocating a dummy node with no data to ensure that there's always at least one node in the queue to separate the node being accessed at the head from that being accessed at the tail. For an empty queue, `head` and `tail` now both point to the dummy node rather than being `NULL`. This is fine, because `try_pop()` doesn't access `head->next` if the queue is empty. If you add a node to the queue (so there's one real node), then `head` and `tail` now point to separate nodes, so there's no race on `head->next` and `tail->next`. The downside is that you have to add an extra level of indirection to store the data by pointer in order to allow the dummy nodes. The following listing shows how the implementation looks now.

Listing 6.5 A simple queue with a dummy node

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;      #1
        std::unique_ptr<node> next;
    };
    std::unique_ptr<node> head;
    node* tail;
public:
    queue():
        head(new node),tail(head.get())    #2
    {}
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        if(head.get()==tail)          #3
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(head->data);    #4
        std::unique_ptr<node> old_head=std::move(head);
        head=old_head->next;
        old_head->next.reset();
        tail=old_head;
    }
};
```

```

        head=std::move(old_head->next);           #5
        return res;                            #6
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));    #7
        std::unique_ptr<node> p(new node);          #8
        tail->data=new_data;                      #9
        node* const new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
};

```

The changes to `try_pop()` are fairly minimal. First, you're comparing `head` against `tail` #3 rather than checking for `NULL`, because the dummy node means that `head` is never `NULL`. Because `head` is a `std::unique_ptr<node>`, you need to call `head.get()` to do the comparison. Second, because the node now stores the data by pointer #1, you can retrieve the pointer directly #4 rather than having to construct a new instance of `T`. The big changes are in `push()`: you must first create a new instance of `T` on the heap and take ownership of it in a `std::shared_ptr<>` #7 (note the use of `std::make_shared` to avoid the overhead of a second memory allocation for the reference count). The new node you create is going to be the new dummy node, so you don't need to supply the `new_value` to the constructor #8. Instead, you set the data on the old dummy node to your newly allocated copy of the `new_value` #9. Finally, in order to have a dummy node, you have to create it in the constructor #2.

By now, I'm sure you're wondering what these changes buy you and how they help with making the queue thread-safe. Well, `push()` now accesses only `tail`, not `head`, which is an improvement. `try_pop()` accesses both `head` and `tail`, but `tail` is needed only for the initial comparison, so the lock is short-lived. The big gain is that the dummy node means `try_pop()` and `push()` are never operating on the same node, so you no longer need an overarching mutex. So, you can have one mutex for `head` and one for `tail`. Where do you put the locks?

You're aiming for the maximum opportunities for concurrency, so you want to hold the locks for the smallest possible length of time. `push()` is easy: the mutex needs to be locked across all accesses to `tail`, which means you lock the mutex after the new node is allocated #8 and before you assign the data to the current `tail` node #9. The lock then needs to be held until the end of the function.

`try_pop()` isn't so easy. First off, you need to lock the mutex on `head` and hold it until you're finished with `head`. In essence, this is the mutex to determine which thread does the popping, so you want to do that first. Once `head` is changed #5, you can unlock the mutex; it doesn't need to be locked when you return the result #6. That leaves the access to `tail` needing a lock on the `tail` mutex. Because you need to access `tail` only once, you can just acquire the mutex for the time it takes to do the read. This is best done by wrapping it in a function. In fact, because the code that needs the `head` mutex locked is only a subset of the member, it's clearer to wrap that in a function too. The final code is shown here.

Listing 6.6 A thread-safe queue with fine-grained locking

```
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;
    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }
    std::unique_ptr<node> pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if(head.get()==get_tail())
        {
            return nullptr;
        }
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }
public:
    threadsafe_queue():
        head(new node),tail(head.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail=p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        tail->next=std::move(p);
        tail=new_tail;
    }
};
```

Let's look at this code with a critical eye, thinking about the guidelines listed in section 6.1.1. Before you look for broken invariants, you should be sure what they are:

- `tail->next==nullptr.`
- `tail->data==nullptr.`
- `head==tail` implies an empty list.
- A single element list has `head->next==tail.`
- For each node `x` in the list, where `x!=tail`, `x->data` points to an instance of `T` and `x->next` points to the next node in the list. `x->next==tail` implies `x` is the last node in the list.
- Following the `next` nodes from `head` will eventually yield `tail.`

On its own, `push()` is straightforward: the only modifications to the data structure are protected by `tail_mutex`, and they uphold the invariant because the new tail node is an empty node and `data` and `next` are correctly set for the old tail node, which is now the last real node in the list.

The interesting part is `try_pop()`. It turns out that not only is the lock on `tail_mutex` necessary to protect the read of `tail` itself, but it's also necessary to ensure that you don't get a data race reading the data from the head. If you didn't have that mutex, it would be quite possible for a thread to call `try_pop()` and a thread to call `push()` concurrently, and there'd be no defined ordering on their operations. Even though each member function holds a lock on a mutex, they hold locks on *different* mutexes, and they potentially access the same data; all data in the queue originates from a call to `push()`, after all. Because the threads would be potentially accessing the same data without a defined ordering, this would be a data race, as you saw in chapter 5, and undefined behavior. Thankfully the lock on the `tail_mutex` in `get_tail()` solves everything. Because the call to `get_tail()` locks the same mutex as the call to `push()`, there's a defined order between the two calls. Either the call to `get_tail()` occurs before the call to `push()`, in which case it sees the old value of `tail`, or it occurs after the call to `push()`, in which case it sees the new value of `tail and the new data attached to the previous value of tail.`

It's also important that the call to `get_tail()` occurs inside the lock on `head_mutex`. If it didn't, the call to `pop_head()` could be stuck in between the call to `get_tail()` and the lock on the `head_mutex`, because other threads called `try_pop()` (and thus `pop_head()`) and acquired the lock first, thus preventing your initial thread from making progress:

```
std::unique_ptr<node> pop_head() #A
{
    node* const old_tail=get_tail(); #1
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if(head.get()==old_tail) #2
    {
        return nullptr;
    }
    std::unique_ptr<node> old_head=std::move(head);
    head=std::move(old_head->next); #3
    return old_head;
}
```

#A This is a broken implementation

#1 Get old tail value outside lock on head_mutex

In this *broken* scenario, where the call to `get_tail()` #1 is made outside the scope of the lock, you might find that both `head` and `tail` have changed by the time your initial thread can acquire the lock on `head_mutex`, and not only is the returned tail node no longer the tail, but it's no longer even part of the list. This could then mean that the comparison of `head` to `old_tail` #2 fails, even if `head` really is the last node. Consequently, when you update `head` #3 you may end up moving `head` beyond `tail` and off the end of the list, destroying the data structure. In the correct implementation from listing 6.6, you keep the call to `get_tail()` inside the lock on `head_mutex`. This ensures that no other threads can change `head`, and `tail` only ever moves further away (as new nodes are added in calls to `push()`), which is perfectly safe. `head` can never pass the value returned from `get_tail()`, so the invariants are upheld.

Once `pop_head()` has removed the node from the queue by updating `head`, the mutex is unlocked, and `try_pop()` can extract the data and delete the node if there was one (and return a `NULL` instance of `std::shared_ptr<>` if not), safe in the knowledge that it's the only thread that can access this node.

Next up, the external interface is a subset of that from listing 6.2, so the same analysis applies: there are no race conditions inherent in the interface.

Exceptions are more interesting. Because you've changed the data allocation patterns, the exceptions can now come from different places. The only operations in `try_pop()` that can throw exceptions are the mutex locks, and the data isn't modified until the locks are acquired. Therefore `try_pop()` is exception-safe. On the other hand, `push()` allocates a new instance of `T` on the heap and a new instance of `node`, either of which might throw an exception. However, both of the newly allocated objects are assigned to smart pointers, so they'll be freed if an exception is thrown. Once the lock is acquired, none of the remaining operations in `push()` can throw an exception, so again you're home and dry and `push()` is exception-safe too.

Because you haven't changed the interface, there are no new external opportunities for deadlock. There are no internal opportunities either; the only place that two locks are acquired is in `pop_head()`, which always acquires the `head_mutex` and then the `tail_mutex`, so this will never deadlock.

The remaining question concerns the actual possibilities for concurrency. This data structure actually has considerably more scope for concurrency than that from listing 6.2, because the locks are more fine-grained and *more is done outside the locks*. For example, in `push()`, the new node and new data item are allocated with no locks held. This means that multiple threads can be allocating new nodes and data items concurrently without a problem. Only one thread can add its new node to the list at a time, but the code to do so is only a few simple pointer assignments, so the lock isn't held for much time at all compared to the `std::queue<>`-based implementation where the lock is held around all the memory allocation operations internal to the `std::queue<>`.

Also, `try_pop()` holds the `tail_mutex` for only a short time, to protect a read from `tail`. Consequently, almost the entirety of a call to `try_pop()` can occur concurrently with a call to `push()`. Also, the operations performed while holding the `head_mutex` are also quite minimal;

the expensive delete (in the destructor of the node pointer) is outside the lock. This will increase the number of calls to `try_pop()` that can happen concurrently; only one thread can call `pop_head()` at a time, but multiple threads can then delete their old nodes and return the data safely.

WAITING FOR AN ITEM TO POP

OK, so listing 6.6 provides a thread-safe queue with fine-grained locking, but it supports only `try_pop()` (and only one overload at that). What about the handy `wait_and_pop()` functions back in listing 6.2? Can you implement an identical interface with your fine-grained locking?

Of course, the answer is, yes, but the real question is, how? Modifying `push()` is easy: just add the `data_cond.notify_one()` call at the end of the function, just like in listing 6.2. Actually, it's not quite that simple; you're using fine-grained locking because you want the maximum possible amount of concurrency. If you leave the mutex locked across the call to `notify_one()` (as in listing 6.2), then if the notified thread wakes up before the mutex has been unlocked, it will have to wait for the mutex. On the other hand, if you unlock the mutex *before* you call `notify_one()`, then the mutex is available for the waiting thread to acquire when it wakes up (assuming no other thread locks it first). This is a minor improvement, but it might be important in some cases.

`wait_and_pop()` is more complicated, because you have to decide where to wait, what the predicate is, and which mutex needs to be locked. The condition you're waiting for is "queue not empty," which is represented by `head!=tail`. Written like that, it would require both `head_mutex` and `tail_mutex` to be locked, but you've already decided in listing 6.6 that you only need to lock `tail_mutex` for the read of `tail` and not for the comparison itself, so you can apply the same logic here. If you make the predicate `head!=get_tail()`, you only need to hold the `head_mutex`, so you can use your lock on that for the call to `data_cond.wait()`. Once you've added the wait logic, the implementation is the same as `try_pop()`.

The second overload of `try_pop()` and the corresponding `wait_and_pop()` overload require careful thought. If you just replace the return of the `std::shared_ptr<T>` retrieved from `old_head` with a copy assignment to the `value` parameter, there's a potential exception-safety issue. At this point, the data item has been removed from the queue and the mutex unlocked; all that remains is to return the data to the caller. However, if the copy assignment throws an exception (as it very well might), the data item is lost because it can't be returned to the queue in the same place.

If the actual type `T` used for the template argument has a no-throw move-assignment operator or a no-throw swap operation, you could use that, but you'd really like a general solution that could be used for any type `T`. In this case, you have to move the potential throwing operation inside the locked region, before the node is removed from the list. This means you need an extra overload of `pop_head()` that retrieves the stored value prior to modifying the list.

In comparison, `empty()` is trivial: just lock `head_mutex` and check for `head== get_tail()` (see listing 6.10). The final code for the queue is shown in listings 6.7, 6.8, 6.9, and 6.10.

Listing 6.7 A thread-safe queue with locking and waiting: internals and interface

```
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;
    std::condition_variable data_cond;
public:
    threadsafe_queue():
        head(new node),tail(head.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
    std::shared_ptr<T> try_pop();
    bool try_pop(T& value);
    std::shared_ptr<T> wait_and_pop();
    void wait_and_pop(T& value);
    void push(T new_value);
    bool empty();
};
```

Pushing new nodes onto the queue is fairly straightforward—the implementation (shown in the following listing) is close to that shown previously.

Listing 6.8 A thread-safe queue with locking and waiting: pushing new values

```
template<typename T>
void threadsafe_queue<T>::push(T new_value)
{
    std::shared_ptr<T> new_data(
        std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        node* const new_tail=p.get();
        tail->next=std::move(p);
        tail=new_tail;
    }
    data_cond.notify_one();
}
```

As already mentioned, the complexity is all in the *pop* side, which makes use of a series of helper functions to simplify matters. The next listing shows the implementation of *wait_and_pop()* and the associated helper functions.

Listing 6.9 A thread-safe queue with locking and waiting: *wait_and_pop()*

```
template<typename T>
class threadsafe_queue
{
private:
    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }
    std::unique_ptr<node> pop_head()      #1
    {
        std::unique_ptr<node> old_head=std::move(head);
        head=std::move(old_head->next);
        return old_head;
    }
    std::unique_lock<std::mutex> wait_for_data()      #2
    {
        std::unique_lock<std::mutex> head_lock(head_mutex);
        data_cond.wait(head_lock,[&]{return head.get()!=get_tail();});
        return std::move(head_lock);                      #3
    }
    std::unique_ptr<node> wait_pop_head()
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());      #4
        return pop_head();
    }
    std::unique_ptr<node> wait_pop_head(T& value)
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());      #5
        value=std::move(*head->data);
        return pop_head();
    }
public:
    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_ptr<node> const old_head=wait_pop_head();
        return old_head->data;
    }
    void wait_and_pop(T& value)
    {
        std::unique_ptr<node> const old_head=wait_pop_head(value);
    }
};
```

The implementation of the pop side shown in listing 6.9 has several little helper functions to simplify the code and reduce duplication, such as *pop_head()* #1, which modifies the list to remove the head item, and *wait_for_data()* #2, which waits for the queue to have some data to pop. *wait_for_data()* is particularly noteworthy, because not only does it wait on the condition variable using a lambda function for the predicate, but it also returns the

lock instance to the caller #3. This is to ensure that the same lock is held while the data is modified by the relevant `wait_pop_head()` overload #4, #5. `pop_head()` is also reused by the `try_pop()` code shown in the next listing.

Listing 6.10 A thread-safe queue with locking and waiting: `try_pop()` and `empty()`

```
template<typename T>
class threadsafe_queue
{
private:
    std::unique_ptr<node> try_pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        return pop_head();
    }
    std::unique_ptr<node> try_pop_head(T& value)
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head.get()==get_tail())
        {
            return std::unique_ptr<node>();
        }
        value=std::move(*head->data);
        return pop_head();
    }
public:
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_head=try_pop_head();
        return old_head?old_head->data:std::shared_ptr<T>();
    }
    bool try_pop(T& value)
    {
        std::unique_ptr<node> const old_head=try_pop_head(value);
        return old_head;
    }
    bool empty()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        return (head.get()==get_tail());
    }
};
```

This queue implementation will serve as the basis for the lock-free queue covered in chapter 7. It's an *unbounded* queue; threads can continue to push new values onto the queue as long as there's available memory, even if no values are removed. The alternative to an unbounded queue is a *bounded* queue, in which the maximum length of the queue is fixed when the queue is created. Once a bounded queue is full, attempts to push further elements onto the queue will either fail or block, until an element has been popped from the queue to make room. Bounded queues can be useful for ensuring an even spread of work when dividing

work between threads based on tasks to be performed (see chapter 8). This prevents the thread(s) populating the queue from running too far ahead of the thread(s) reading items from the queue.

The unbounded queue implementation shown here can easily be extended to limit the length of the queue by waiting on the condition variable in `push()`. Rather than waiting for the queue to have items (as is done in `pop()`), you need to wait for the queue to have fewer than the maximum number of items. Further discussion of bounded queues is outside the scope of this book; for now let's move beyond queues and on to more complex data structures.

6.3 Designing more complex lock-based data structures

Stacks and queues are simple: the interface is exceedingly limited, and they're very tightly focused on a specific purpose. Not all data structures are that simple; most data structures support a variety of operations. In principle, this can then lead to greater opportunities for concurrency, but it also makes the task of protecting the data that much harder because the multiple access patterns need to be taken into account. The precise nature of the various operations that can be performed is important when designing such data structures for concurrent access.

To see some of the issues involved, let's look at the design of a lookup table.

6.3.1 Writing a thread-safe lookup table using locks

A lookup table or dictionary associates values of one type (the key type) with values of either the same or a different type (the mapped type). In general, the intention behind such a structure is to allow code to query the data associated with a given key. In the C++ Standard Library, this facility is provided by the associative containers: `std::map<>`, `std::multimap<>`, `std::unordered_map<>`, and `std::unordered_multimap<>`.

A lookup table has a different usage pattern than a stack or a queue. Whereas almost every operation on a stack or a queue modifies it in some way, either to add an element or remove one, a lookup table might be modified rarely. The simple DNS cache in listing 3.13 is one example of such a scenario, which features a greatly reduced interface compared to `std::map<>`. As you saw with the stack and queue, the interfaces of the standard containers aren't suitable when the data structure is to be accessed from multiple threads concurrently, because there are inherent race conditions in the interface design, so they need to be cut down and revised.

The biggest problem with the `std::map<>` interface from a concurrency perspective is the iterators. Although it's possible to have an iterator that provides safe access into a container even when other threads can access (and modify) the container, this is a tricky proposition. Correctly handling iterators requires you to deal with issues such as another thread deleting the element that the iterator is referring to, which can get rather involved. For the first cut at a thread-safe lookup table interface, you'll skip the iterators. Given that the interface to `std::map<>` (and the other associative containers in the standard library) is so heavily

iterator-based, it's probably worth setting them aside and designing the interface from the ground up.

There are only a few basic operations on a lookup table:

- Add a new key/value pair.
- Change the value associated with a given key.
- Remove a key and its associated value.
- Obtain the value associated with a given key if any.

There are also a few container-wide operations that might be useful, such as a check on whether the container is empty, a snapshot of the complete list of keys, or a snapshot of the complete set of key/value pairs.

If you stick to the simple thread-safety guidelines such as not returning references and put a simple mutex lock around the entirety of each member function, all of these are safe; they either come before some modification from another thread or come after it. The biggest potential for a race condition is when a new key/value pair is being added; if two threads add a new value, only one will be first, and the second will therefore fail. One possibility is to combine add and change into a single member function, as you did for the DNS cache in listing 3.13.

The only other interesting point from an interface perspective is the *if any* part of obtaining an associated value. One option is to allow the user to provide a "default" result that's returned in the case when the key isn't present:

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

In this case, a default-constructed instance of `mapped_type` could be used if the `default_value` wasn't explicitly provided. This could also be extended to return a `std::pair<mapped_type, bool>` instead of just an instance of `mapped_type`, where the `bool` indicates whether the value was present. Another option is to return a smart pointer referring to the value; if the pointer value is `NULL`, there was no value to return.

As already mentioned, once the interface has been decided, then (assuming no interface race conditions) the thread safety could be guaranteed by using a single mutex and a simple lock around every member function to protect the underlying data structure. However, this would squander the possibilities for concurrency provided by the separate functions for reading the data structure and modifying it. One option is to use a mutex that supports multiple reader threads or a single writer thread, such as the `std::shared_mutex` used in listing 3.13. Although this would indeed improve the possibilities for concurrent access, only one thread could modify the data structure at a time. Ideally, you'd like to do better than that.

DESIGNING A MAP DATA STRUCTURE FOR FINE-GRAINED LOCKING

As with the queue discussed in section 6.2.3, in order to permit fine-grained locking you need to look carefully at the details of the data structure rather than just wrapping a preexisting container such as `std::map<>`. There are three common ways of implementing an associative container like your lookup table:

- A binary tree, such as a red-black tree
- A sorted array
- A hash table

A binary tree doesn't provide much scope for extending the opportunities for concurrency; every lookup or modification has to start by accessing the root node, which therefore has to be locked. Although this lock can be released as the accessing thread moves down the tree, this isn't much better than a single lock across the whole data structure.

A sorted array is even worse, because you can't tell in advance where in the array a given data value is going to be, so you need a single lock for the whole array.

That leaves the hash table. Assuming a fixed number of buckets, which bucket a key belongs to is purely a property of the key and its hash function. This means you can safely have a separate lock per bucket. If you again use a mutex that supports multiple readers or a single writer, you increase the opportunities for concurrency N -fold, where N is the number of buckets. The downside is that you need a good hash function for the key. The C++ Standard Library provides the `std::hash<>` template, which you can use for this purpose. It's already specialized for the fundamental types such as `int` and common library types such as `std::string`, and the user can easily specialize it for other key types. If you follow the lead of the standard unordered containers and take the type of the function object to use for doing the hashing as a template parameter, the user can choose whether to specialize `std::hash<>` for their key type or provide a separate hash function.

So, let's look at some code. What might the implementation of a thread-safe lookup table look like? One possibility is shown here.

Listing 6.11 A thread-safe lookup table

```
template<typename Key,typename Value,typename Hash=std::hash<Key> >
class threadsafe_lookup_table
{
private:
    class bucket_type
    {
private:
        typedef std::pair<Key,Value> bucket_value;
        typedef std::list<bucket_value> bucket_data;
        typedef typename bucket_data::iterator bucket_iterator;
        bucket_data data;
        mutable std::shared_mutex mutex;      #1

        bucket_iterator find_entry_for(Key const& key) const  #2
        {
            return std::find_if(data.begin(),data.end(),
                [&](bucket_value const& item)
                {return item.first==key;});
        }
public:
    Value value_for(Key const& key,Value const& default_value) const
    {
        std::shared_lock<std::shared_mutex> lock(mutex);      #3
        bucket_iterator const found_entry=find_entry_for(key);
```

```

        return (found_entry==data.end())?
            default_value:found_entry->second;
    }
    void add_or_update_mapping(Key const& key,Value const& value)
    {
        std::unique_lock<std::shared_mutex> lock(mutex);      #4
        bucket_iterator const found_entry=find_entry_for(key);
        if(found_entry==data.end())
        {
            data.push_back(bucket_value(key,value));
        }
        else
        {
            found_entry->second=value;
        }
    }
    void remove_mapping(Key const& key)
    {
        std::unique_lock<std::shared_mutex> lock(mutex);      #5
        bucket_iterator const found_entry=find_entry_for(key);
        if(found_entry!=data.end())
        {
            data.erase(found_entry);
        }
    }
};

std::vector<std::unique_ptr<bucket_type> > buckets;      #6
Hash hasher;
bucket_type& get_bucket(Key const& key) const      #7
{
    std::size_t const bucket_index=hasher(key)%buckets.size();
    return *buckets[bucket_index];
}
public:
    typedef Key key_type;
    typedef Value mapped_type;
    typedef Hash hash_type;
    threadsafe_lookup_table(
        unsigned num_buckets=19,Hash const& hasher_=Hash()):
        buckets(num_buckets),hasher(hasher_)
    {
        for(unsigned i=0;i<num_buckets;++i)
        {
            buckets[i].reset(new bucket_type);
        }
    }
    threadsafe_lookup_table(threadsafe_lookup_table const& other)=delete;
    threadsafe_lookup_table& operator=(
        threadsafe_lookup_table const& other)=delete;
    Value value_for(Key const& key,
                    Value const& default_value=Value()) const
    {
        return get_bucket(key).value_for(key,default_value);      #8
    }
    void add_or_update_mapping(Key const& key,Value const& value)
    {
        get_bucket(key).add_or_update_mapping(key,value);      #9
    }
    void remove_mapping(Key const& key)

```

```

    {
        get_bucket(key).remove_mapping(key);      #10
    }
};

```

This implementation uses a `std::vector<std::unique_ptr<bucket_type>>` #6 to hold the buckets, which allows the number of buckets to be specified in the constructor. The default is 19, which is an arbitrary prime number; hash tables work best with a prime number of buckets. Each bucket is protected with an instance of `std::shared_mutex` #1 to allow many concurrent reads or a single call to either of the modification functions *per bucket*.

Because the number of buckets is fixed, the `get_bucket()` function #7 can be called without any locking #8, #9, #10, and then the bucket mutex can be locked either for shared (read-only) ownership #3 or unique (read/write) ownership #4, #5 as appropriate for each function.

All three functions make use of the `find_entry_for()` member function #2 on the bucket to determine whether the entry is in the bucket. Each bucket contains just a `std::list<>` of key/value pairs, so adding and removing entries is easy.

I've already covered the concurrency angle, and everything is suitably protected with mutex locks, so what about exception safety? `value_for` doesn't modify anything, so that's fine; if it throws an exception, it won't affect the data structure. `remove_mapping` modifies the list with the call to `erase`, but this is guaranteed not to throw, so that's safe. This leaves `add_or_update_mapping`, which might throw in either of the two branches of the `if`. `push_back` is exception-safe and will leave the list in the original state if it throws, so that branch is fine. The only problem is with the assignment in the case where you're replacing an existing value; if the assignment throws, you're relying on it leaving the original unchanged. However, this doesn't affect the data structure as a whole and is entirely a property of the user-supplied type, so you can safely leave it up to the user to handle this.

At the beginning of this section, I mentioned that one nice-to-have feature of such a lookup table would be the option of retrieving a snapshot of the current state into, for example, a `std::map<>`. This would require locking the entire container in order to ensure that a consistent copy of the state is retrieved, which requires locking all the buckets. Because the "normal" operations on the lookup table require a lock on only one bucket at a time, this would be the only operation that requires a lock on all the buckets. Therefore, provided you lock them in the same order every time (for example, increasing bucket index), there'll be no opportunity for deadlock. Such an implementation is shown in the following listing.

Listing 6.12 Obtaining contents of a threadsafe_lookup_table as a std::map<>

```

std::map<Key,Value> threadsafe_lookup_table::get_map() const
{
    std::vector<std::unique_lock<std::shared_mutex>> locks;
    for(unsigned i=0;i<buckets.size();++i)
    {
        locks.push_back(
            std::unique_lock<std::shared_mutex>(buckets[i].mutex));
    }
}

```

```

std::map<Key,Value> res;
for(unsigned i=0;i<buckets.size();++i)
{
    for(bucket_iterator it=buckets[i].data.begin();
        it!=buckets[i].data.end();
        ++it)
    {
        res.insert(*it);
    }
}
return res;
}

```

The lookup table implementation from listing 6.11 increases the opportunity for concurrency of the lookup table as a whole by locking each bucket separately and by using a `std::shared_mutex` to allow reader concurrency on each bucket. But what if you could increase the potential for concurrency on a bucket by even finer-grained locking? In the next section, you'll do just that by using a thread-safe list container with iterator support.

6.3.2 Writing a thread-safe list using locks

A list is one of the most basic data structures, so it should be straightforward to write a thread-safe one, shouldn't it? Well, that depends on what facilities you're after, and you need one that offers iterator support, something I shied away from adding to your map on the basis that it was too complicated. The basic issue with STL-style iterator support is that the iterator must hold some kind of reference into the internal data structure of the container. If the container can be modified from another thread, this reference must somehow remain valid, which essentially requires that the iterator hold a lock on some part of the structure. Given that the lifetime of an STL-style iterator is completely outside the control of the container, this is a bad idea.

The alternative is to provide iteration functions such as `for_each` as part of the container itself. This puts the container squarely in charge of the iteration and locking, but it does fall foul of the deadlock avoidance guidelines from chapter 3. In order for `for_each` to do anything useful, it must call user-supplied code while holding the internal lock. Not only that, but it must also pass a reference to each item to this user-supplied code in order for the user-supplied code to work on this item. You could avoid this by passing a copy of each item to the user-supplied code, but that would be expensive if the data items were large.

So, for now you'll leave it up to the user to ensure that they don't cause deadlock by acquiring locks in the user-supplied operations and don't cause data races by storing the references for access outside the locks. In the case of the list being used by the lookup table, this is perfectly safe, because you know you're not going to do anything naughty.

That leaves you with the question of which operations to supply for your list. If you cast your eyes back on listings 6.11 and 6.12, you can see the sorts of operations you require:

- Add an item to the list.
- Remove an item from the list if it meets a certain condition.
- Find an item in the list that meets a certain condition.

- Update an item that meets a certain condition.
- Copy each item in the list to another container.

For this to be a good general-purpose list container, it would be helpful to add further operations such as a positional insert, but this is unnecessary for your lookup table, so I'll leave it as an exercise for the reader.

The basic idea with fine-grained locking for a linked list is to have one mutex per node. If the list gets big, that's a lot of mutexes! The benefit here is that operations on separate parts of the list are truly concurrent: each operation holds only the locks on the nodes it's actually interested in and unlocks each node as it moves on to the next. The next listing shows an implementation of just such a list.

Listing 6.13 A thread-safe list with iteration support

```
template<typename T>
class threadsafe_list
{
    struct node      #1
    {
        std::mutex m;
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
        node():      #2
            next()
        {}
        node(T const& value):   #3
            data(std::make_shared<T>(value))
        {}
    };
    node head;
public:
    threadsafe_list()
    {}
    ~threadsafe_list()
    {
        remove_if([](node const&){return true;});
    }
    threadsafe_list(threadsafe_list const& other)=delete;
    threadsafe_list& operator=(threadsafe_list const& other)=delete;
    void push_front(T const& value)
    {
        std::unique_ptr<node> new_node(new node(value));  #4
        std::lock_guard<std::mutex> lk(head.m);
        new_node->next=std::move(head.next);    #5
        head.next=std::move(new_node);      #6
    }
    template<typename Function>
    void for_each(Function f)    #7
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);      #8
        while(node* const next=current->next.get())    #9
        {
            std::unique_lock<std::mutex> next_lk(next->m);  #10
            lk.unlock();                                #11
        }
    }
}
```

```

        f(*next->data);          #12
        current=next;
        lk=std::move(next_lk);    #13
    }
}
template<typename Predicate>
std::shared_ptr<T> find_first_if(Predicate p)    #14
{
    node* current=&head;
    std::unique_lock<std::mutex> lk(head.m);
    while(node* const next=current->next.get())
    {
        std::unique_lock<std::mutex> next_lk(next->m);
        lk.unlock();
        if(p(*next->data))    #15
        {
            return next->data;    #16
        }
        current=next;
        lk=std::move(next_lk);
    }
    return std::shared_ptr<T>();
}
template<typename Predicate>
void remove_if(Predicate p)           #17
{
    node* current=&head;
    std::unique_lock<std::mutex> lk(head.m);
    while(node* const next=current->next.get())
    {
        std::unique_lock<std::mutex> next_lk(next->m);
        if(p(*next->data))    #18
        {
            std::unique_ptr<node> old_next=std::move(current->next);
            current->next=std::move(next->next);           #19
            next_lk.unlock();                                #20
        }
        else
        {
            lk.unlock();      #21
            current=next;
            lk=std::move(next_lk);
        }
    }
}
};


```

The `threadsafe_list` from listing 6.13 is a singly linked list, where each entry is a `node` structure #1. A default-constructed `node` is used for the `head` of the list, which starts with a `NULL` `next` pointer #2. New nodes are added with the `push_front()` function; first a new node is constructed #4, which allocates the stored data on the heap #3, while leaving the `next` pointer as `NULL`. You then need to acquire the lock on the mutex for the `head` node in order to get the appropriate `next` value #5 and insert the node at the front of the list by setting `head.next` to point to your new node #6. So far, so good: you only need to lock one mutex in order to add a new item to the list, so there's no risk of deadlock. Also, the slow memory

allocation happens outside the lock, so the lock is only protecting the update of a couple of pointer values that can't fail. On to the iterative functions.

First up, let's look at `for_each()` #7. This operation takes a `Function` of some type to apply to each element in the list; in common with most standard library algorithms, it takes this function by value and will work with either a genuine function or an object of a type with a function call operator. In this case, the function must accept a value of type `T` as the sole parameter. Here's where you do the hand-over-hand locking. To start with, you lock the mutex on the head node #8. It's then safe to obtain the pointer to the next node (using `get()` because you're not taking ownership of the pointer). If that pointer isn't `NULL` #9, you lock the mutex on that node #10 in order to process the data. Once you have the lock on that node, you can release the lock on the previous node #11 and call the specified function #12. Once the function completes, you can update the `current` pointer to the node you just processed and `move` the ownership of the lock from `next_lk` out to `lk` #13. Because `for_each` passes each data item directly to the supplied `Function`, you can use this to update the items if necessary or copy them into another container, or whatever. This is entirely safe if the function is well behaved, because the mutex for the node holding the data item is held across the call.

`find_first_if()` #14 is similar to `for_each()`; the crucial difference is that the supplied `Predicate` must return `true` to indicate a match or `false` to indicate no match #15. Once you have a match, you just return the found data #16 rather than continuing to search. You could do this with `for_each()`, but it would needlessly continue processing the rest of the list even once a match had been found.

`remove_if()` #17 is slightly different, because this function has to actually update the list; you can't use `for_each()` for this. If the `Predicate` returns `true` #18, you remove the node from the list by updating `current->next` #19. Once you've done that, you can release the lock held on the mutex for the next node. The node is deleted when the `std::unique_ptr<node>` you moved it into goes out of scope #20. In this case, you don't update `current` because you need to check the new `next` node. If the `Predicate` returns `false`, you just want to move on as before #21.

So, are there any deadlocks or race conditions with all these mutexes? The answer here is quite definitely *no*, provided that the supplied predicates and functions are well behaved. The iteration is always one way, always starting from the head node, and always locking the next mutex before releasing the current one, so there's no possibility of different lock orders in different threads. The only potential candidate for a race condition is the deletion of the removed node in `remove_if()` #20 because you do this after you've unlocked the mutex (it's undefined behavior to destroy a locked mutex). However, a few moments' thought reveals that this is indeed safe, because you still hold the mutex on the previous node (`current`), so no new thread can try to acquire the lock on the node you're deleting.

What about opportunities for concurrency? The whole point of such fine-grained locking was to improve the possibilities for concurrency over a single mutex, so have you achieved that? Yes, you have: different threads can be working on different nodes in the list at the

same time, whether they're just processing each item with `for_each()`, searching with `find_first_if()`, or removing items with `remove_if()`. But because the mutex for each node must be locked in turn, the threads can't pass each other. If one thread is spending a long time processing a particular node, other threads will have to wait when they reach that particular node.

6.4 Summary

This chapter started by looking at what it means to design a data structure for concurrency and providing some guidelines for doing so. We then worked through several common data structures (stack, queue, hash map, and linked list), looking at how to apply those guidelines to implement them in a way designed for concurrent access, using locks to protect the data and prevent data races. You should now be able to look at the design of your own data structures to see where the opportunities for concurrency lie and where there's potential for race conditions.

In chapter 7 we'll look at ways of avoiding locks entirely, using the low-level atomic operations to provide the necessary ordering constraints, while sticking to the same set of guidelines.

7

Designing lock-free concurrent data structures

This chapter covers

- Implementations of data structures designed for concurrency without using locks
- Techniques for managing memory in lock-free data structures
- Simple guidelines to aid in the writing of lock-free data structures

In the last chapter we looked at general aspects of designing data structures for concurrency, with guidelines for thinking about the design to ensure they're safe. We then examined several common data structures and looked at example implementations that used mutexes and locks to protect the shared data. The first couple of examples used one mutex to protect the entire data structure, but later ones used more than one to protect various smaller parts of the data structure and allow greater levels of concurrency in accesses to the data structure.

Mutexes are powerful mechanisms for ensuring that multiple threads can safely access a data structure without encountering race conditions or broken invariants. It's also relatively straightforward to reason about the behavior of code that uses them: either the code has the lock on the mutex protecting the data or it doesn't. However, it's not all a bed of roses; you saw in chapter 3 how the incorrect use of locks can lead to deadlock, and you've just seen with the lock-based queue and lookup table examples how the granularity of locking can affect the potential for true concurrency. If you can write data structures that are safe for concurrent access without locks, there's the potential to avoid these problems. Such a data structure is called a *lock-free* data structure.

In this chapter we'll look at how the memory-ordering properties of the atomic operations introduced in chapter 5 can be used to build lock-free data structures. **It is vital for the**

understanding of this chapter that you have read and understood all of chapter 5.

You need to take extreme care when designing such data structures, because they're hard to get right, and the conditions that cause the design to fail may occur very rarely. We'll start by looking at what it means for data structures to be lock-free; then we'll move on to the reasons for using them before working through some examples and drawing out some general guidelines.

7.1 Definitions and consequences

Algorithms and data structures that use mutexes, condition variables, and futures to synchronize the data are called *blocking* data structures and algorithms. The application calls library functions that will suspend the execution of a thread until another thread performs an action. Such library calls are termed *blocking* calls because the thread can't progress past this point until the block is removed. Typically, the OS will suspend a blocked thread completely (and allocate its time slices to another thread) until it's *unblocked* by the appropriate action of another thread, whether that's unlocking a mutex, notifying a condition variable, or making a future *ready*.

Data structures and algorithms that don't use blocking library functions are said to be *nonblocking*. Not all such data structures are *lock-free*, though, so let's look at the various types of nonblocking data structures.

7.1.1 Types of nonblocking data structures

Back in chapter 5, we implemented a basic mutex using `std::atomic_flag` as a spin lock. The code is reproduced in the following listing.

Listing 7.1 Implementation of a spin-lock mutex using std::atomic_flag

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
};
```

This code doesn't call any blocking functions; `lock()` just keeps looping until the call to `test_and_set()` returns `false`. This is why it gets the name *spin lock*—the code “spins” around the loop. Anyway, there are no blocking calls, so any code that uses this mutex to protect shared data is consequently *nonblocking*. It's not *lock-free*, though. It's still a mutex

and can still be locked by only one thread at a time. For that reason, knowing something is “non-blocking” is not enough in most circumstances. Instead, you need to know which (if any) of the more specific terms defined below apply. These are:

- Obstruction Free — If all other threads are paused then any given thread will complete its operation in a bounded number of steps
- Lock Free — If multiple threads are operating on a data structure, then after a bounded number of steps **one** of them will complete its operation.
- Wait Free — **Every** thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure.

For the most part, obstruction free algorithms aren't particularly useful — it's not often that all other threads are paused, so this is more useful as a characterization of a failed lock-free implementation. Let's look more at what's involved in these characterizations, starting with *lock-free* so you can see what kinds of data structures are covered.

7.1.2 Lock-free data structures

For a data structure to qualify as lock-free, more than one thread must be able to access the data structure concurrently. They don't have to be able to do the same operations; a lock-free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time. Not only that, but if one of the threads accessing the data structure is suspended by the scheduler midway through its operation, the other threads must still be able to complete their operations without waiting for the suspended thread.

Algorithms that use compare/exchange operations on the data structure often have loops in them. The reason for using a compare/exchange operation is that another thread might have modified the data in the meantime, in which case the code will need to redo part of its operation before trying the compare/exchange again. Such code can still be lock-free if the compare/exchange would eventually succeed if the other threads were suspended. If it wouldn't, you'd essentially have a spin lock, which is nonblocking but not lock-free.

Lock-free algorithms with such loops can result in one thread being subject to *starvation*. If another thread performs operations with the “wrong” timing, the other thread might make progress while the first thread continually has to retry its operation. Data structures that avoid this problem are wait-free as well as lock-free.

7.1.3 Wait-free data structures

A wait-free data structure is a lock-free data structure with the additional property that every thread accessing the data structure can complete its operation within a bounded number of steps, regardless of the behavior of other threads. Algorithms that can involve an unbounded number of retries because of clashes with other threads are thus not wait-free. Most of the examples in this chapter have that property — they have a while loop on a `compare_exchange_weak` or `compare_exchange_strong` operation, with no upper bound on the number of times the loop can run. The scheduling of threads by the OS may thus mean that a

given thread can loop an exceedingly large number of times, while other threads loop very few times. These operations are thus not wait-free.

Writing wait-free data structures correctly is extremely hard. In order to ensure that every thread can complete its operations within a bounded number of steps, you have to ensure that each operation can be performed in a single pass and that the steps performed by one thread don't cause an operation on another thread to fail. This can make the overall algorithms for the various operations considerably more complex.

Given how hard it is to get a lock-free or wait-free data structure right, you need some pretty good reasons to write one; you need to be sure that the benefit outweighs the cost. Let's therefore examine the points that affect the balance.

7.1.4 The pros and cons of lock-free data structures

When it comes down to it, the primary reason for using lock-free data structures is to enable maximum concurrency. With lock-based containers, there's always the potential for one thread to have to block and wait for another to complete its operation before the first thread can proceed; preventing concurrency through mutual exclusion is the entire purpose of a mutex lock. With a lock-free data structure, *some* thread makes progress with every step. With a wait-free data structure, every thread can make forward progress, regardless of what the other threads are doing; there's no need for waiting. This is a desirable property to have but hard to achieve. It's all too easy to end up writing what's essentially a spin lock.

A second reason to use lock-free data structures is robustness. If a thread dies while holding a lock, that data structure is broken forever. But if a thread dies partway through an operation on a lock-free data structure, nothing is lost except that thread's data; other threads can proceed normally.

The flip side here is that if you can't exclude threads from accessing the data structure, then you must be careful to ensure that the invariants are upheld or choose alternative invariants that can be upheld. Also, you must pay attention to the ordering constraints you impose on the operations. To avoid the undefined behavior associated with a data race, you must use atomic operations for the modifications. But that alone isn't enough; you must ensure that changes become visible to other threads in the correct order. All this means that writing thread-safe data structures without using locks is considerably harder than writing them with locks.

Because there aren't any locks, deadlocks are impossible with lock-free data structures, although there is the possibility of live locks instead. A *live lock* occurs when two threads each try to change the data structure, but for each thread the changes made by the other require the operation to be restarted, so both threads loop and try again. Imagine two people trying to go through a narrow gap. If they both go at once, they get stuck, so they have to come out and try again. Unless someone gets there first (either by agreement, by being quicker, or by sheer luck), the cycle will repeat. As in this simple example, live locks are typically short lived because they depend on the exact scheduling of threads. They therefore sap performance rather than cause long-term problems, but they're still something to watch out for. By

definition, wait-free code can't suffer from live lock because there's always an upper limit on the number of steps needed to perform an operation. The flip side here is that the algorithm is likely more complex than the alternative and may require more steps even when no other thread is accessing the data structure.

This brings us to another downside of lock-free and wait-free code: although it can increase the potential for concurrency of operations on a data structure and reduce the time an individual thread spends waiting, it may well *decrease* overall performance. First, the atomic operations used for lock-free code can be much slower than nonatomic operations, and there'll likely be more of them in a lock-free data structure than in the mutex locking code for a lock-based data structure. Not only that, but the hardware must synchronize data between threads that access the same atomic variables. As you'll see in chapter 8, the cache ping-pong associated with multiple threads accessing the same atomic variables can be a significant performance drain. As with everything, it's important to check the relevant performance aspects (whether that's worst-case wait time, average wait time, overall execution time, or something else) both with a lock-based data structure and a lock-free one before committing either way.

Now let's look at some examples.

7.2 Examples of lock-free data structures

In order to demonstrate some of the techniques used in designing lock-free data structures, we'll look at the lock-free implementation of a series of simple data structures. Not only will each example describe the implementation of a useful data structure, but I'll use the examples to highlight particular aspects of lock-free data structure design.

As already mentioned, lock-free data structures rely on the use of atomic operations and the associated memory-ordering guarantees in order to ensure that data becomes visible to other threads in the correct order. Initially, we'll use the default `memory_order_seq_cst` memory ordering for all atomic operations, because that's the easiest to reason about (remember that all `memory_order_seq_cst` operations form a total order). But for later examples we'll look at reducing some of the ordering constraints to `memory_order_acquire`, `memory_order_release`, or even `memory_order_relaxed`. Although none of these examples use mutex locks directly, it's worth bearing in mind that only `std::atomic_flag` is guaranteed not to use locks in the implementation. On some platforms what appears to be lock-free code might actually be using locks internal to the C++ Standard Library implementation (see chapter 5 for more details). On these platforms, a simple lock-based data structure might actually be more appropriate, but there's more to it than that; before choosing an implementation, you must identify your requirements and profile the various options that meet those requirements.

So, back to the beginning with the simplest of data structures: a stack.

7.2.1 Writing a thread-safe stack without locks

The basic premise of a stack is relatively simple: nodes are retrieved in the reverse order to which they were added—last in, first out (LIFO). It's therefore important to ensure that once a value is added to the stack, it can safely be retrieved immediately by another thread, and it's also important to ensure that only one thread returns a given value. The simplest stack is just a linked list; the head pointer identifies the first node (which will be the next to retrieve), and each node then points to the next node in turn.

Under such a scheme, adding a node is relatively simple:

1. Create a new node.
2. Set its `next` pointer to the current `head` node.
3. Set the `head` node to point to it.

This works fine in a single-threaded context, but if other threads are also modifying the stack, it's not enough. Crucially, if two threads are adding nodes, there's a race condition between steps 2 and 3: a second thread could modify the value of `head` between when your thread reads it in step 2 and you update it in step 3. This would then result in the changes made by that other thread being discarded or even worse consequences. Before we look at addressing this race condition, it's also important to note that once `head` has been updated to point to your new node, another thread could read that node. It's therefore vital that your new node is thoroughly prepared *before* `head` is set to point to it; you can't modify the node afterward.

OK, so what can you do about this nasty race condition? The answer is to use an atomic compare/exchange operation at step 3 to ensure that `head` hasn't been modified since you read it in step 2. If it has, you can loop and try again. The following listing shows how you can implement a thread-safe `push()` without locks.

Listing 7.2 Implementing `push()` without locks

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        T data;
        node* next;
        node(T const& data_): #1
            data(data_)
    {};
        std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data); #2
        new_node->next=head.load(); #3
        while(!head.compare_exchange_weak(new_node->next,new_node)); #4
    }
}
```

```
};
```

This code neatly matches the three-point plan from above: create a new node #2, set the node's `next` pointer to the current head #3, and set the `head` pointer to the new node #4. By populating the data in the node structure itself from the node constructor #1, you've ensured that the node is ready to roll as soon as it's constructed, so that's the easy problem down. Then you use `compare_exchange_weak()` to ensure that the `head` pointer still has the same value as you stored in `new_node->next` #3, and you set it to `new_node` if so. This bit of code also uses a nifty part of the compare/exchange functionality: if it returns `false` to indicate that the comparison failed (for example, because `head` was modified by another thread), the value supplied as the first parameter (`new_node->next`) is updated to the current value of `head`. You therefore don't have to reload `head` each time through the loop, because the compiler does that for you. Also, because you're just looping directly on failure, you can use `compare_exchange_weak`, which can result in more optimal code than `compare_exchange_strong` on some architectures (see chapter 5).

So, you might not have a `pop()` operation yet, but you can quickly check `push()` for compliance with the guidelines. The only place that can throw an exception is the construction of the new node #1, but this will clean up after itself, and the list hasn't been modified yet, so that's perfectly safe. Because you build the data to be stored as part of the node, and you use `compare_exchange_weak()` to update the `head` pointer, there are no problematic race conditions here. Once the compare/exchange succeeds, the node is on the list and ready for the taking. There are no locks, so there's no possibility of deadlock, and your `push()` function passes with flying colors.

Of course, now that you have a means of adding data to the stack, you need a way of getting it off again. On the face of it, this is quite simple:

1. Read the current value of `head`.
2. Read `head->next`.
3. Set `head` to `head->next`.
4. Return the data from the retrieved node.
5. Delete the retrieved node.

However, in the presence of multiple threads, this isn't so simple. If there are two threads removing items from the stack, they both might read the same value of `head` at step 1. If one thread then proceeds all the way through to step 5 before the other gets to step 2, the second thread will be dereferencing a dangling pointer. This is one of the biggest issues in writing lock-free code, so for now you'll just leave out step 5 and leak the nodes.

This doesn't resolve all the problems, though. There's another problem: if two threads read the same value of `head`, they'll return the same node. This violates the intent of the stack data structure, so you need to avoid this. You can resolve this the same way you resolved the race in `push()`: use compare/exchange to update `head`. If the compare/exchange fails, either a new node has been pushed on or another thread just popped the node you were trying to pop.

Either way, you need to return to step 1 (although the compare/exchange call rereads `head` for you).

Once the compare/exchange call succeeds, you know you're the only thread that's popping the given node off the stack, so you can safely execute step 4. Here's a first cut at `pop()`:

```
template<typename T>
class lock_free_stack
{
public:
    void pop(T& result)
    {
        node* old_head=head.load();
        while(!head.compare_exchange_weak(old_head,old_head->next));
        result=old_head->data;
    }
};
```

Although this is nice and succinct, there are still a couple of problems aside from the leaking node. First, it doesn't work on an empty list: if `head` is a null pointer, it will cause undefined behavior as it tries to read the `next` pointer. This is easily fixed by checking for `nullptr` in the `while` loop and either throwing an exception on an empty stack or returning a `bool` to indicate success or failure.

The second problem is an exception-safety issue. When we first introduced the thread-safe stack back in chapter 3, you saw how just returning the object by value left you with an exception safety issue: if an exception is thrown when copying the return value, the value is lost. In that case, passing in a reference to the `result` was an acceptable solution because you could ensure that the stack was left unchanged if an exception was thrown. Unfortunately, here you don't have that luxury; you can only safely copy the data once you know you're the only thread returning the node, *which means the node has already been removed from the queue*. Consequently, passing in the target for the return value by reference is no longer an advantage: you might as well just return by value. If you want to return the value safely, you have to use the other option from chapter 3: return a (smart) pointer to the data value.

If you return a smart pointer, you can just return `nullptr` to indicate that there's no value to return, but this requires that the data be allocated on the heap. If you do the heap allocation as part of the `pop()`, you're *still* no better off, because the heap allocation might throw an exception. Instead, you can allocate the memory when you `push()` the data onto the stack—you have to allocate memory for the `node` anyway. Returning a `std::shared_ptr<>` won't throw an exception, so `pop()` is now safe. Putting all this together gives the following listing.

Listing 7.3 A lock-free stack that leaks nodes

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
```

```

        std::shared_ptr<T> data;           #1
        node* next;
        node(T const& data_):
            data(std::make_shared<T>(data_))    #2
        {}
    };
    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data);
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=head.load();          #3
        while(old_head &&                  #3
              !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();      #4
    }
};

```

#1 Data is now held by pointer

#2 Create std::shared_ptr for newly allocated T

#3 Check old_head is not a null pointer before you dereference it

The data is held by the pointer now #1, so you have to allocate the data on the heap in the node constructor #2. You also have to check for a null pointer before you dereference `old_head` in the `compare_exchange_weak()` loop #3. Finally, you either return the data associated with your node, if there is one, or a null pointer if not #4. Note that although this is *lock-free*, it's not *wait-free*, because the `while` loops in both `push()` and `pop()` could in theory loop forever if the `compare_exchange_weak()` keeps failing.

If you have a garbage collector picking up after you (like in managed languages such as C# or Java), you're finished; the old node will be collected and recycled once it's no longer being accessed by any threads. However, not many C++ compilers ship with a garbage collector, so you generally have to tidy up after yourself.

7.2.2 Stopping those pesky leaks: managing memory in lock-free data structures

When we first looked at `pop()`, we opted to leak nodes in order to avoid the race condition where one thread deletes a node while another thread still holds a pointer to it that it's just about to dereference. However, leaking memory isn't acceptable in any sensible C++ program, so we have to do something about that. Now it's time to look at the problem and work out a solution.

The basic problem is that you want to free a node, but you can't do so until you're sure there are no other threads that still hold pointers to it. If only one thread ever calls `pop()` on a particular stack instance, you're home free. Nodes are created in calls to `push()`, and `push()` doesn't access the contents of existing nodes, so the only threads that can access a given node are the thread that added that node to the stack, and any threads that call `pop()`.

`push()` doesn't touch the node once it's been added to the stack, so that leaves the threads that call `pop()` — if there's only one of them, then the thread that called `pop()` must be the only thread that can touch the node, and it can safely delete it.

On the other hand, if you need to handle multiple threads calling `pop()` on the same stack instance, you need some way to track when it's safe to delete a node. This essentially means you need to write a special-purpose garbage collector just for nodes. Now, this might sound scary, but although it's certainly tricky, it's not *that* bad: you're only checking for nodes, and you're only checking for nodes accessed from `pop()`. You're not worried about nodes in `push()`, because they're only accessible from one thread until they're on the stack, whereas multiple threads might be accessing the same node in `pop()`.

If there are no threads calling `pop()`, it's perfectly safe to delete all the nodes currently awaiting deletion. Therefore, if you add the nodes to a "to be deleted" list when you've extracted the data, then you can delete them all when there are no threads calling `pop()`. How do you know there aren't any threads calling `pop()`? Simple—count them. If you increment a counter on entry and decrement that counter on exit, it's safe to delete the nodes from the "to be deleted" list when the counter is zero. Of course, it will have to be an atomic counter so it can safely be accessed from multiple threads. The following listing shows the amended `pop()` function, and listing 7.5 shows the supporting functions for such an implementation.

Listing 7.4 Reclaiming nodes when no threads are in `pop()`

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<unsigned> threads_in_pop;    #1
    void try_reclaim(node* old_head);
public:
    std::shared_ptr<T> pop()
    {
        ++threads_in_pop;                  #2
        node* old_head=head.load();
        while(old_head &&
              !head.compare_exchange_weak(old_head,old_head->next));
        std::shared_ptr<T> res;
        if(old_head)
        {
            res.swap(old_head->data);    #3
        }
        try_reclaim(old_head);           #4
        return res;
    }
};

#1 Atomic variable
#2 Increase counter before doing anything else
#3 Reclaim deleted nodes if you can
#4 Extract data from node rather than copying pointer
```

The atomic variable `threads_in_pop` #1 is used to count the threads currently trying to pop an item off the stack. It's incremented at the start of `pop()` #2 and decremented inside `try_reclaim()`, which is called once the node has been removed #4. Because you're going to potentially delay the deletion of the node itself, you can use `swap()` to remove the data from the node #3 rather than just copying the pointer, so that the data will be deleted automatically when you no longer need it rather than it being kept alive because there's still a reference in a not-yet-deleted node. The next listing shows what goes into `try_reclaim()`.

Listing 7.5 The reference-counted reclamation machinery

```
template<typename T>
class lock_free_stack
{
private:
    std::atomic<node*> to_be_deleted;
    static void delete_nodes(node* nodes)
    {
        while(nodes)
        {
            node* next=nodes->next;
            delete nodes;
            nodes=next;
        }
    }
    void try_reclaim(node* old_head)
    {
        if(threads_in_pop==1)      #1
        {
            node* nodes_to_delete=to_be_deleted.exchange(nullptr); #2
            if(!--threads_in_pop)                                #3
            {
                delete_nodes(nodes_to_delete);                   #4
            }
            else if(nodes_to_delete)                            #5
            {
                chain_pending_nodes(nodes_to_delete);          #6
            }
            delete old_head;                                  #7
        }
        else
        {
            chain_pending_node(old_head);                  #8
            --threads_in_pop;
        }
    }
    void chain_pending_nodes(node* nodes)
    {
        node* last=nodes;
        while(node* const next=last->next)                 #9
        {
            last=next;
        }
        chain_pending_nodes(nodes,last);
    }
    void chain_pending_nodes(node* first,node* last)
    {
```

```

        last->next=to_be_deleted;                      #10
        while(!to_be_deleted.compare_exchange_weak(
            last->next,first));                     #11
    }
    void chain_pending_node(node* n)
    {
        chain_pending_nodes(n,n);      #12
    }
};

#2 Claim list of to-be-deleted nodes
#3 Are you the only thread in pop()?
#9 Follow the next pointer chain to the end
#11 Loop to guarantee that last->next is correct

```

If the count of `threads_in_pop` is 1 when you're trying to reclaim the node #1, you're the only thread currently in `pop()`, which means it's safe to delete the node you just removed #7, and it *may* also be safe to delete the pending nodes. If the count is *not* 1, it's not safe to delete any nodes, so you have to add the node to the pending list #8.

Assume for a moment that `threads_in_pop` is 1. You now need to try to reclaim the pending nodes; if you don't, they'll stay pending until you destroy the stack. To do this, you first claim the list for yourself with an atomic exchange operation #2 and then decrement the count of `threads_in_pop` #3. If the count is zero after the decrement, you know that no other thread can be accessing this list of pending nodes. There may be new pending nodes, but you're not bothered about them for now, as long as it's safe to reclaim your list. You can then just call `delete_nodes` to iterate down the list and delete them #4.

If the count is not zero after the decrement, it's not safe to reclaim the nodes, so if there are any #5, you must chain them back onto the list of nodes pending deletion #6. This can happen if there are multiple threads accessing the data structure concurrently. Other threads might have called `pop()` in between the first test of `threads_in_pop` #1 and the "claiming" of the list #2, potentially adding new nodes to the list that are still being accessed by one or more of those other threads. In figure 7.1, thread C adds node Y to the `to_be_deleted` list, even though thread B is still referencing it as `old_head`, and will thus try and read its `next` pointer. Thread A can't therefore delete the nodes without potentially causing undefined behavior for thread B.

To chain the nodes that are pending deletion onto the pending list, you reuse the `next` pointer from the nodes to link them together. In the case of relinking an existing chain back onto the list, you traverse the chain to find the end #9, replace the `next` pointer from the last node with the current `to_be_deleted` pointer #10, and store the first node in the chain as the new `to_be_deleted` pointer #11. You have to use `compare_exchange_weak` in a loop here in order to ensure that you don't leak any nodes that have been added by another thread. This has the benefit of updating the `next` pointer from the end of the chain if it has been changed. Adding a single node onto the list is a special case where the first node in the chain to be added is the same as the last one #12.

This works reasonably well in low-load situations, where there are suitable quiescent points at which no threads are in `pop()`. However, this is potentially a transient situation, which is why you need to test that the `threads_in_pop` count decrements to zero #3 before doing the reclaim and why this test occurs before you delete the just-removed node #7. Deleting a node is potentially a time-consuming operation, and you want the window in which other threads can modify the list to be as small as possible. The longer the time between when the thread first finds `threads_in_pop` to be equal to 1 and the attempt to delete the nodes, the more chance there is that another thread has called `pop()`, and that `threads_in_pop` is no longer equal to 1, thus preventing the nodes from actually being deleted.

In high-load situations, there may never be such a quiescent state, because other threads have entered `pop()` before all the threads initially in `pop()` have left. Under such a scenario, the `to_be_deleted` list would grow without bounds, and you'd be essentially leaking memory again. If there aren't going to be any quiescent periods, you need to find an alternative mechanism for reclaiming the nodes. The key is to identify when no more threads are accessing a particular node so that it can be reclaimed. By far the easiest such mechanism to reason about is the use of hazard pointers.

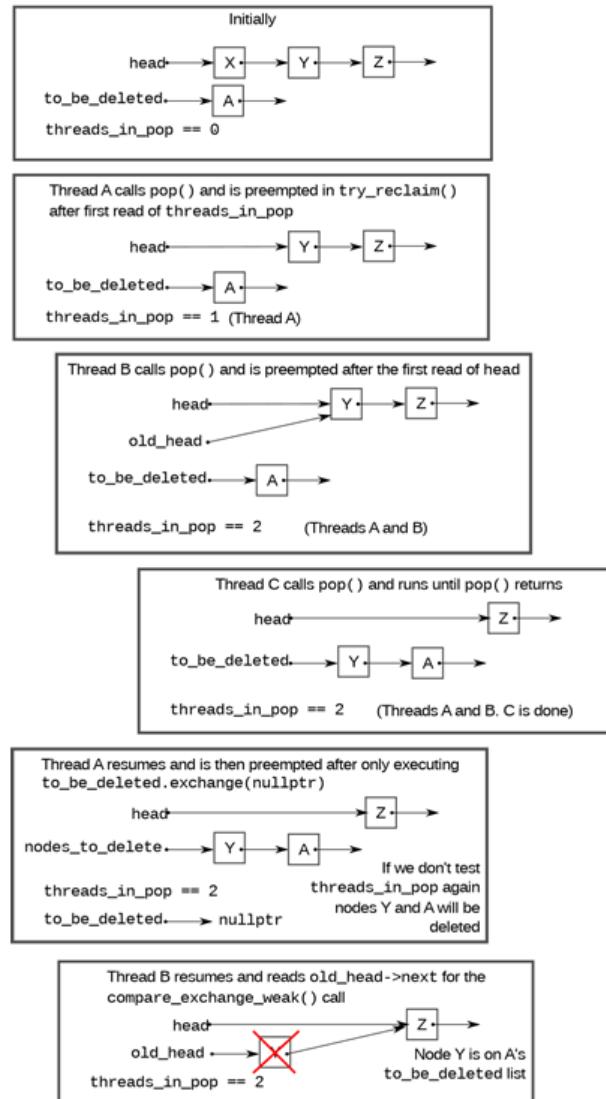


Figure 7.1 Three threads call `pop()` concurrently, showing why you must check `threads_in_pop` after claiming the nodes to be deleted in `try_reclaim()`.

7.2.3 Detecting nodes that can't be reclaimed using hazard pointers

The term *hazard pointers* is a reference to a technique discovered by Maged Michael.¹ They are so called because deleting a node that might still be referenced by other threads is hazardous. If other threads do indeed hold references to that node and proceed to access the node through that reference, you have undefined behavior. The basic idea is that if a thread is going to access an object that another thread might want to delete, it first sets a hazard pointer to reference the object, thus informing the other thread that deleting the object would indeed be hazardous. Once the object is no longer needed, the hazard pointer is cleared. If you've ever watched the Oxford/Cambridge boat race, you've seen a similar mechanism used when starting the race: the cox of either boat can raise their hand to indicate that they aren't ready. While either cox has their hand raised, the umpire may not start the race. If both coxes have their hands down, the race may start, but a cox may raise their hand again if the race hasn't started and they feel the situation has changed.

When a thread wishes to delete an object, it must first check the hazard pointers belonging to the other threads in the system. If none of the hazard pointers reference the object, it can safely be deleted. Otherwise, it must be left until later. Periodically, the list of objects that have been left until later is checked to see if any of them can now be deleted.

Described at such a high level, it sounds relatively straightforward, so how do you do this in C++?

Well, first off you need a location in which to store the pointer to the object you're accessing, the *hazard pointer* itself. This location must be visible to all threads, and you need one of these for each thread that might access the data structure. Allocating them correctly and efficiently can be a challenge, so you'll leave that for later and assume you have a function `get_hazard_pointer_for_current_thread()` that returns a reference to your hazard pointer. You then need to set it when you read a pointer that you intend to dereference—in this case the `head` value from the list:

```
std::shared_ptr<T> pop()
{
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();      #1
    node* temp;
    do
    {
        temp=old_head;
        hp.store(old_head);        #2
        old_head=head.load();
    } while(old_head!=temp);        #3
    // ...
}
```

You have to do this in a `while` loop to ensure that the node hasn't been deleted between the reading of the old head pointer #1 and the setting of the hazard pointer #2. During this

¹ "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes," Maged M. Michael, in *PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (2002), ISBN 1-58113-485-1.

window no other thread knows you're accessing this particular node. Fortunately, if the old head node is going to be deleted, head itself must have changed, so you can check this and keep looping until you know that the head pointer still has the same value you set your hazard pointer to #3. Using hazard pointers like this relies on the fact that it's safe to use the value of a pointer after the object it references has been deleted. This is technically undefined behavior if you are using the default implementation of `new` and `delete`, so either you need to ensure that your implementation permits it, or you need to use a custom allocator that permits such usage.

Now that you've set your hazard pointer, you can proceed with the rest of `pop()`, safe in the knowledge that no other thread will delete the nodes from under you. Well, almost: every time you reload `old_head`, you need to update the hazard pointer before you dereference the freshly read pointer value. Once you've extracted a node from the list, you can clear your hazard pointer. If there are no other hazard pointers referencing your node, you can safely delete it; otherwise, you have to add it to a list of nodes to be deleted later. The following listing shows a full implementation of `pop()` using such a scheme.

Listing 7.6 An implementation of `pop()` using hazard pointers

```
std::shared_ptr<T> pop()
{
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();
    do
    {
        node* temp;
        do                      #1
        {
            temp=old_head;
            hp.store(old_head);
            old_head=head.load();
        } while(old_head!=temp);
    }
    while(old_head &&
          !head.compare_exchange_strong(old_head,old_head->next));
    hp.store(nullptr);                      #2
    std::shared_ptr<T> res;
    if(old_head)
    {
        res.swap(old_head->data);
        if(outstanding_hazard_pointers_for(old_head))           #3
        {
            reclaim_later(old_head);      #4
        }
        else
        {
            delete old_head;           #5
        }
        delete_nodes_with_no_hazards();   #6
    }
    return res;
}
```

```
#1 Loop until you've set the hazard pointer to head  
#2 Clear hazard pointer once you're finished  
#3 Check for hazard pointers referencing a node before you delete it
```

First off, you've moved the loop that sets the hazard pointer *inside* the outer loop for reloading `old_head` if the compare/exchange fails #1. You're using `compare_exchange_strong()` here because you're actually doing work inside the `while` loop: a spurious failure on `compare_exchange_weak()` would result in resetting the hazard pointer unnecessarily. This ensures that the hazard pointer is correctly set before you dereference `old_head`. Once you've claimed the node as yours, you can clear your hazard pointer #2. If you did get a node, you need to check the hazard pointers belonging to other threads to see if they reference it #3. If so, you can't delete it just yet, so you must put it on a list to be reclaimed later #4; otherwise, you can delete it right away #5. Finally, you put in a call to check for any nodes for which you had to call `reclaim_later()`. If there are no longer any hazard pointers referencing those nodes, you can safely delete them #6. Any nodes for which there are still outstanding hazard pointers will be left for the next thread that calls `pop()`.

Of course, there's still a lot of detail hidden in these new functions—`get_hazard_pointer_for_current_thread()`, `reclaim_later()`, `outstanding_hazard_pointers_for()`, and `delete_nodes_with_no_hazards()`—so let's draw back the curtain and look at how they work.

The exact scheme for allocating hazard pointer instances to threads used by `get_hazard_pointer_for_current_thread()` doesn't really matter for the program logic (although it can affect the efficiency, as you'll see later). So for now you'll go with a simple structure: a fixed-size array of pairs of thread IDs and pointers. `get_hazard_pointer_for_current_thread()` then searches through the array to find the first free slot and sets the ID entry of that slot to the ID of the current thread. When the thread exits, the slot is freed by resetting the ID entry to a default-constructed `std::thread::id()`. This is shown in the following listing.

Listing 7.7 A simple implementation of `get_hazard_pointer_for_current_thread()`

```
unsigned const max_hazard_pointers=100;  
struct hazard_pointer  
{  
    std::atomic<std::thread::id> id;  
    std::atomic<void*> pointer;  
};  
hazard_pointer hazard_pointers[max_hazard_pointers];  
class hp_owner  
{  
    hazard_pointer* hp;  
  
public:  
    hp_owner(hp_owner const&)=delete;  
    hp_owner operator=(hp_owner const&)=delete;  
    hp_owner():  
        hp(nullptr)  
    {
```

```

for(unsigned i=0;i<max_hazard_pointers;++i)
{
    std::thread::id old_id;
    if(hazard_pointers[i].id.compare_exchange_strong(      #A
        old_id,std::this_thread::get_id()))
    {
        hp=&hazard_pointers[i];
        break;
    }
}
if(!hp)                                              #1
{
    throw std::runtime_error("No hazard pointers available");
}
std::atomic<void*>& get_pointer()
{
    return hp->pointer;
}
~hp_owner()                                         #2
{
    hp->pointer.store(nullptr);
    hp->id.store(std::thread::id());
}
};

std::atomic<void*>& get_hazard_pointer_for_current_thread()   #3
{
    thread_local static hp_owner hazard;                      #4
    return hazard.get_pointer();                            #5
}

```

#A Try to claim ownership of a hazard pointer

#4 Each thread has its own hazard pointer

The actual implementation of `get_hazard_pointer_for_current_thread()` itself is deceptively simple #3: it has a `thread_local` variable of type `hp_owner` #4 that stores the hazard pointer for the current thread. It then just returns the pointer from that object #5. This works as follows: The first time *each thread* calls this function, a new instance of `hp_owner` is created. The constructor for this new instance #1 then searches through the table of owner/pointer pairs looking for an entry without an owner. It uses `compare_exchange_strong()` to check for an entry without an owner and claim it in one go #2. If the `compare_exchange_strong()` fails, another thread owns that entry, so you move on to the next. If the exchange succeeds, you've successfully claimed the entry for the current thread, so you store it and stop the search #3. If you get to the end of the list without finding a free entry #4, there are too many threads using hazard pointers, so you throw an exception.

Once the `hp_owner` instance has been created for a given thread, further accesses are much faster because the pointer is cached, so the table doesn't have to be scanned again.

When each thread exits, if an instance of `hp_owner` was created for that thread, then it's destroyed. The destructor then resets the actual pointer to `nullptr` before setting the owner ID to `std::thread::id()`, allowing another thread to reuse the entry later #5.

With this implementation of `get_hazard_pointer_for_current_thread()`, the implementation of `outstanding_hazard_pointers_for()` is really simple: just scan through the hazard pointer table looking for entries:

```
bool outstanding_hazard_pointers_for(void* p)
{
    for(unsigned i=0;i<max_hazard_pointers;++i)
    {
        if(hazard_pointers[i].pointer.load()==p)
        {
            return true;
        }
    }
    return false;
}
```

It's not even worth checking whether each entry has an owner: unowned entries will have a null pointer, so the comparison will return `false` anyway, and it simplifies the code.

`reclaim_later()` and `delete_nodes_with_no_hazards()` can then work on a simple linked list: `reclaim_later()` just adds nodes to the list, and `delete_nodes_with_no_hazards()` scans through the list, deleting entries with no outstanding hazards. The next listing shows just such an implementation.

Listing 7.8 A simple implementation of the reclaim functions

```
template<typename T>
void do_delete(void* p)
{
    delete static_cast<T*>(p);
}
struct data_to_reclaim
{
    void* data;
    std::function<void(void*)> deleter;
    data_to_reclaim* next;
    template<typename T>
    data_to_reclaim(T* p):                      #1
        data(p),
        deleter(&do_delete<T>),
        next(0)
    {}
    ~data_to_reclaim()
    {
        deleter(data);                         #2
    }
};
std::atomic<data_to_reclaim*> nodes_to_reclaim;
void add_to_reclaim_list(data_to_reclaim* node)      #3
{
    node->next=nodes_to_reclaim.load();
    while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
}
template<typename T>
void reclaim_later(T* data)                          #4
{
```

```

        add_to_reclaim_list(new data_to_reclaim(data));          #5
    }
void delete_nodes_with_no_hazards()
{
    data_to_reclaim* current=nodes_to_reclaim.exchange(nullptr);    #6
    while(current)
    {
        data_to_reclaim* const next=current->next;
        if(!outstanding_hazard_pointers_for(current->data))      #7
        {
            delete current;                                     #8
        }
        else
        {
            add_to_reclaim_list(current);                      #9
        }
        current=next;
    }
}

```

First off, I expect you've spotted that `reclaim_later()` is a function template rather than a plain function #4. This is because hazard pointers are a general-purpose utility, so you don't want to tie yourselves to stack nodes. You've been using `std::atomic<void*>` for storing the pointers already. You therefore need to handle any pointer type, but you can't use `void*` because you want to delete the data items when you can, and `delete` requires the real type of the pointer. The constructor of `data_to_reclaim` handles that nicely, as you'll see in a minute: `reclaim_later()` just creates a new instance of `data_to_reclaim` for your pointer and adds it to the reclaim list #5. `add_to_reclaim_list()` itself #3 is just a simple `compare_exchange_weak()` loop on the list head like you've seen before.

So, back to the constructor of `data_to_reclaim` #1: the constructor is also a template. It stores the data to be deleted as a `void*` in the `data` member and then stores a pointer to the appropriate instantiation of `do_delete()`—a simple function that casts the supplied `void*` to the chosen pointer type and then deletes the pointed-to object. `std::function<>` wraps this function pointer safely, so that the destructor of `data_to_reclaim` can then delete the data just by invoking the stored function #2.

The destructor of `data_to_reclaim` isn't called when you're adding nodes to the list; it's called when there are no more hazard pointers to that node. This is the responsibility of `delete_nodes_with_no_hazards()`.

`delete_nodes_with_no_hazards()` first claims the entire list of nodes to be reclaimed for itself with a simple `exchange()` #6. This simple but crucial step ensures that this is the only thread trying to reclaim this particular set of nodes. Other threads are now free to add further nodes to the list or even try to reclaim them without impacting the operation of this thread.

Then, as long as there are still nodes left in the list, you check each node in turn to see if there are any outstanding hazard pointers #7. If there aren't, you can safely delete the entry (and thus clean up the stored data) #8. Otherwise, you just add the item back on the list for reclaiming later #9.

Although this simple implementation does indeed safely reclaim the deleted nodes, it adds quite a bit of overhead to the process. Scanning the hazard pointer array requires checking `max_hazard_pointers` atomic variables, and this is done for every `pop()` call. Atomic operations are inherently slow—often 100 times slower than an equivalent nonatomic operation on desktop CPUs—so this makes `pop()` an expensive operation. Not only do you scan the hazard pointer list for the node you’re about to remove, but you also scan it for each node in the waiting list. Clearly this is a bad idea. There may well be `max_hazard_pointers` nodes in the list, and you’re checking all of them against `max_hazard_pointers` stored hazard pointers. Ouch! There has to be a better way.

BETTER RECLAMATION STRATEGIES USING HAZARD POINTERS

Of course, there *is* a better way. What I’ve shown here is a simple and naïve implementation of hazard pointers to help explain the technique. The first thing you can do is trade memory for performance. Rather than checking every node on the reclamation list every time you call `pop()`, you don’t try to reclaim any nodes at all unless there are more than `max_hazard_pointers` nodes on the list. That way you’re guaranteed to be able to reclaim at least one node. If you just wait until there are `max_hazard_pointers+1` nodes on the list, you’re not much better off. Once you get to `max_hazard_pointers` nodes, you’ll be trying to reclaim nodes for most calls to `pop()`, so you’re not doing much better. But if you wait until there are $2 * \text{max_hazard_pointers}$ nodes on the list, then at most `max_hazard_pointers` of those will still be active, so you’re guaranteed to be able to reclaim at least `max_hazard_pointers` nodes, and it will then be at least `max_hazard_pointers` calls to `pop()` before you try to reclaim any nodes again. This is much better. Rather than checking around `max_hazard_pointers` nodes every call to `push()` (and not necessarily reclaiming any), you’re checking $2 * \text{max_hazard_pointers}$ nodes every `max_hazard_pointers` calls to `pop()` and reclaiming at least `max_hazard_pointers` nodes. That’s effectively two nodes checked for every `pop()`, one of which is reclaimed.

Even this has a downside (beyond the increased memory usage from the larger reclamation list, and the larger number of potentially reclaimable nodes): you now have to count the nodes on the reclamation list, which means using an atomic count, and you still have multiple threads competing to access the reclamation list itself. If you have memory to spare, you can trade increased memory usage for an even better reclamation scheme: each thread keeps its own reclamation list in a thread-local variable. There’s thus no need for atomic variables for the count or the list access. Instead, you have `max_hazard_pointers * max_hazard_pointers` nodes allocated. If a thread exits before all its nodes have been reclaimed, they can be stored in the global list as before and added to the local list of the next thread doing a reclamation process.

Another downside of hazard pointers is that they’re covered by a patent application submitted by IBM.² Though I believe this patent has now expired, if you write software for use

² Maged M. Michael, U.S. Patent and Trademark Office application number 20040107227, “Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation.”

in a country where the patents are valid, it is as well to get a patent lawyer to verify that for you, or you need to make sure you have a suitable licensing arrangement in place. This is something common to many of the lock-free memory reclamation techniques; this is an active research area, so large companies are taking out patents where they can. You may well be asking why I've devoted so many pages to a technique that people may be unable to use, and that's a fair question. First, it may be possible to use the technique without paying for a license. For example, if you're developing free software licensed under the GPL,³ your software may be covered by IBM's statement of non-assertion.⁴ Second, and most important, the explanation of the techniques shows some of the things that are important to think about when writing lock-free code, such as the costs of atomic operations.

So, are there any unpatented memory reclamation techniques that can be used with lock-free code? Luckily, there are. One such mechanism is reference counting.

7.2.4 Detecting nodes in use with reference counting

Back in section 7.2.2, you saw that the problem with deleting nodes is detecting which nodes are still being accessed by reader threads. If you could safely identify precisely which nodes were being referenced and when no threads were accessing these nodes, you could delete them. Hazard pointers tackle the problem by storing a list of the nodes in use. Reference counting tackles the problem by storing a count of the number of threads accessing each node.

This may seem nice and straightforward, but it's quite hard to manage in practice. At first, you might think that something like `std::shared_ptr<>` would be up to the task; after all, it's a reference-counted pointer. Unfortunately, although some operations on `std::shared_ptr<>` are atomic, they aren't guaranteed to be lock-free. Although by itself this is no different than any of the operations on the atomic types, `std::shared_ptr<>` is intended for use in many contexts, and making the atomic operations lock-free would likely impose an overhead on all uses of the class. If your platform supplies an implementation for which `std::atomic_is_lock_free(&some_shared_ptr)` returns true, the whole memory reclamation issue goes away. Just use `std::shared_ptr<node>` for the list, as in the following listing. Note the need to clear the next pointer from the popped node in order to avoid the potential for deeply-nested destruction of nodes when the last `std::shared_ptr` referencing a given node is destroyed.

Listing 7.9 A lock-free stack using a lock-free `std::shared_ptr<>` implementation

```
template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;
```

³ GNU General Public License <http://www.gnu.org/licenses/gpl.html>.

⁴ IBM Statement of Non-Assertion of Named Patents Against OSS, <http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf>.

```

        std::shared_ptr<node> next;
    node(T const& data_):
        data(std::make_shared<T>(data_))
    {}
};

std::shared_ptr<node> head;
public:
    void push(T const& data)
    {
        std::shared_ptr<node> const new_node=std::make_shared<node>(data);
        new_node->next=std::atomic_load(&head);
        while(!std::atomic_compare_exchange_weak(&head,
            &new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        std::shared_ptr<node> old_head=std::atomic_load(&head);
        while(old_head && !std::atomic_compare_exchange_weak(&head,
            &old_head,std::atomic_load(&old_head->next)));
        if(old_head) {
            std::atomic_store(&old_head->next,std::shared_ptr<node>());
            return old_head->data;
        }
        return std::shared_ptr<T>();
    }
    ~lock_free_stack(){
        while(pop());
    }
};

```

Not only is it rare for an implementation to provide lock-free atomic operations on `std::shared_ptr<T>`, but remembering to use the atomic operations consistently is hard. The Concurrency TS helps us out, if you have an implementation available, since it provides `std::experimental::atomic_shared_ptr<T>` in the `<experimental/atomic>` header. This is in many ways equivalent to a theoretical `std::atomic<std::shared_ptr<T>>`, except that `std::shared_ptr<T>` can't be used with `std::atomic<T>`, because it has non-trivial copy semantics to ensure that the reference count is handled correctly. `std::experimental::atomic_shared_ptr<T>` handles the reference counting correctly, while still ensuring atomic operations. Just like the other atomic types described in chapter 5, it may or may not be lock free on any given implementation. Listing 7.9 can thus be rewritten as in listing 7.10. See how much simpler it is without having to remember to include the `atomic_load` and `atomic_store` calls.

Listing 7.10 A lock-free stack using a lock-free `std::experimental::atomic_shared_ptr<T>` implementation

```

template<typename T>
class lock_free_stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;

```

```

        std::experimental::atomic_shared_ptr<node> next;
    node(T const& data_):
        data(std::make_shared<T>(data_))
    {}
};

std::experimental::atomic_shared_ptr<node> head;
public:
    void push(T const& data)
    {
        std::shared_ptr<node> const new_node=std::make_shared<node>(data);
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        std::shared_ptr<node> old_head=head.load();
        while(old_head && !head.compare_exchange_weak(
            old_head,old_head->next.load()));
        if(old_head) {
            old_head->next=std::shared_ptr<node>();
            return old_head->data;
        }
        return std::shared_ptr<T>();
    }
    ~lock_free_stack(){
        while(pop());
    }
};

```

In the probable case that your `std::shared_ptr<>` implementation isn't lock-free, and your implementation doesn't provide a lock-free `std::experimental::atomic_shared_ptr<>` either, you need to manage the reference counting manually.

One possible technique involves the use of not one but two reference counts for each node: an internal count and an external count. The sum of these values is the total number of references to the node. The external count is kept alongside the pointer to the node and is increased every time the pointer is read. When the reader is finished with the node, it decreases the *internal* count. A simple operation that reads the pointer will thus leave the external count increased by one and the internal count decreased by one when it's finished.

When the external count/pointer pairing is no longer required (that is, the node is no longer accessible from a location accessible to multiple threads), the internal count is increased by the value of the external count minus one and the external counter is discarded. Once the internal count is equal to zero, there are no outstanding references to the node and it can be safely deleted. It's still important to use atomic operations for updates of shared data. Let's now look at an implementation of a lock-free stack that uses this technique to ensure that the nodes are reclaimed only when it's safe to do so.

The following listing shows the internal data structure and the implementation of `push()`, which is nice and straightforward.

Listing 7.11 Pushing a node on a lock-free stack using split reference counts

```
template<typename T>
```

```

class lock_free_stack
{
private:
    struct node;
    struct counted_node_ptr      #1
    {
        int external_count;
        node* ptr;
    };
    struct node
    {
        std::shared_ptr<T> data;
        std::atomic<int> internal_count;      #2
        counted_node_ptr next;              #3
        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };
    std::atomic<counted_node_ptr> head;      #4
public:
    ~lock_free_stack()
    {
        while(pop());
    }
    void push(T const& data)           #5
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load();
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
    }
};

```

First, the external count is wrapped together with the node pointer in the `counted_node_ptr` structure #1. This can then be used for the `next` pointer in the `node` structure #3 alongside the internal count #2. Because `counted_node_ptr` is just a simple `struct`, you can use it with the `std::atomic<>` template for the `head` of the list #4.

On those platforms that support a double-word-compare-and-swap operation, this structure will be small enough for `std::atomic<counted_node_ptr>` to be lock-free. If it isn't on your platform, you might be better off using the `std::shared_ptr<>` version from listing 7.9, because `std::atomic<>` will use a mutex to guarantee atomicity when the type is too large for the platform's atomic instructions (thus rendering your "lock-free" algorithm lock-based after all). Alternatively, if you're willing to limit the size of the counter, and you know that your platform has spare bits in a pointer (for example, because the address space is only 48 bits but a pointer is 64 bits), you can store the count inside the spare bits of the pointer to fit it all back in a single machine word. Such tricks require platform-specific knowledge and are thus outside the scope of this book.

`push()` is relatively simple #5. You construct a `counted_node_ptr` that refers to a freshly allocated `node` with associated data and set the `next` value of the `node` to the current value of `head`. You can then use `compare_exchange_weak()` to set the value of `head`, just as in the

previous listings. The counts are set up so the `internal_count` is zero, and the `external_count` is one. Because this is a new node, there's currently only one external reference to the node (the head pointer itself).

As usual, the complexities come to light in the implementation of `pop()`, which is shown in the following listing.

Listing 7.12 Popping a node from a lock-free stack using split reference counts

```
template<typename T>
class lock_free_stack
{
private:
    // other parts as in listing 7.11
    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;
        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter)); #1
        old_counter.external_count=new_counter.external_count;
    }
public:
    std::shared_ptr<T> pop()#
    {
        counted_node_ptr old_head=head.load();
        for(;;)
        {
            increase_head_count(old_head);
            node* const ptr=old_head.ptr;                      #2
            if(!ptr)
            {
                return std::shared_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next)) #3
            {
                std::shared_ptr<T> res;
                res.swap(ptr->data);                           #4
                int const count_increase=old_head.external_count-2; #5
                if(ptr->internal_count.fetch_add(count_increase)== #6
                   -count_increase)
                {
                    delete ptr;
                }
                return res;                                     #7
            }
            else if(ptr->internal_count.fetch_sub(1)==1)      #8
            {
                delete ptr;
            }
        }
    }
};
```

This time, once you've loaded the value of `head`, you must first increase the count of external references to the `head` node to indicate that you're referencing it and to ensure that it's safe to dereference it. If you dereference the pointer *before* increasing the reference count, another thread could free the node before you access it, thus leaving you with a dangling pointer. *This is the primary reason for using the split reference count:* by incrementing the external reference count, you ensure that the pointer remains valid for the duration of your access. The increment is done with a `compare_exchange_strong()` loop #1 that compares and sets the whole structure to ensure that the pointer hasn't been changed by another thread in the meantime.

Once the count has been increased, you can safely dereference the `ptr` field of the value loaded from `head` in order to access the pointed-to node #2. If the pointer is a null pointer, you're at the end of the list: no more entries. If the pointer isn't a null pointer, you can try to remove the node by a `compare_exchange_strong()` call on `head` #3.

If the `compare_exchange_strong()` succeeds, you've taken ownership of the node and can swap out the data in preparation for returning it #4. This ensures that the data isn't kept alive just because other threads accessing the stack happen to still have pointers to its node. Then you can add the external count to the internal count on the node with an atomic `fetch_add` #6. If the reference count is now zero, the *previous* value (which is what `fetch_add` returns) was the negative of what you just added, in which case you can delete the node. It's important to note that the value you add is actually *two less* than the external count #5; you've removed the node from the list, so you drop one off the count for that, and you're no longer accessing the node from this thread, so you drop another off the count for that. Whether or not you deleted the node, you've finished, so you can return the data #7.

If the compare/exchange #3 *fails*, another thread removed your node before you did, or another thread added a new node to the stack. Either way, you need to start again with the fresh value of `head` returned by the compare/exchange call. But first you must decrease the reference count on the node you were trying to remove. This thread won't access it anymore. If you're the last thread to hold a reference (because another thread removed it from the stack), the internal reference count will be 1, so subtracting 1 will set the count to zero. In this case, you can delete the node here before you loop #8.

So far, you've been using the default `std::memory_order_seq_cst` memory ordering for all your atomic operations. On most systems these are more expensive in terms of execution time and synchronization overhead than the other memory orderings, and on some systems considerably so. Now that you have the logic of your data structure right, you can think about relaxing some of these memory-ordering requirements; you don't want to impose any unnecessary overhead on the users of the stack. So, before leaving your stack behind and moving on to the design of a lock-free queue, let's examine the stack operations and ask ourselves, can we use more relaxed memory orderings for some operations and still get the same level of safety?

7.2.5 Applying the memory model to the lock-free stack

Before you go about changing the memory orderings, you need to examine the operations and identify the required relationships between them. You can then go back and find the minimum memory orderings that provide these required relationships. In order to do this, you'll have to look at the situation from the point of view of threads in several different scenarios. The simplest possible scenario has to be where one thread pushes a data item onto the stack and another thread then pops that data item off the stack some time later, so we'll start from there.

In this simple case, three important pieces of data are involved. First is the `counted_node_ptr` used for transferring the data: `head`. Second is the node structure that `head` refers to, and third is the data item pointed to by that node.

The thread doing the `push()` first constructs the data item and the node and then sets `head`. The thread doing the `pop()` first loads the value of `head`, then does a compare/exchange loop on `head` to increase the reference count, and then reads the node structure to obtain the `next` value. Right here you can see a required relationship; the `next` value is a plain nonatomic object, so in order to read this safely, there must be a happens-before relationship between the store (by the pushing thread) and the load (by the popping thread). Because the only atomic operation in the `push()` is the `compare_exchange_weak()`, and you need a *release* operation to get a happens-before relationship between threads, the `compare_exchange_weak()` must be `std::memory_order_release` or stronger. If the `compare_exchange_weak()` call fails, nothing has changed and you keep looping, so you need only `std::memory_order_relaxed` in that case:

```
void push(T const& data)
{
    counted_node_ptr new_node;
    new_node.ptr=new_node(data);
    new_node.external_count=1;
    new_node.ptr->next=head.load(std::memory_order_relaxed);
    while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
                                      std::memory_order_release,std::memory_order_relaxed));
}
```

What about the `pop()` code? In order to get the happens-before relationship you need, you must have an operation that's `std::memory_order_acquire` or stronger before the access to `next`. The pointer you dereference to access the `next` field is the old value read by the `compare_exchange_strong()` in `increase_head_count()`, so you need the ordering on that if it succeeds. As with the call in `push()`, if the exchange fails, you just loop again, so you can use relaxed ordering on failure:

```
void increase_head_count(counted_node_ptr& old_counter)
{
    counted_node_ptr new_counter;
    do
    {
        new_counter=old_counter;
        ++new_counter.external_count;
```

```

    }
    while(!head.compare_exchange_strong(old_counter,new_counter,
        std::memory_order_acquire,std::memory_order_relaxed));
    old_counter.external_count=new_counter.external_count;
}

```

If the `compare_exchange_strong()` call succeeds, you know that the value read had the `ptr` field set to what's now stored in `old_counter`. Because the store in `push()` was a release operation, and this `compare_exchange_strong()` is an acquire operation, the store synchronizes with the load and you have a happens-before relationship. Consequently, the store to the `ptr` field in the `push()` happens before the `ptr->next` access in `pop()`, and you're safe.

Note that the memory ordering on the initial `head.load()` didn't matter to this analysis, so you can safely use `std::memory_order_relaxed` for that.

Next up, let us consider the `compare_exchange_strong()` to set `head` to `old_head.ptr->next`. Do you need anything from this operation to guarantee the data integrity of this thread? If the exchange succeeds, you access `ptr->data`, so you need to ensure that the store to `ptr->data` in the `push()` thread happens before the load. However, you already have that guarantee: the acquire operation in `increase_head_count()` ensures that there's a synchronizes-with relationship between the store in the `push()` thread and that compare/exchange. Because the store to `data` in the `push()` thread is sequenced before the store to `head` and the call to `increase_head_count()` is sequenced before the load of `ptr->data`, there's a happens-before relationship, and all is well even if this compare/exchange in `pop()` uses `std::memory_order_relaxed`. The only other place where `ptr->data` is changed is the very call to `swap()` that you're looking at, and no other thread can be operating on the same node; that's the whole point of the compare/exchange.

If the `compare_exchange_strong()` fails, the new value of `old_head` isn't touched until next time around the loop, and you already decided that the `std::memory_order_acquire` in `increase_head_count()` was enough, so `std::memory_order_relaxed` is enough there also.

What about other threads? Do you need anything stronger here to ensure other threads are still safe? The answer is, no, because `head` is only ever modified by compare/exchange operations. Because these are read-modify-write operations, they form part of the release sequence headed by the compare/exchange in `push()`. Therefore, the `compare_exchange_weak()` in `push()` synchronizes with a call to `compare_exchange_strong()` in `increase_head_count()`, which reads the value stored, even if many other threads modify `head` in the meantime.

So you've nearly finished: the only remaining operations to deal with are the `fetch_add()` operations for modifying the reference count. The thread that got to return the data from this node can proceed, safe in the knowledge that no other thread can have modified the node data. However, any thread that did *not* successfully retrieve the data knows that another thread *did* modify the node data; the successful thread used `swap()` to extract the referenced data item. Therefore you need to ensure that the `swap()` happens-before the delete in order to avoid a data race. The easy way to do this is to make the `fetch_add()` in the successful-

return branch use `std::memory_order_release` and the `fetch_add()` in the loop-again branch use `std::memory_order_acquire`. However, this is still overkill: only one thread does the delete (the one that sets the count to zero), so only that thread needs to do an acquire operation. Thankfully, because `fetch_add()` is a read-modify-write operation, it forms part of the release sequence, so you can do that with an additional `load()`. If the loop-again branch decreases the reference count to zero, it can reload the reference count with `std::memory_order_acquire` in order to ensure the required synchronizes-with relationship, and the `fetch_add()` itself can use `std::memory_order_relaxed`. The final stack implementation with the new version of `pop()` is shown here.

Listing 7.13 A lock-free stack with reference counting and relaxed atomic operations

```
template<typename T>
class lock_free_stack
{
private:
    struct node;
    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };
    struct node
    {
        std::shared_ptr<T> data;
        std::atomic<int> internal_count;
        counted_node_ptr next;
        node(T const& data_):
            data(std::make_shared<T>(data_)),
            internal_count(0)
        {}
    };
    std::atomic<counted_node_ptr> head;
    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;
        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter,
                                             std::memory_order_acquire,
                                             std::memory_order_relaxed));
        old_counter.external_count=new_counter.external_count;
    }
public:
    ~lock_free_stack()
    {
        while(pop());
    }
    void push(T const& data)
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
```

```

new_node.external_count=1;
new_node.ptr->next=head.load(std::memory_order_relaxed)
while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
                                 std::memory_order_release,
                                 std::memory_order_relaxed));
}
std::shared_ptr<T> pop()
{
    counted_node_ptr old_head=
        head.load(std::memory_order_relaxed);
    for(;;)
    {
        increase_head_count(old_head);
        node* const ptr=old_head.ptr;
        if(!ptr)
        {
            return std::shared_ptr<T>();
        }
        if(head.compare_exchange_strong(old_head,ptr->next,
                                         std::memory_order_relaxed))
        {
            std::shared_ptr<T> res;
            res.swap(ptr->data);
            int const count_increase=old_head.external_count-2;
            if(ptr->internal_count.fetch_add(count_increase,
                                              std::memory_order_release)==-count_increase)
            {
                delete ptr;
            }
            return res;
        }
        else if(ptr->internal_count.fetch_add(-1,
                                              std::memory_order_relaxed)==1)
        {
            ptr->internal_count.load(std::memory_order_acquire);
            delete ptr;
        }
    }
}
};

```

That was quite a workout, but you got there in the end, and the stack is better for it. By using more relaxed operations in a carefully thought-through manner, the performance is improved without impacting the correctness. As you can see, the implementation of `pop()` is now 37 lines rather than the 8 lines of the equivalent `pop()` in the lock-based stack of listing 6.1 and the 7 lines of the basic lock-free stack without memory management in listing 7.2. As we move on to look at writing a lock-free queue, you'll see a similar pattern: lots of the complexity in lock-free code comes from managing memory.

7.2.6 Writing a thread-safe queue without locks

A queue offers a slightly different challenge to a stack, because the `push()` and `pop()` operations access different parts of the data structure in a queue, whereas they both access the same head node for a stack. Consequently, the synchronization needs are different. You

need to ensure that changes made to one end are correctly visible to accesses at the other. However, the structure of `try_pop()` for the queue in listing 6.6 isn't actually that far off that of `pop()` for the simple lock-free stack in listing 7.2, so you can reasonably assume that the lock-free code won't be that dissimilar. Let's see how.

If you take listing 6.6 as a basis, you need two `node` pointers: one for the `head` of the list and one for the `tail`. You're going to be accessing these from multiple threads, so they'd better be atomic in order to allow you to do away with the corresponding mutexes. Let's start by making that small change and see where it gets you. The following listing shows the result.

Listing 7.14 A single-producer, single-consumer lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;
        node():
            next(nullptr)
        {}
    };
    std::atomic<node*> head;
    std::atomic<node*> tail;
    node* pop_head()
    {
        node* const old_head=head.load();
        if(old_head==tail.load())          #1
        {
            return nullptr;
        }
        head.store(old_head->next);
        return old_head;
    }
public:
    lock_free_queue():
        head(new node),tail(head.load())
    {}
    lock_free_queue(const lock_free_queue& other)=delete;
    lock_free_queue& operator=(const lock_free_queue& other)=delete;
    ~lock_free_queue()
    {
        while(node* const old_head=head.load())
        {
            head.store(old_head->next);
            delete old_head;
        }
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=pop_head();
        if(!old_head)
        {
            return std::shared_ptr<T>();
        }
        else
        {
            std::shared_ptr<T> result=old_head->data;
            old_head->data.reset();
            old_head->next=tail.load();
            tail.store(old_head);
            return result;
        }
    }
};
```

```

    }
    std::shared_ptr<T> const res(old_head->data);      #2
    delete old_head;
    return res;
}
void push(T new_value)
{
    std::shared_ptr<T> new_data(std::make_shared<T>(new_value));
    node* p=new node;                                #3
    node* const old_tail=tail.load();                 #4
    old_tail->data.swap(new_data);                  #5
    old_tail->next=p;                            #6
    tail.store(p);                           #7
}
};

```

At first glance, this doesn't seem too bad, and if there's only one thread calling `push()` at a time, and only one thread calling `pop()`, then this is actually perfectly fine. The important thing in that case is the happens-before relationship between the `push()` and the `pop()` to ensure that it's safe to retrieve the data. The store to `tail` #7 synchronizes with the load from `tail` #1; the store to the preceding node's data pointer #5 is sequenced before the store to `tail`; and the load from `tail` is sequenced before the load from the data pointer #2, so the store to data happens before the load, and everything is OK. This is therefore a perfectly serviceable *single-producer, single-consumer (SPSC)* queue.

The problems come when multiple threads call `push()` concurrently or multiple threads call `pop()` concurrently. Let's look at `push()` first. If you have two threads calling `push()` concurrently, they both allocate new nodes to be the new dummy node #3, both read the *same* value for `tail` #4, and consequently both update the data members of the same node when setting the `data` and `next` pointers #5, #6. This is a data race!

There are similar problems in `pop_head()`. If two threads call concurrently, they will both read the same value of `head`, and both then overwrite the `old` value with the same `next` pointer. Both threads will now think they've retrieved the same node—a recipe for disaster. Not only do you have to ensure that only one thread `pop()`s a given item, but you also need to ensure that other threads can safely access the `next` member of the node they read from `head`. This is exactly the problem you saw with `pop()` for your lock-free stack, so any of the solutions for that could be used here.

So if `pop()` is a "solved problem," what about `push()`? The problem here is that in order to get the required happens-before relationship between `push()` and `pop()`, you need to set the data items on the dummy node before you update `tail`. But this then means that concurrent calls to `push()` are racing over those very same data items, because they've read the same `tail` pointer.

HANDLING MULTIPLE THREADS IN PUSH()

One option is to add a dummy node between the real nodes. This way, the only part of the current `tail` node that needs updating is the `next` pointer, which could therefore be made atomic. If a thread manages to successfully change the `next` pointer from `nullptr` to its new

node, then it has successfully added the pointer; otherwise, it would have to start again and reread the tail. This would then require a minor change to `pop()` in order to discard nodes with a null data pointer and loop again. The downside here is that every `pop()` call will typically have to remove two nodes, and there are twice as many memory allocations.

A second option is to make the data pointer atomic and set that with a call to compare/exchange. If the call succeeds, this is your tail node, and you can safely set the next pointer to your new node and then update tail. If the compare/exchange fails because another thread has stored the data, you loop around, reread tail, and start again. If the atomic operations on `std::shared_ptr<T>` are lock-free, you're home free. If not, you need an alternative. One possibility is to have `pop()` return a `std::unique_ptr<T>` (after all, it's the only reference to the object) and store the data as a plain pointer in the queue. This would allow you to store it as a `std::atomic<T*>`, which would then support the necessary `compare_exchange_strong()` call. If you're using the reference-counting scheme from listing 7.12 to handle multiple threads in `pop()`, `push()` now looks like this.

Listing 7.15 A (broken) first attempt at revising `push()`

```
void push(T new_value)
{
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new_node;
    new_next.external_count=1;
    for(;;)
    {
        node* const old_tail=tail.load();           #1
        T* old_data=nullptr;
        if(old_tail->data.compare_exchange_strong(
            old_data,new_data.get()))               #2
        {
            old_tail->next=new_next;
            tail.store(new_next.ptr);             #3
            new_data.release();
            break;
        }
    }
}
```

Using the reference-counting scheme avoids this particular race, but it's not the only race in `push()`. If you look at the revised version of `push()` in listing 7.15, you'll see a pattern you saw in the stack: load an atomic pointer #1 and dereference that pointer #2. In the meantime, another thread could update the pointer #3, eventually leading to the node being deallocated (in `pop()`). If the node is deallocated before you dereference the pointer, you have undefined behavior. Ouch! It's tempting to add an external count in `tail` the same as you did for `head`, but each node already has an external count in the `next` pointer of the previous node in the queue. Having two external counts for the same node requires a modification to the reference-counting scheme to avoid deleting the node too early. You can address this by also counting the number of external counters inside the `node` structure and decreasing this

number when each external counter is destroyed (as well as adding the corresponding external count to the internal count). If the internal count is zero and there are no external counters, you know the node can safely be deleted. This is a technique I first encountered through Joe Seigh's Atomic Ptr Plus Project.⁵ The following listing shows how `push()` looks under this scheme.

Listing 7.16 Implementing `push()` for a lock-free queue with a reference-counted tail

```
template<typename T>
class lock_free_queue
{
private:
    struct node;
    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };
    std::atomic<counted_node_ptr> head;
    std::atomic<counted_node_ptr> tail;      #1
    struct node_counter
    {
        unsigned internal_count:30;
        unsigned external_counters:2;          #2
    };
    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;      #3
        counted_node_ptr next;
        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
            new_count.external_counters=2;    #4
            count.store(new_count);

            next.ptr=nullptr;
            next.external_count=0;
        }
    };
public:
    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();
        for(;;)
        {
            increase_external_count(tail,old_tail);    #5
            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(  #6
```

⁵ Atomic Ptr Plus Project, <http://atomic-ptr-plus.sourceforge.net/>.

```

        old_data,new_data.get()))
{
    old_tail.ptr->next=new_next;
    old_tail=tail.exchange(new_next);
    free_external_counter(old_tail);           #7
    new_data.release();
    break;
}
old_tail.ptr->release_ref();
}
};


```

In listing 7.16, tail is now an `atomic<counted_node_ptr>` the same as head #1, and the node structure has a `count` member to replace the `internal_count` from before #3. This `count` is a structure containing the `internal_count` and an additional `external_counters` member #2. Note that you need only 2 bits for the `external_counters` because there are at most two such counters. By using a bit field for this and specifying `internal_count` as a 30-bit value, you keep the total counter size to 32 bits. This gives you plenty of scope for large internal count values while ensuring that the whole structure fits inside a machine word on 32-bit and 64-bit machines. It's important to update these counts together as a single entity in order to avoid race conditions, as you'll see shortly. Keeping the structure within a machine word makes it more likely that the atomic operations can be lock-free on many platforms.

The node is initialized with the `internal_count` set to zero and the `external_counters` set to 2 #4 because every new node starts out referenced from tail and from the next pointer of the previous node once you've actually added it to the queue. `push()` itself is similar to listing 7.15, except that before you dereference the value loaded from tail in order to call to `compare_exchange_strong()` on the data member of the node #6, you call a new function `increase_external_count()` to increase the count #5, and then afterward you call `free_external_counter()` on the old tail value #7.

With the `push()` side dealt with, let's take a look at `pop()`. This is shown in the following listing and blends the reference-counting logic from the `pop()` implementation in listing 7.12 with the queue-pop logic from listing 7.14.

Listing 7.17 Popping a node from a lock-free queue with a reference-counted tail

```

template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        void release_ref();
    };
public:
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed); #1
        for(;;)
        {


```

```

increase_external_count(head,old_head);                      #2
node* const ptr=old_head.ptr;
if(ptr==tail.load().ptr)
{
    ptr->release_ref();                                     #3
    return std::unique_ptr<T>();
}
if(head.compare_exchange_strong(old_head,ptr->next))      #4
{
    T* const res=ptr->data.exchange(nullptr);
    free_external_counter(old_head);                         #5
    return std::unique_ptr<T>(res);
}
ptr->release_ref();                                       #6
}
};


```

You prime the pump by loading the `old_head` value before you enter the loop #1 and before you increase the external count on the loaded value #2. If the `head` node is the same as the `tail` node, you can release the reference #3 and return a null pointer because there's no data in the queue. If there is data, you want to try to claim it for yourself, and you do this with the call to `compare_exchange_strong()` #4. As with the stack in listing 7.12, this compares the external count and pointer as a single entity; if either changes, you need to loop again, after releasing the reference #6. If the exchange succeeded, you've claimed the data in the node as yours, so you can return that to the caller after you've released the external counter to the popped node #5. Once both the external reference counts have been freed and the internal count has dropped to zero, the node itself can be deleted. The reference-counting functions that take care of all this are shown in listings 7.18, 7.19, and 7.20.

Listing 7.18 Releasing a node reference in a lock-free queue

```

template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        void release_ref()
        {
            node_counter old_counter=
                count.load(std::memory_order_relaxed);
            node_counter new_counter;
            do
            {
                new_counter=old_counter;
                --new_counter.internal_count;          #1
            }
            while(!count.compare_exchange_strong(      #2
                old_counter,new_counter,
                std::memory_order_acquire,std::memory_order_relaxed));
            if(!new_counter.internal_count &&
                !new_counter.external_counters)
            {


```

```

        delete this; #3
    }
};

};


```

The implementation of `node::release_ref()` is only slightly changed from the equivalent code in the implementation of `lock_free_stack::pop()` from listing 7.12. Whereas the code in listing 7.12 only has to handle a single external count, so you could just use a simple `fetch_sub`, the whole count structure now has to be updated atomically, even though you only want to modify the `internal_count` field #1. This therefore requires a compare/exchange loop #2. Once you've decremented the `internal_count`, if both the internal and external counts are now zero, this is the last reference, so you can delete the node #3.

Listing 7.19 Obtaining a new reference to a node in a lock-free queue

```

template<typename T>
class lock_free_queue
{
private:
    static void increase_external_count(
        std::atomic<counted_node_ptr>& counter,
        counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;
        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!counter.compare_exchange_strong(
            old_counter,new_counter,
            std::memory_order_acquire,std::memory_order_relaxed));
        old_counter.external_count=new_counter.external_count;
    }
};

```

Listing 7.19 is the other side. This time, rather than releasing a reference, you're obtaining a fresh one and increasing the external count. `increase_external_count()` is similar to the `increase_head_count()` function from listing 7.13, except that it has been made into a static member function that takes the external counter to update as the first parameter rather than operating on a fixed counter.

Listing 7.20 Freeing an external counter to a node in a lock-free queue

```

template<typename T>
class lock_free_queue
{
private:
    static void free_external_counter(counted_node_ptr &old_node_ptr)
    {
        node* const ptr=old_node_ptr.ptr;

```

```

int const count_increase=old_node_ptr.external_count-2;
node_counter old_counter=
    ptr->count.load(std::memory_order_relaxed);
node_counter new_counter;
do
{
    new_counter=old_counter;
    --new_counter.external_counters;           #1
    new_counter.internal_count+=count_increase;   #2
}
while(!ptr->count.compare_exchange_strong(      #3
    old_counter,new_counter,
    std::memory_order_acquire,std::memory_order_relaxed));
if(!new_counter.internal_count &&
    !new_counter.external_counters)
{
    delete ptr;                           #4
}
};


```

The counterpart to `increase_external_count()` is `free_external_counter()`. This is similar to the equivalent code from `lock_free_stack::pop()` in listing 7.12 but modified to handle the `external_counters` count. It updates the two counts using a single `compare_exchange_strong()` on the whole `count` structure #3, just as you did when decreasing the `internal_count` in `release_ref()`. The `internal_count` value is updated as in listing 7.12 #2, and the `external_counters` value is decreased by one #1. If *both* the values are now zero, there are no more references to the node, so it can be safely deleted #4. This has to be done as a single action (which therefore requires the compare/exchange loop) to avoid a race condition. If they're updated separately, two threads may both think they are the last one and thus both delete the node, resulting in undefined behavior.

Although this now works and is race-free, there's still a performance issue. Once one thread has started a `push()` operation by successfully completing the `compare_exchange_strong()` on `old_tail.ptr->data` (#5 from listing 7.16), no other thread can perform a `push()` operation. Any thread that tries will see the new value rather than `nullptr`, which will cause the `compare_exchange_strong()` call to fail and make that thread loop again. This is a busy wait, which consumes CPU cycles without achieving anything. Consequently, this is effectively a lock. The first `push()` call blocks other threads until it has completed, so *this code is no longer lock-free*. Not only that, but whereas the operating system can give priority to the thread that holds the lock on a mutex if there are blocked threads, it can't do so in this case, so the blocked threads will waste CPU cycles until the first thread is done. This calls for the next trick from the lock-free bag of tricks: the waiting thread can help the thread that's doing the `push()`.

MAKING THE QUEUE LOCK-FREE BY HELPING OUT ANOTHER THREAD

In order to restore the lock-free property of the code, you need to find a way for a waiting thread to make progress even if the thread doing the `push()` is stalled. One way to do this is to help the stalled thread by doing its work for it.

In this case, you know exactly what needs to be done: the `next` pointer on the tail node needs to be set to a new dummy node, and then the `tail` pointer itself must be updated. The thing about dummy nodes is that they're all equivalent, so it doesn't matter if you use the dummy node created by the thread that successfully pushed the data or the dummy node from one of the threads that's waiting to push. If you make the `next` pointer in a node atomic, you can then use `compare_exchange_strong()` to set the pointer. Once the `next` pointer is set, you can then use a `compare_exchange_weak()` loop to set the `tail` while ensuring that it's still referencing the same original node. If it isn't, someone else has updated it, and you can stop trying and loop again. This requires a minor change to `pop()` as well in order to load the `next` pointer; this is shown in the following listing.

Listing 7.21 `pop()` modified to allow helping on the `push()` side

```
template<typename T>
class lock_free_queue
{
private:
    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;
        std::atomic<counted_node_ptr> next;      #1
    };
public:
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_external_count(head,old_head);
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                return std::unique_ptr<T>();
            }
            counted_node_ptr next=ptr->next.load();      #2
            if(head.compare_exchange_strong(old_head,next))
            {
                T* const res=ptr->data.exchange(nullptr);
                free_external_counter(old_head);
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();
        }
    }
};
```

As I mentioned, the changes here are simple: the `next` pointer is now atomic #1, so the `load` at #2 is atomic. In this example, you're using the default `memory_order_seq_cst` ordering, so you could omit the explicit call to `load()` and rely on the load in the implicit conversion to `counted_node_ptr`, but putting in the explicit call reminds you where to add the explicit memory ordering later.

The code for `push()` is more involved and is shown here.

Listing 7.22 A sample `push()` with helping for a lock-free queue

```
template<typename T>
class lock_free_queue
{
private:
    void set_new_tail(counted_node_ptr &old_tail,           #1
                      counted_node_ptr const &new_tail)
    {
        node* const current_tail_ptr=old_tail.ptr;
        while(!tail.compare_exchange_weak(old_tail,new_tail) &&   #2
              old_tail.ptr==current_tail_ptr);
        if(old_tail.ptr==current_tail_ptr)                      #3
            free_external_counter(old_tail);                   #4
        else
            current_tail_ptr->release_ref();      #5
    }
public:
    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();
        for(;;)
        {
            increase_external_count(tail,old_tail);
            T* old_data=nullptr;
            if(old_tail.ptr->data.compare_exchange_strong(      #6
                old_data,new_data.get()))
            {
                counted_node_ptr old_next={0};
                if(!old_tail.ptr->next.compare_exchange_strong(   #7
                    old_next,new_next))
                {
                    delete new_next.ptr;                         #8
                    new_next=old_next;                          #9
                }
                set_new_tail(old_tail, new_next);
                new_data.release();
                break;
            }
            else                                              #10
            {
                counted_node_ptr old_next={0};
                if(old_tail.ptr->next.compare_exchange_strong(   #11
                    old_next,new_next))
                {
                    old_next=new_next;                      #12
                    new_next.ptr=new node;                  #13
                }
                set_new_tail(old_tail, old_next);          #14
            }
        }
    }
}
```

```
};
```

This is similar to the original `push()` from listing 7.16, but there are a few crucial differences. If you *do* set the data pointer #6, you need to handle the case where another thread has helped you, and there's now an `else` clause to do the helping #10.

Having set the data pointer in the node #6, this new version of `push()` updates the `next` pointer using `compare_exchange_strong()` #7. You use `compare_exchange_ strong()` to avoid looping. If the exchange fails, you know that another thread has already set the `next` pointer, so you don't need the new node you allocated at the beginning, and you can delete it #8. You also want to use the `next` value that the other thread set for updating `tail` #9.

The actual update of the `tail` pointer has been extracted into `set_new_tail()` #1. This uses a `compare_exchange_weak()` loop #2 to update the `tail`, because if other threads are trying to `push()` a new node, the `external_count` part may have changed, and you don't want to lose it. However, you also need to take care that you don't replace the value if another thread has successfully changed it already; otherwise, you may end up with loops in the queue, which would be a rather bad idea. Consequently, you need to ensure that the `ptr` part of the loaded value is the same if the compare/exchange fails. If the `ptr` is the same once the loop has exited #3, then you must have successfully set the `tail`, so you need to free the old external counter #4. If the `ptr` value is different, then another thread will have freed the counter, so you just need to release the single reference held by this thread #5.

If the thread calling `push()` failed to set the data pointer this time through the loop, it can help the successful thread to complete the update. First off, you try to update the `next` pointer to the new node allocated on this thread #11. If this succeeds, you want to use the node you allocated as the new `tail` node #12, and you need to allocate another new node in anticipation of actually managing to push an item on the queue #13. You can then try to set the `tail` node by calling `set_new_tail` before looping around again #14.

You may have noticed that there are rather a lot of `new` and `delete` calls for such a small piece of code, because new nodes are allocated on `push()` and destroyed in `pop()`. The efficiency of the memory allocator therefore has a considerable impact on the performance of this code; a poor allocator can completely destroy the scalability properties of a lock-free container such as this. The selection and implementation of such allocators is beyond the scope of this book, but it's important to bear in mind that the only way to know that an allocator is better is to try it and measure the performance of the code before and after. Common techniques for optimizing memory allocation include having a separate memory allocator on each thread and using free lists to recycle nodes rather than returning them to the allocator.

That's enough examples for now; instead, let's look at extracting some guidelines for writing lock-free data structures from the examples.

7.3 Guidelines for writing lock-free data structures

If you've followed through all the examples in this chapter, you'll appreciate the complexities involved in getting lock-free code right. If you're going to design your own data

structures, it helps to have some guidelines to focus on. The general guidelines regarding concurrent data structures from the beginning of chapter 6 still apply, but you need more than that. I've pulled a few useful guidelines out from the examples, which you can then refer to when designing your own lock-free data structures.

7.3.1 Guideline: use `std::memory_order_seq_cst` for prototyping

`std::memory_order_seq_cst` is much easier to reason about than any other memory ordering because all such operations form a total order. In all the examples in this chapter, you've started with `std::memory_order_seq_cst` and only relaxed the memory-ordering constraints once the basic operations were working. In this sense, using other memory orderings is an *optimization*, and as such you need to avoid doing it prematurely. In general, you can only determine which operations can be relaxed when you can see the full set of code that can operate on the guts of the data structure. Attempting to do otherwise just makes your life harder. This is complicated by the fact that the code may work when tested but isn't guaranteed. Unless you have an algorithm checker that can systematically test all possible combinations of thread visibilities that are consistent with the specified ordering guarantees (and such things do exist), just running the code isn't enough.

7.3.2 Guideline: use a lock-free memory reclamation scheme

One of the biggest difficulties with lock-free code is managing memory. It's essential to avoid deleting objects when other threads might still have references to them, but you still want to delete the object as soon as possible in order to avoid excessive memory consumption. In this chapter you've seen three techniques for ensuring that memory can safely be reclaimed:

- Waiting until no threads are accessing the data structure and deleting all objects that are pending deletion
- Using hazard pointers to identify that a thread is accessing a particular object
- Reference counting the objects so that they aren't deleted until there are no outstanding references

In all cases the key idea is to use some method to keep track of how many threads are accessing a particular object and only delete each object when it's no longer referenced from anywhere. There are many other ways of reclaiming memory in lock-free data structures. For example, this is the ideal scenario for using a garbage collector. It's much easier to write the algorithms if you know that the garbage collector will free the nodes when they're no longer used, but not before.

Another alternative is to recycle nodes and only free them completely when the data structure is destroyed. Because the nodes are reused, the memory never becomes invalid, so some of the difficulties in avoiding undefined behavior go away. The downside here is that another problem becomes more prevalent. This is the so-called *ABA problem*.

7.3.3 Guideline: watch out for the ABA problem

The ABA problem is something to be wary of in any compare/exchange-based algorithm. It goes like this:

1. Thread 1 reads an atomic variable x and finds it has value A .
2. Thread 1 performs some operation based on this value, such as dereferencing it (if it's a pointer) or doing a lookup or something.
3. Thread 1 is stalled by the operating system.
4. Another thread performs some operations on x that changes its value to B .
5. A thread then changes the data associated with the value A such that the value held by thread 1 is no longer valid. This may be as drastic as freeing the pointed-to memory or just changing an associated value.
6. A thread then changes x back to A based on this new data. If this is a pointer, it may be a new object that just happens to share the same address as the old one.
7. Thread 1 resumes and performs a compare/exchange on x , comparing against A . The compare/exchange succeeds (because the value is indeed A), but this is the wrong A value. The data originally read at step 2 is no longer valid, but thread 1 has no way of telling and will thus corrupt the data structure.

None of the algorithms presented here suffer from this problem, but it's easy to write lock-free algorithms that do. The most common way to avoid the problem is to include an ABA counter alongside the variable x . The compare/exchange operation is then done on the combined structure of x plus the counter as a single unit. Every time the value is replaced, the counter is incremented, so even if x has the same value, the compare/exchange will fail if another thread has modified x .

The ABA problem is particularly prevalent in algorithms that use free lists or otherwise recycle nodes rather than returning them to the allocator.

7.3.4 Guideline: identify busy-wait loops and help the other thread

In the final queue example you saw how a thread performing a push operation had to wait for another thread also performing a push to complete its operation before it could proceed. Left alone, this would have been a busy-wait loop, with the waiting thread wasting CPU time while failing to proceed. If you end up with a busy-wait loop, you effectively have a blocking operation and might as well use mutexes and locks. By modifying the algorithm so that the waiting thread performs the incomplete steps if it's scheduled to run before the original thread completes the operation, you can remove the busy-wait and the operation is no longer blocking. In the queue example this required changing a data member to be an atomic variable rather than a nonatomic variable and using compare/exchange operations to set it, but in more complex data structures it might require more extensive changes.

7.4 Summary

Following from the lock-based data structures of chapter 6, this chapter has described simple implementations of various lock-free data structures, starting with a stack and a queue, as before. You saw how you must take care with the memory ordering on your atomic operations to ensure that there are no data races and that each thread sees a coherent view of the data structure. You also saw how memory management becomes much harder for lock-free data structures than lock-based ones and examined a couple of mechanisms for handling it. You also saw how to avoid creating wait loops by helping the thread you're waiting for to complete its operation.

Designing lock-free data structures is a difficult task, and it's easy to make mistakes, but such data structures have scalability properties that are important in some situations. Hopefully, by following through the examples in this chapter and reading the guidelines, you'll be better equipped to design your own lock-free data structure, implement one from a research paper, or find the bug in the one your former colleague wrote just before he left the company.

Wherever data is shared between threads, you need to think about the data structures used and how the data is synchronized between threads. By designing data structures for concurrency, you can encapsulate that responsibility in the data structure itself, so the rest of the code can focus on the task it's trying to perform *with* the data rather than the data synchronization. You'll see this in action in chapter 8 as we move on from concurrent data structures to concurrent code in general. Parallel algorithms use multiple threads to improve their performance, and the choice of concurrent data structure is crucial where the algorithms need their worker threads to share data.

8

Designing concurrent code

This chapter covers

- Techniques for dividing data between threads
- Factors that affect the performance of concurrent code
- How performance factors affect the design of data structures
- Exception safety in multithreaded code
- Scalability
- Example implementations of several parallel algorithms

Most of the preceding chapters have focused on the tools you have in your C++ toolbox for writing concurrent code. In chapters 6 and 7 we looked at how to use those tools to design basic data structures that are safe for concurrent access by multiple threads. Much as a carpenter needs to know more than just how to build a hinge or a joint in order to make a cupboard or a table, there's more to designing concurrent code than the design and use of basic data structures. You now need to look at the wider context so you can build bigger structures that perform useful work. I'll be using multithreaded implementations of some of the C++ Standard Library algorithms as examples, but the same principles apply at all scales of an application.

Just as with any programming project, it's vital to think carefully about the design of concurrent code. However, with multithreaded code, there are even more factors to consider than with sequential code. Not only must you think about the usual factors such as encapsulation, coupling, and cohesion (which are amply described in the many books on software design), but you also need to consider which data to share, how to synchronize

accesses to that data, which threads need to wait for which other threads to complete certain operations, and so forth.

In this chapter we'll be focusing on these issues, from the high-level (but fundamental) considerations of how many threads to use, which code to execute on which thread, and how this can affect the clarity of the code, to the low-level details of how to structure the shared data for optimal performance.

Let's start by looking at techniques for dividing work between threads.

8.1 Techniques for dividing work between threads

Imagine for a moment that you've been tasked with building a house. In order to complete the job, you'll need to dig the foundation, build walls, put in plumbing, add the wiring, and so forth. Theoretically, you could do it all yourself with sufficient training, but it would probably take a long time, and you'd be continually switching tasks as necessary. Alternatively, you could hire a few other people to help out. You now have to choose how many people to hire and decide what skills they need. You could, for example, hire a couple of people with general skills and have everybody chip in with everything. You'd still all switch tasks as necessary, but now things can be done more quickly because there are more of you.

Alternatively, you could hire a team of specialists: a bricklayer, a carpenter, an electrician, and a plumber, for example. Your specialists just do whatever their specialty is, so if there's no plumbing needed, your plumber sits around drinking tea or coffee. Things still get done quicker than before, because there are more of you, and the plumber can put the toilet in while the electrician wires up the kitchen, but there's more waiting around when there's no work for a particular specialist. Even with the idle time, you might find that the work is done faster with specialists than with a team of general handymen. Your specialists don't need to keep changing tools, and they can probably each do their tasks quicker than the generalists can. Whether or not this is the case depends on the particular circumstances—you'd have to try it and see.

Even if you hire specialists, you can still choose to hire different numbers of each. It might make sense to have more bricklayers than electricians, for example. Also, the makeup of your team and the overall efficiency might change if you had to build more than one house. Even though your plumber might not have lots of work to do on any given house, you might have enough work to keep him busy all the time if you're building many houses at once. Also, if you don't have to pay your specialists when there's no work for them to do, you might be able to afford a larger team overall even if you have only the same number of people working at any one time.

OK, enough about building; what does all this have to do with threads? Well, with threads the same issues apply. You need to decide how many threads to use and what tasks they should be doing. You need to decide whether to have "generalist" threads that do whatever work is necessary at any point in time or "specialist" threads that do one thing well, or some combination. You need to make these choices whatever the driving reason for using concurrency, and quite how you do this will have a crucial effect on the performance and

clarity of the code. It's therefore vital to understand the options so you can make an appropriately informed decision when designing the structure of your application. In this section, we'll look at several techniques for dividing the tasks, starting with dividing data between threads before we do any other work.

8.1.1 Dividing data between threads before processing begins

The easiest algorithms to parallelize are simple algorithms such as `std::for_each` that perform an operation on each element in a data set. In order to parallelize such an algorithm, you can assign each element to one of the processing threads. How the elements are best divided for optimal performance depends very much on the details of the data structure, as you'll see later in this chapter when we look at performance issues.

The simplest means of dividing the data is to allocate the first N elements to one thread, the next N elements to another thread, and so on, as shown in figure 8.1, but other patterns could be used too. No matter how the data is divided, each thread then processes just the elements it has been assigned without any communication with the other threads until it has completed its processing.

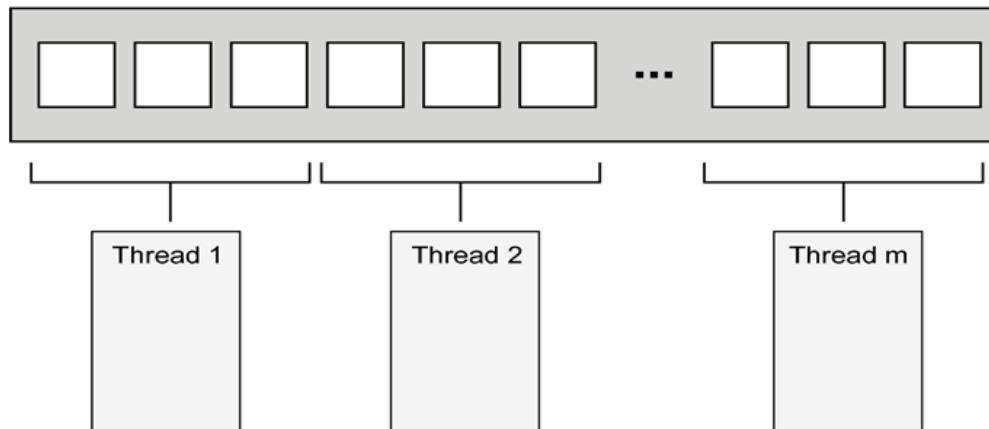


Figure 8.1 Distributing consecutive chunks of data between threads

This structure will be familiar to anyone who has programmed using the Message Passing Interface (MPI)¹ or OpenMP² frameworks: a task is split into a set of parallel tasks, the worker threads run these tasks independently, and the results are combined in a final *reduction* step. It's the approach used by the `accumulate` example from section 2.4; in this case, both the parallel tasks and the final reduction step are accumulations. For a simple `for_each`, the final step is a no-op because there are no results to reduce.

¹ <http://www.mpi-forum.org/>

² <http://www.openmp.org/>

Identifying this final step as a reduction is important; a naïve implementation such as listing 2.9 will perform this reduction as a final serial step. However, this step can often be parallelized as well; accumulate actually *is* a reduction operation itself, so listing 2.9 could be modified to call itself recursively where the number of threads is larger than the minimum number of items to process on a thread, for example. Alternatively, the worker threads could be made to perform some of the reduction steps as each one completes its task, rather than spawning new threads each time.

Although this technique is powerful, it can't be applied to everything. Sometimes the data can't be divided neatly up front because the necessary divisions become apparent only as the data is processed. This is particularly apparent with recursive algorithms such as Quicksort; they therefore need a different approach.

8.1.2 Dividing data recursively

The Quicksort algorithm has two basic steps: partition the data into items that come before or after one of the elements (the pivot) in the final sort order and recursively sort those two "halves." You can't parallelize this by simply dividing the data up front, because it's only by processing the items that you know which "half" they go in. If you're going to parallelize such an algorithm, you need to make use of the recursive nature. With each level of recursion there are *more* calls to the `quick_sort` function, because you have to sort both the elements that belong before the pivot *and* those that belong after it. These recursive calls are entirely independent, because they access separate sets of elements, and so are prime candidates for concurrent execution. Figure 8.2 shows such recursive division.

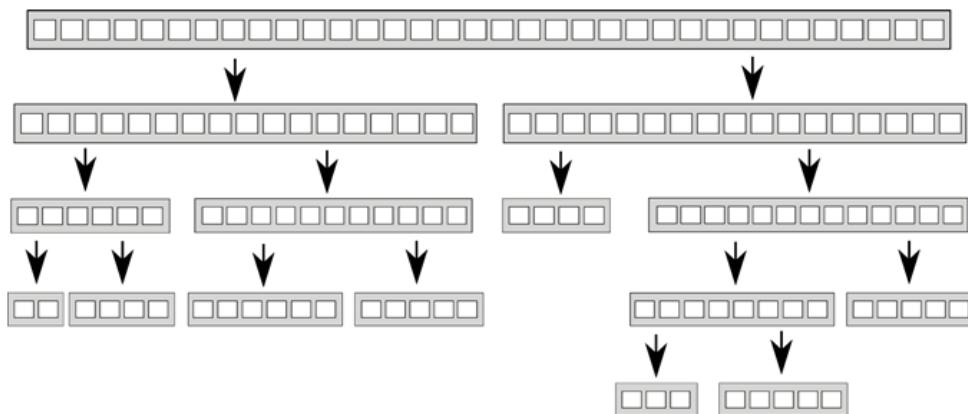


Figure 8.2 Recursively dividing data

In chapter 4, you saw such an implementation. Rather than just performing two recursive calls for the higher and lower chunks, you used `std::async()` to spawn asynchronous tasks

for the lower chunk at each stage. By using `std::async()`, you ask the C++ Thread Library to decide when to actually run the task on a new thread and when to run it synchronously.

This is important: if you're sorting a large set of data, spawning a new thread for each recursion would quickly result in a lot of threads. As you'll see when we look at performance, if you have too many threads, you might actually *slow down* the application. There's also a possibility of running out of threads if the data set is very large. The idea of dividing the overall task in a recursive fashion like this is a good one; you just need to keep a tighter rein on the number of threads. `std::async()` can handle this in simple cases, but it's not the only choice.

One alternative is to use the `std::thread::hardware_concurrency()` function to choose the number of threads, as you did with the parallel version of `accumulate()` from listing 2.9. Then, rather than starting a new thread for the recursive calls, you can just push the chunk to be sorted onto a thread-safe stack such as one of those described in chapters 6 and 7. If a thread has nothing else to do, either because it has finished processing all its chunks or because it's waiting for a chunk to be sorted, it can take a chunk from the stack and sort that.

The following listing shows a sample implementation that uses this technique. As with most of the examples, this is intended to demonstrate an idea rather than being production-ready code. If you're using C++17 compiler and library supports it, you're better off using the parallel algorithms provided by Standard Library, as covered in chapter 10.

Listing 8.1 Parallel Quicksort using a stack of pending chunks to sort

```
template<typename T>
struct sorter          #1
{
    struct chunk_to_sort
    {
        std::list<T> data;
        std::promise<std::list<T> > promise;
    };
    thread_safe_stack<chunk_to_sort> chunks;      #2
    std::vector<std::thread> threads;              #3
    unsigned const max_thread_count;
    std::atomic<bool> end_of_data;
    sorter():
        max_thread_count(std::thread::hardware_concurrency()-1),
        end_of_data(false)
    {}
    ~sorter()                                     #4
    {
        end_of_data=true;                         #5
        for(unsigned i=0;i<threads.size();++i)
        {
            threads[i].join();                   #6
        }
    }
    void try_sort_chunk()
    {
        boost::shared_ptr<chunk_to_sort> chunk=chunks.pop();   #7
        if(chunk)
        {
```

```

        sort_chunk(chunk);                                #8
    }
}
std::list<T> do_sort(std::list<T>& chunk_data)          #9
{
    if(chunk_data.empty())
    {
        return chunk_data;
    }
    std::list<T> result;
    result.splice(result.begin(),chunk_data,chunk_data.begin());
    T const& partition_val=result.begin();
    typename std::list<T>::iterator divide_point=           #10
        std::partition(chunk_data.begin(),chunk_data.end(),
                        [&](T const& val){return val<partition_val;});
    chunk_to_sort new_lower_chunk;
    new_lower_chunk.data.splice(new_lower_chunk.data.end(),
                                chunk_data,chunk_data.begin(),
                                divide_point);
    std::future<std::list<T> > new_lower=                  #11
        new_lower_chunk.promise.get_future();
    chunks.push(std::move(new_lower_chunk));      #12
    if(threads.size()<max_thread_count)           #12
    {
        threads.push_back(std::thread(&sorter<T>::sort_thread,this));
    }
    std::list<T> new_higher(do_sort(chunk_data));
    result.splice(result.end(),new_higher);
    while(new_lower.wait_for(std::chrono::seconds(0)) !=      #13
            std::future_status::ready)
    {
        try_sort_chunk();                                #14
    }
    result.splice(result.begin(),new_lower.get());
    return result;
}
void sort_chunk(boost::shared_ptr<chunk_to_sort > const& chunk)
{
    chunk->promise.set_value(do_sort(chunk->data));       #15
}
void sort_thread()
{
    while(!end_of_data)                                    #16
    {
        try_sort_chunk();                                #17
        std::this_thread::yield();                         #18
    }
}
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)      #19
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);                            #20
}

```

Here, the `parallel_quick_sort` function #19 delegates most of the functionality to the `sorter` class #1, which provides an easy way of grouping the stack of unsorted chunks #2 and the set of threads #3. The main work is done in the `do_sort` member function #9, which does the usual partitioning of the data #10. This time, rather than spawning a new thread for one chunk, it pushes it onto the stack #11 and spawns a new thread while you still have processors to spare #12. Because the lower chunk might be handled by another thread, you then have to wait for it to be ready #13. In order to help things along (in case you're the only thread or all the others are already busy), you try to process chunks from the stack on this thread while you're waiting #14. `try_sort_chunk` just pops a chunk off the stack #7 and sorts it #8, storing the result in the `promise`, ready to be picked up by the thread that posted the chunk on the stack #15.

Your freshly spawned threads sit in a loop trying to sort chunks off the stack #17 while the `end_of_data` flag isn't set #16. In between checking, they yield to other threads #18 to give them a chance to put some more work on the stack. This code relies on the destructor of your `sorter` class #4 to tidy up these threads. When all the data has been sorted, `do_sort` will return (even though the worker threads are still running), so your main thread will return from `parallel_quick_sort` #20 and thus destroy your `sorter` object. This sets the `end_of_data` flag #5 and waits for the threads to finish #6. Setting the flag terminates the loop in the `thread` function #16.

With this approach you no longer have the problem of unbounded threads that you have with a `spawn_task` that launches a new thread, and you're no longer relying on the C++ Thread Library to choose the number of threads for you, as it does with `std::async()`. Instead, you limit the number of threads to the value of `std::thread::hardware_concurrency()` in order to avoid excessive task switching. You do, however, have another potential problem: the management of these threads and the communication between them add quite a lot of complexity to the code. Also, although the threads are processing separate data elements, they all access the stack to add new chunks and to remove chunks for processing. This heavy contention can reduce performance, even if you use a lock-free (and hence nonblocking) stack, for reasons that you'll see shortly.

This approach is a specialized version of a *thread pool*—there's a set of threads that each take work to do from a list of pending work, do the work, and then go back to the list for more. Some of the potential problems with thread pools (including the contention on the work list) and ways of addressing them are covered in chapter 9. The problems of scaling your application to multiple processors are discussed in more detail later in this chapter (see section 8.2.1).

Both dividing the data before processing begins and dividing it recursively presume that the data itself is fixed beforehand, and you're just looking at ways of dividing it. This isn't always the case; if the data is dynamically generated or is coming from external input, this approach doesn't work. In this case, it might make more sense to divide the work by task type rather than dividing based on the data.

8.1.3 Dividing work by task type

Dividing work between threads by allocating different chunks of data to each thread (whether up front or recursively during processing) still rests on the assumption that the threads are going to be doing essentially the same work on each chunk of data. An alternative to dividing the work is to make the threads specialists, where each performs a distinct task, just as plumbers and electricians perform distinct tasks when building a house. Threads may or may not work on the same data, but if they do, it's for different purposes.

This is the sort of division of work that results from separating concerns with concurrency: each thread has a different task, which it carries out independently of other threads. Occasionally other threads may give it data or trigger events that it needs to handle, but in general each thread focuses on doing one thing well. In itself, this is basic good design; each piece of code should have a single responsibility.

DIVIDING WORK BY TASK TYPE TO SEPARATE CONCERNs

A single-threaded application has to handle conflicts with the single responsibility principle where there are multiple tasks that need to be run continuously over a period of time, or where the application needs to be able to handle incoming events (such as user key presses or incoming network data) in a timely fashion, even while other tasks are ongoing. In the single-threaded world you end up manually writing code that performs a bit of task A, performs a bit of task B, checks for key presses, checks for incoming network packets, and then loops back to perform another bit of task A. This means that the code for task A ends up being complicated by the need to save its state and return control to the main loop periodically. If you add too many tasks to the loop, things might slow down too far, and the user may find it takes too long to respond to the key press. I'm sure you've all seen the extreme form of this in action with some application or other: you set it doing some task, and the interface freezes until it has completed the task.

This is where threads come in. If you run each of the tasks in a separate thread, the operating system handles this for you. In the code for task A, you can focus on performing the task and not worry about saving state and returning to the main loop or how long you spend before doing so. The operating system will automatically save the state and switch to task B or C when appropriate, and if the target system has multiple cores or processors, tasks A and B may well be able to run truly concurrently. The code for handling the key press or network packet will now be run in a timely fashion, and everybody wins: the user gets timely responses, and you as developer have simpler code because each thread can focus on doing operations related directly to its responsibilities, rather than getting mixed up with control flow and user interaction.

That sounds like a nice, rosy vision. Can it really be like that? As with everything, it depends on the details. If everything really is independent, and the threads have no need to communicate with each other, then it really is this easy. Unfortunately, the world is rarely like that. These nice background tasks are often doing something that the user requested, and they need to let the user know when they're done by updating the user interface in some

manner. Alternatively, the user might want to cancel the task, which therefore requires the user interface to somehow send a message to the background task telling it to stop. Both these cases require careful thought and design and suitable synchronization, but the concerns are still separate. The user interface thread still just handles the user interface, but it might have to update it when asked to do so by other threads. Likewise, the thread running the background task still just focuses on the operations required for that task; it just happens that one of them is “allow task to be stopped by another thread.” In neither case do the threads care where the request came from, only that it was intended for them and relates directly to their responsibilities.

There are two big dangers with separating concerns with multiple threads. The first is that you’ll end up separating the *wrong* concerns. The symptoms to check for are that there is a lot of data shared between the threads or the different threads end up waiting for each other; both cases boil down to too much communication between threads. If this happens, it’s worth looking at the reasons for the communication. If all the communication relates to the same issue, maybe that should be the key responsibility of a single thread and extracted from all the threads that refer to it. Alternatively, if two threads are communicating a lot with each other but much less with other threads, maybe they should be combined into a single thread.

When dividing work across threads by task type, you don’t have to limit yourself to completely isolated cases. If multiple sets of input data require the same *sequence* of operations to be applied, you can divide the work so each thread performs one stage from the overall sequence.

DIVIDING A SEQUENCE OF TASKS BETWEEN THREADS

If your task consists of applying the same sequence of operations to many independent data items, you can use a *pipeline* to exploit the available concurrency of your system. This is by analogy to a physical pipeline: data flows in at one end through a series of operations (pipes) and out at the other end.

To divide the work this way, you create a separate thread for each stage in the pipeline—one thread for each of the operations in the sequence. When the operation is completed, the data element is put on a queue to be picked up by the next thread. This allows the thread performing the first operation in the sequence to start on the next data element while the second thread in the pipeline is working on the first element.

This is an alternative to just dividing the data between threads, as described in section 8.1.1, and is appropriate in circumstances where the input data itself isn’t all known when the operation is started. For example, the data might be coming in over a network, or the first operation in the sequence might be to scan a filesystem in order to identify files to process.

Pipelines are also good where each operation in the sequence is time consuming; by dividing the tasks between threads rather than the data, you change the performance profile. Suppose you have 20 data items to process, on four cores, and each data item requires four steps, which take 3 seconds each. If you divide the data between four threads, then each thread has 5 items to process. Assuming there’s no other processing that might affect the

timings, after 12 seconds you'll have 4 items processed, after 24 seconds 8 items processed, and so forth. All 20 items will be done after 1 minute. With a pipeline, things work differently. Your four steps can be assigned one to each processing core. Now the first item has to be processed by each core, so it still takes the full 12 seconds. Indeed, after 12 seconds you only have one item processed, which isn't as good as with the division by data. However, once the pipeline is *primed*, things proceed a bit differently; after the first core has processed the first item, it moves on to the second, so once the final core has processed the first item, it can perform its step on the second. You now get one item processed every 3 seconds rather than having the items processed in batches of four every 12 seconds.

The overall time to process the entire batch takes longer because you have to wait 9 seconds before the final core starts processing the first item. But smoother, more regular processing can be beneficial in some circumstances. Consider, for example, a system for watching high-definition digital videos. In order for the video to be watchable, you typically need at least 25 frames per second and ideally more. Also, the viewer needs these to be evenly spaced to give the impression of continuous movement; an application that can decode 100 frames per second is still no use if it pauses for a second, then displays 100 frames, then pauses for another second, and displays another 100 frames. On the other hand, viewers are probably happy to accept a delay of a couple of seconds when they *start* watching a video. In this case, parallelizing using a pipeline that outputs frames at a nice steady rate is probably preferable.

Having looked at various techniques for dividing the work between threads, let's take a look at the factors affecting the performance of a multithreaded system and how that can impact your choice of techniques.

8.2 Factors affecting the performance of concurrent code

If you're using concurrency in order to improve the performance of your code on systems with multiple processors, you need to know what factors are going to affect the performance. Even if you're just using multiple threads to separate concerns, you need to ensure that this doesn't adversely affect the performance. Customers won't thank you if your application runs *more slowly* on their shiny new 16-core machine than it did on their old single-core one.

As you'll see shortly, many factors affect the performance of multithreaded code—even something as simple as changing *which* data elements are processed by each thread (while keeping everything else identical) can have a dramatic effect on performance. So, without further ado, let's look at some of these factors, starting with the obvious one: how many processors does your target system have?

8.2.1 How many processors?

The number (and structure) of processors is the first big factor that affects the performance of a multithreaded application, and it's quite a crucial one. In some cases you do know exactly what the target hardware is and can thus design with this in mind, taking real measurements on the target system or an exact duplicate. If so, you're one of the lucky ones;

in general you don't have that luxury. You might be developing on a *similar* system, but the differences can be crucial. For example, you might be developing on a dual- or quad-core system, but your customers' systems may have one multicore processor (with any number of cores), or multiple single-core processors, or even multiple multicore processors. The behavior and performance characteristics of a concurrent program can vary considerably under such different circumstances, so you need to think carefully about what the impact may be and test things where possible.

To a first approximation, a single 16-core processor is the same as 4 quad-core processors or 16 single-core processors: in each case the system can run 16 threads concurrently. If you want to take advantage of this, your application must have at least 16 threads. If it has fewer than 16, you're leaving processor power on the table (unless the system is running other applications too, but we'll ignore that possibility for now). On the other hand, if you have more than 16 threads actually ready to run (and not blocked, waiting for something), your application will waste processor time switching between the threads, as discussed in chapter 1. When this happens, the situation is called *oversubscription*.

To allow applications to scale the number of threads in line with the number of threads the hardware can run concurrently, the C++11 Standard Thread Library provides `std::thread::hardware_concurrency()`. You've already seen how that can be used to scale the number of threads to the hardware.

Using `std::thread::hardware_concurrency()` directly requires care; your code doesn't take into account any of the other threads that are running on the system unless you explicitly share that information. In the worst case, if multiple threads call a function that uses `std::thread::hardware_concurrency()` for scaling at the same time, there will be huge oversubscription. `std::async()` avoids this problem because the library is aware of all calls and can schedule appropriately. Careful use of thread pools can also avoid this problem.

However, even if you take into account all threads running in your application, you're still subject to the impact of other applications running at the same time. Although the use of multiple CPU-intensive applications simultaneously is rare on single-user systems, there are some domains where it's more common. Systems designed to handle this scenario typically offer mechanisms to allow each application to choose an appropriate number of threads, although these mechanisms are outside the scope of the C++ Standard. One option is for a `std::async()`-like facility to take into account the total number of asynchronous tasks run by all applications when choosing the number of threads. Another is to limit the number of processing cores that can be used by a given application. I'd expect such a limit to be reflected in the value returned by `std::thread::hardware_concurrency()` on such platforms, although this isn't guaranteed. If you need to handle this scenario, consult your system documentation to see what options are available to you.

One additional twist to this situation is that the ideal algorithm for a problem can depend on the size of the problem compared to the number of processing units. If you have a *massively parallel* system with many processing units, an algorithm that performs more

operations overall may finish more quickly than one that performs fewer operations, because each processor performs only a few operations.

As the number of processors increases, so does the likelihood and performance impact of another problem: that of multiple processors trying to access the same data.

8.2.2 Data contention and cache ping-pong

If two threads are executing concurrently on different processors and they're both *reading* the same data, this usually won't cause a problem; the data will be copied into their respective caches, and both processors can proceed. However, if one of the threads *modifies* the data, this change then has to propagate to the cache on the other core, which takes time. Depending on the nature of the operations on the two threads, and the memory orderings used for the operations, such a modification may cause the second processor to stop in its tracks and wait for the change to propagate through the memory hardware. In terms of CPU instructions, this can be a *phenomenally* slow operation, equivalent to many hundreds of individual instructions, although the exact timing depends primarily on the physical structure of the hardware.

Consider the following simple piece of code:

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
    while(counter.fetch_add(1, std::memory_order_relaxed) < 100000000)
    {
        do_something();
    }
}
```

The counter is global, so any threads that call `processing_loop()` are modifying the same variable. Therefore, for each increment the processor must ensure it has an up-to-date copy of `counter` in its cache, modify the value, and publish it to other processors. Even though you're using `std::memory_order_relaxed`, so the compiler doesn't have to synchronize any other data, `fetch_add` is a read-modify-write operation and therefore needs to retrieve the most recent value of the variable. If another thread on another processor is running the same code, the data for `counter` must therefore be passed back and forth between the two processors and their corresponding caches so that each processor has the latest value for `counter` when it does the increment. If `do_something()` is short enough, or if there are too many processors running this code, the processors might actually find themselves *waiting* for each other; one processor is ready to update the value, but another processor is currently doing that, so it has to wait until the second processor has completed its update and the change has propagated. This situation is called *high contention*. If the processors rarely have to wait for each other, you have *low contention*.

In a loop like this one, the data for `counter` will be passed back and forth between the caches many times. This is called *cache ping-pong*, and it can seriously impact the performance of the application. If a processor stalls because it has to wait for a cache transfer,

it can't do *any* work in the meantime, even if there are other threads waiting that could do useful work, so this is bad news for the whole application.

You might think that this won't happen to you; after all, you don't have any loops like that. Are you sure? What about mutex locks? If you acquire a mutex in a loop, your code is similar to the previous code from the point of view of data accesses. In order to lock the mutex, another thread must transfer the data that makes up the mutex to its processor and modify it. When it's done, it modifies the mutex again to unlock it, and the mutex data has to be transferred to the next thread to acquire the mutex. This transfer time is *in addition* to any time that the second thread has to wait for the first to release the mutex:

```
std::mutex m;
my_data data;
void processing_loop_with_mutex()
{
    while(true)
    {
        std::lock_guard<std::mutex> lk(m);
        if(done_processing(data)) break;
    }
}
```

Now, here's the worst part: if the data and mutex really are accessed by more than one thread, then as you add more cores and processors to the system, it becomes *more likely* that you will get high contention and one processor having to wait for another. If you're using multiple threads to process the same data more quickly, the threads are competing for the data and thus competing for the same mutex. The more of them there are, the more likely they'll try to acquire the mutex at the same time, or access the atomic variable at the same time, and so forth.

The effects of contention with mutexes are usually different from the effects of contention with atomic operations for the simple reason that the use of a mutex naturally serializes threads at the operating system level rather than at the processor level. If you have enough threads ready to run, the operating system can schedule another thread to run while one thread is waiting for the mutex, whereas a processor stall prevents any threads from running on that processor. However, it will still impact the performance of those threads that *are* competing for the mutex; they can only run one at a time, after all.

Back in chapter 3, you saw how a rarely updated data structure can be protected with a single-writer, multiple-reader mutex (see section 3.3.2). Cache ping-pong effects can nullify the benefits of such a mutex if the workload is unfavorable, because all threads accessing the data (even reader threads) still have to modify the mutex itself. As the number of processors accessing the data goes up, the contention on the mutex itself increases, and the cache line holding the mutex must be transferred between cores, thus potentially increasing the time taken to acquire and release locks to undesirable levels. There are techniques to ameliorate this problem, essentially by spreading out the mutex across multiple cache lines, but unless you implement your own such mutex, you are subject to whatever your system provides.

If this cache ping-pong is bad, how can you avoid it? As you'll see later in the chapter, the answer ties in nicely with general guidelines for improving the potential for concurrency: do what you can to reduce the potential for two threads competing for the same memory location.

It's not quite that simple, though; things never are. Even if a particular memory location is only ever accessed by one thread, you can *still* get cache ping-pong due to an effect known as *false sharing*.

8.2.3 False sharing

Processor caches don't generally deal in individual memory locations; instead, they deal in blocks of memory called *cache lines*. These blocks of memory are typically 32 or 64 bytes in size, but the exact details depend on the particular processor model being used. Because the cache hardware only deals in cache-line-sized blocks of memory, small data items in adjacent memory locations will be in the same cache line. Sometimes this is good: if a set of data accessed by a thread is in the same cache line, this is better for the performance of the application than if the same set of data was spread over multiple cache lines. However, if the data items in a cache line are unrelated and need to be accessed by different threads, this can be a major cause of performance problems.

Suppose you have an array of `int` values and a set of threads that each access their own entry in the array but do so repeatedly, including updates. Since an `int` is typically much smaller than a cache line, quite a few of those array entries will be in the same cache line. Consequently, even though each thread only accesses its own array entry, the cache hardware *still* has to play cache ping-pong. Every time the thread accessing entry 0 needs to update the value, ownership of the cache line needs to be transferred to the processor running that thread, only to be transferred to the cache for the processor running the thread for entry 1 when that thread needs to update its data item. The cache line is shared, even though none of the data is, hence the term *false sharing*. The solution here is to structure the data so that data items to be accessed by the same thread are close together in memory (and thus more likely to be in the same cache line), whereas those that are to be accessed by separate threads are far apart in memory and thus more likely to be in separate cache lines. You'll see how this affects the design of the code and data later in this chapter.

If having multiple threads access data from the same cache line is bad, how does the memory layout of data accessed by a single thread affect things?

8.2.4 How close is your data?

Whereas false sharing is caused by having data accessed by one thread too close to data accessed by another thread, another pitfall associated with data layout directly impacts the performance of a single thread on its own. The issue is data proximity: if the data accessed by a single thread is spread out in memory, it's likely that it lies on separate cache lines. On the flip side, if the data accessed by a single thread is close together in memory, it's more likely to lie on the same cache line. Consequently, if data is spread out, more cache lines must be

loaded from memory onto the processor cache, which can increase memory access latency and reduce performance compared to data that's located close together.

Also, if the data is spread out, there's an increased chance that a given cache line containing data for the current thread also contains data that's *not* for the current thread. At the extreme there'll be more data in the cache that you don't care about than data that you do. This wastes precious cache space and thus increases the chance that the processor will experience a cache miss and have to fetch a data item from main memory even if it once held it in the cache, because it had to remove the item from the cache to make room for another.

Now, this is important with single-threaded code, so why am I bringing it up here? The reason is *task switching*. If there are more threads than cores in the system, each core is going to be running multiple threads. This increases the pressure on the cache, as you try to ensure that different threads are accessing different cache lines in order to avoid false sharing. Consequently, when the processor switches threads, it's more likely to have to reload the cache lines if each thread uses data spread across multiple cache lines than if each thread's data is close together in the same cache line.

If there are more threads than cores or processors, the operating system might also choose to schedule a thread on one core for one time slice and then on another core for the next time slice. This will therefore require transferring the cache lines for that thread's data from the cache for the first core to the cache for the second; the more cache lines that need transferring, the more time consuming this will be. Although operating systems typically avoid this when they can, it does happen and does impact performance when it happens.

Task-switching problems are particularly prevalent when lots of threads are *ready to run* as opposed to *waiting*. This is an issue we've already touched on: oversubscription.

8.2.5 Oversubscription and excessive task switching

In multithreaded systems, it's typical to have more threads than processors, unless you're running on *massively parallel* hardware. However, threads often spend time waiting for external I/O to complete or blocked on mutexes or waiting for condition variables and so forth, so this isn't a problem. Having the extra threads enables the application to perform useful work rather than having processors sitting idle while the threads wait.

This isn't always a good thing. If you have *too many* additional threads, there will be more threads *ready to run* than there are available processors, and the operating system will have to start task switching quite heavily in order to ensure they all get a fair time slice. As you saw in chapter 1, this can increase the overhead of the task switching as well as compound any cache problems resulting from lack of proximity. Oversubscription can arise when you have a task that repeatedly spawns new threads without limits, as the recursive quick sort from chapter 4 did, or where the natural number of threads when you separate by task type is more than the number of processors and the work is naturally CPU bound rather than I/O bound.

If you're simply spawning too many threads because of data division, you can limit the number of worker threads, as you saw in section 8.1.2. If the oversubscription is due to the natural division of work, there's not a lot you can do to ameliorate the problem save for

choosing a different division. In that case, choosing the appropriate division may require more knowledge of the target platform than you have available and is only worth doing if performance is unacceptable and it can be demonstrated that changing the division of work does improve performance.

Other factors can affect the performance of multithreaded code. The cost of cache ping-pong can vary quite considerably between two single-core processors and a single dual-core processor, even if they're the same CPU type and clock speed, for example, but these are the major ones that will have a very visible impact. Let's now look at how that affects the design of the code and data structures.

8.3 Designing data structures for multithreaded performance

In section 8.1 we looked at various ways of dividing work between threads, and in section 8.2 we looked at various factors that can affect the performance of your code. How can you use this information when designing data structures for multithreaded performance? This is a different question than that addressed in chapters 6 and 7, which were about designing data structures that are safe for concurrent access. As you've just seen in section 8.2, the layout of the data used by a single thread can have an impact, even if that data isn't shared with any other threads.

The key things to bear in mind when designing your data structures for multithreaded performance are *contention*, *false sharing*, and *data proximity*. All three of these can have a big impact on performance, and you can often improve things just by altering the data layout or changing which data elements are assigned to which thread. First off, let's look at an easy win: dividing array elements between threads.

8.3.1 Dividing array elements for complex operations

Suppose you're doing some heavy-duty math, and you need to multiply two large square matrices together. To multiply matrices, you multiply each element in the first *row* of the first matrix with the corresponding element of the first *column* of the second matrix and add up the products to give the top-left element of the result. You then repeat this with the second row and the first column to give the second element in the first column of the result, and with the first row and second column to give the first element in the second column of the result, and so forth. This is shown in figure 8.3; the highlighting shows that the second row of the first matrix is paired with the third column of the second matrix to give the entry in the second row of the third column of the result.

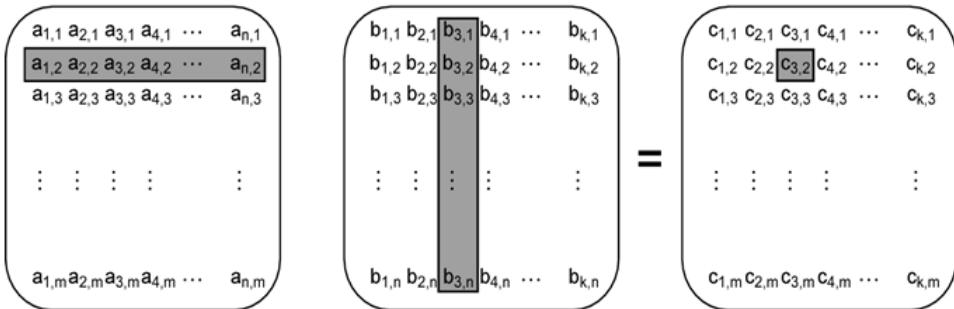


Figure 8.3 Matrix multiplication

Now let's assume that these are *large* matrices with several thousand rows and columns, in order to make it worthwhile using multiple threads to optimize the multiplication. Typically, a non-sparse matrix is represented by a big array in memory, with all the elements of the first row followed by all the elements of the second row, and so forth. To multiply your matrices you thus have three of these huge arrays. In order to get optimal performance, you need to pay careful attention to the data access patterns, particularly the writes to the third array.

There are many ways you can divide the work between threads. Assuming you have more rows/columns than available processors, you could have each thread calculate the values for a number of columns in the result matrix, or have each thread calculate the results for a number of rows, or even have each thread calculate the results for a rectangular subset of the matrix.

Back in sections 8.2.3 and 8.2.4, you saw that it's better to access contiguous elements from an array rather than values all over the place, because this reduces cache usage and the chance of false sharing. If you have each thread compute a set of columns, it needs to read every value from the first matrix and the values from the corresponding columns in the second matrix, but you only have to write the column values. Given that the matrices are stored with the rows contiguous, this means that you're accessing N elements from the first row, N elements from the second, and so forth (where N is the number of columns you're processing). Since other threads will be accessing the other elements of each row, it's clear that you ought to be accessing adjacent columns, so the N elements from each row are adjacent, and you minimize false sharing. Of course, if the space occupied by your N elements is an exact number of cache lines, there'll be no false sharing because threads will be working on separate cache lines.

On the other hand, if you have each thread compute a set of *rows*, then it needs to read every value from the *second* matrix and the values from the corresponding *rows* of the *first* matrix, but it only has to write the row values. Because the matrices are stored with the rows contiguous, you're now accessing *all* elements from N rows. If you again choose adjacent rows, this means that the thread is now the *only* thread writing to those N rows; it has a contiguous block of memory that's not touched by any other thread. This is likely an improvement over having each thread compute a set of columns, because the only possibility

of false sharing is for the last few elements of one block with the first few of the next, but it's worth timing it on the target architecture to confirm.

What about your third option—dividing into rectangular blocks? This can be viewed as dividing into columns and then dividing into rows. As such, it has the same false-sharing potential as division by columns. If you can choose the number of columns in the block to avoid this possibility, there's an advantage to rectangular division from the *read* side: you don't need to read the entirety of either source matrix. You only need to read the values corresponding to the rows and columns of the target rectangle. To look at this in concrete terms, consider multiplying two matrices that have 1,000 rows and 1,000 columns. That's 1 million elements. If you have 100 processors, they can compute 10 rows each for a nice round 10,000 elements. However, to calculate the results of those 10,000 elements, they need to access the entirety of the second matrix (1 million elements) plus the 10,000 elements from the corresponding rows in the first matrix, for a grand total of 1,010,000 elements. On the other hand, if they each compute a block of 100 elements by 100 elements (which is still 10,000 elements total), they need to access the values from 100 rows of the first matrix ($100 \times 1,000 = 100,000$ elements) and 100 columns of the second matrix (another 100,000). This is only 200,000 elements, which is a five-fold reduction in the number of elements read. If you're reading fewer elements, there's less chance of a cache miss and the potential for greater performance.

It may therefore be better to divide the result matrix into small square or almost-square blocks rather than have each thread compute the entirety of a small number of rows. Of course, you can adjust the size of each block at runtime, depending on the size of the matrices and the available number of processors. As ever, if performance is important, it's vital to profile various options on the target architecture, and check the literature relevant to the field — I make no claim that these are the only or best options if you really are doing matrix multiplication.

Chances are you're not doing matrix multiplication, so how does this apply to you? The same principles apply to any situation where you have large blocks of data to divide between threads; look at all the aspects of the data access patterns carefully, and identify the potential causes of performance hits. There may be similar circumstances in your problem domain where changing the division of work can improve performance without requiring any change to the basic algorithm.

OK, so we've looked at how access patterns in arrays can affect performance. What about other types of data structures?

8.3.2 Data access patterns in other data structures

Fundamentally, the same considerations apply when trying to optimize the data access patterns of other data structures as when optimizing access to arrays:

- Try to adjust the data distribution between threads so that data that's close together is worked on by the same thread.
- Try to minimize the data required by any given thread.

- Try to ensure that data accessed by separate threads is sufficiently far apart to avoid false sharing.

Of course, that's not easy to apply to other data structures. For example, binary trees are inherently difficult to subdivide in any unit other than a subtree, which may or may not be useful, depending on how balanced the tree is and how many sections you need to divide it into. Also, the nature of the trees means that the nodes are likely dynamically allocated and thus end up in different places on the heap.

Now, having data end up in different places on the heap isn't a particular problem in itself, but it does mean that the processor has to keep more things in cache. This can actually be beneficial. If multiple threads need to traverse the tree, then they all need to access the tree nodes, but if the tree nodes only contain *pointers* to the real data held at the node, then the processor only has to load the data from memory if it's actually needed. If the data is being modified by the threads that need it, this can avoid the performance hit of false sharing between the node data itself and the data that provides the tree structure.

There's a similar issue with data protected by a mutex. Suppose you have a simple class that contains a few data items and a mutex used to protect accesses from multiple threads. If the mutex and the data items are close together in memory, this is ideal for a thread that acquires the mutex; the data it needs may well already be in the processor cache, because it was just loaded in order to modify the mutex. But there's also a downside: if other threads try to lock the mutex while it's held by the first thread, they'll need access to that memory. Mutex locks are typically implemented as a read-modify-write atomic operation on a memory location within the mutex to try to acquire the mutex, followed by a call to the operating system kernel if the mutex is already locked. This read-modify-write operation may well cause the data held in the cache by the thread that owns the mutex to be invalidated. As far as the mutex goes, this isn't a problem; that thread isn't going to touch the mutex until it unlocks it. However, if the mutex shares a cache line with the data being used by the thread, the thread that owns the mutex can take a performance hit *because another thread tried to lock the mutex!*

One way to test whether this kind of false sharing is a problem is to add huge blocks of padding between the data elements that can be concurrently accessed by different threads. For example, you can use

```
struct protected_data
{
    std::mutex m;
    char padding[65536];      #A
    my_data data_to_protect;
};
```

to test the mutex contention issue or

```
struct my_data
{
    data_item1 d1;
    data_item2 d2;
    char padding[65536];
};
```

```
my_data some_array[256];  
#A 65536 bytes is orders of magnitude larger than a cache line
```

to test for false sharing of array data. If this improves the performance, you know that false sharing was a problem, and you can either leave the padding in or work to eliminate the false sharing in another way by rearranging the data accesses.

Of course, there's more than just the data access patterns to consider when designing for concurrency, so let's look at some of these additional considerations.

8.4 Additional considerations when designing for concurrency

So far in this chapter we've looked at ways of dividing work between threads, factors affecting performance, and how these factors affect your choice of data access patterns and data structures. There's more to designing code for concurrency than just that, though. You also need to consider things such as exception safety and scalability. Code is said to be *scalable* if the performance (whether in terms of reduced speed of execution or increased throughput) increases as more processing cores are added to the system. Ideally, the performance increase is linear, so a system with 100 processors performs 100 times better than a system with one processor.

Although code can work even if it isn't scalable—a single-threaded application is certainly not scalable, for example—exception safety is a matter of correctness. If your code isn't exception safe, you can end up with broken invariants or race conditions, or your application might terminate unexpectedly because an operation threw an exception. With this in mind, we'll look at exception safety first.

8.4.1 Exception safety in parallel algorithms

Exception safety is an essential aspect of good C++ code, and code that uses concurrency is no exception. In fact, parallel algorithms often require that you take more care with regard to exceptions than normal sequential algorithms. If an operation in a sequential algorithm throws an exception, the algorithm only has to worry about ensuring that it tidies up after itself to avoid resource leaks and broken invariants; it can merrily allow the exception to propagate to the caller for them to handle. By contrast, in a parallel algorithm many of the operations will be running on separate threads. In this case, the exception can't be allowed to propagate because it's on the wrong call stack. If a function spawned on a new thread exits with an exception, the application is terminated.

As a concrete example, let's revisit the `parallel_accumulate` function from listing 2.9, which is reproduced here.

Listing 8.2 A naïve parallel version of `std::accumulate` (from listing 2.9)

```
template<typename Iterator,typename T>
struct accumulate_block
{
    void operator()(Iterator first,Iterator last,T& result)
```

```

    {
        result=std::accumulate(first,last,result);      #1
    }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);      #2
    if(!length)
        return init;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;
    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();
    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;
    std::vector<T> results(num_threads);                  #3
    std::vector<std::thread> threads(num_threads-1);       #4
    Iterator block_start=first;                           #5
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;                  #6
        std::advance(block_end,block_size);
        threads[i]=std::thread(                      #7
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;                          #8
    }
    accumulate_block<Iterator,T>()(                #9
        block_start,last,results[num_threads-1]);
    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));
    return std::accumulate(results.begin(),results.end(),init);   #10
}

```

Now let's go through and identify the places where an exception can be thrown: basically anywhere where you call a function you know can throw or you perform an operation on a user-defined type that may throw.

First up, you have the call to `distance` #2, which performs operations on the user-supplied iterator type. Because you haven't yet done any work, and this is on the calling thread, it's fine. Next up, you have the allocation of the `results` vector #3 and the `threads` vector #4. Again, these are on the calling thread, and you haven't done any work or spawned any threads, so this is fine. Of course, if the construction of `threads` throws, the memory allocated for `results` will have to be cleaned up, but the destructor will take care of that for you.

Skipping over the initialization of `block_start` #5 because that's similarly safe, you come to the operations in the thread-spawning loop #6, #7, #8. Once you've been through the creation of the first thread at #7, you're in trouble if you throw any exceptions; the destructors of your new `std::thread` objects will call `std::terminate` and abort your program. This isn't a good place to be.

The call to `accumulate_block` #9 can potentially throw, with similar consequences; your thread objects will be destroyed and call `std::terminate`. On the other hand, the final call to `std::accumulate` #10 can throw without causing any hardship, because all the threads have been joined by this point.

That's it for the main thread, but there's more: the calls to `accumulate_block` on the new threads might throw at #1. There aren't any catch blocks, so this exception will be left unhandled and cause the library to call `std::terminate()` to abort the application.

In case it's not glaringly obvious, *this code isn't exception-safe*.

ADDING EXCEPTION SAFETY

OK, so we've identified all the possible throw points and the nasty consequences of exceptions. What can you do about it? Let's start by addressing the issue of the exceptions thrown on your new threads.

You encountered the tool for this job in chapter 4. If you look carefully at what you're trying to achieve with new threads, it's apparent that you're trying to calculate a result to return while allowing for the possibility that the code might throw an exception. This is precisely what the combination of `std::packaged_task` and `std::future` is designed for. If you rearrange your code to use `std::packaged_task`, you end up the following code.

Listing 8.3 A parallel version of `std::accumulate` using `std::packaged_task`

```
template<typename Iterator,typename T>
struct accumulate_block
{
    T operator()(Iterator first,Iterator last)      #1
    {
        return std::accumulate(first,last,T());
    }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return init;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;
    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();
    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;
    std::vector<std::future<T> > futures(num_threads-1);      #3
    std::vector<std::thread> threads(num_threads-1);
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_task<T(Iterator,Iterator)> task(      #4
```

```

        accumulate_block<Iterator,T>());
futures[i]=task.get_future();                                #5
threads[i]=std::thread(std::move(task),block_start,block_end); #6
block_start=block_end;
}
T last_result=accumulate_block<Iterator,T>()(block_start,last);    #7
std::for_each(threads.begin(),threads.end(),
    std::mem_fn(&std::thread::join));
T result=init;                                         #8
for(unsigned long i=0;i<(num_threads-1);++i)
{
    result+=futures[i].get();                         #9
}
result += last_result;                                #10
return result;
}

```

The first change is that the function call operator of `accumulate_block` now returns the result directly, rather than taking a reference to somewhere to store it #1. You're using `std::packaged_task` and `std::future` for the exception safety, so you can use it to transfer the result too. This does require that you explicitly pass in a default-constructed `T` in the call to `std::accumulate` #2 rather than reusing the supplied `result` value, but that's a minor change.

The next change is that rather than having a vector of results, you have a vector of `futures` #3 to store a `std::future<T>` for each spawned thread. In the thread-spawning loop, you first create a task for `accumulate_block` #4. `std::packaged_task<T(Iterator,` `Iterator)>` declares a task that takes two `Iterators` and returns a `T`, which is what your function does. You then get the future for that task #5 and run that task on a new thread, passing in the start and end of the block to process #6. When the task runs, the result will be captured in the future, as will any exception thrown.

Since you've been using futures, you don't have a result array, so you must store the result from the final block in a variable #7 rather than in a slot in the array. Also, because you have to get the values out of the futures, it's now simpler to use a basic `for` loop rather than `std::accumulate`, starting with the supplied initial value #8 and adding in the result from each future #9. If the corresponding task threw an exception, this will have been captured in the future and will now be thrown again by the call to `get()`. Finally, you add the result from the last block #10 before returning the overall result to the caller.

So, that's removed one of the potential problems: exceptions thrown in the worker threads are rethrown in the main thread. If more than one of the worker threads throws an exception, only one will be propagated, but that's not too big a deal. If it really matters, you can use something like `std::nested_exception` to capture all the exceptions and throw that instead.

The remaining problem is the leaking threads if an exception is thrown between when you spawn the first thread and when you've joined with them all. The simplest solution is just to catch any exceptions, join with the threads that are still `joinable()`, and rethrow the exception:

```
try
```

```

    {
        for(unsigned long i=0;i<(num_threads-1);++i)
        {
            // ... as before
        }
        T last_result=accumulate_block<Iterator,T>()(block_start,last);
        std::for_each(threads.begin(),threads.end(),
                     std::mem_fn(&std::thread::join));
    }
catch(...)
{
    for(unsigned long i=0;i<(num_thread-1);++i)
    {
        if(threads[i].joinable())
            thread[i].join();
    }
    throw;
}

```

Now this works. All the threads will be joined, no matter how the code leaves the block. However, try-catch blocks are ugly, and you have duplicate code. You're joining the threads both in the "normal" control flow *and* in the catch block. Duplicate code is rarely a good thing, because it means more places to change. Instead, let's extract this out into the destructor of an object; it is, after all, the idiomatic way of cleaning up resources in C++. Here's your class:

```

class join_threads
{
    std::vector<std::thread>& threads;
public:
    explicit join_threads(std::vector<std::thread>& threads_):
        threads(threads_)
    {}
    ~join_threads()
    {
        for(unsigned long i=0;i<threads.size();++i)
        {
            if(threads[i].joinable())
                threads[i].join();
        }
    }
};

```

This is similar to your `thread_guard` class from listing 2.3, except it's extended for the whole vector of threads. You can then simplify your code as follows.

Listing 8.4 An exception-safe parallel version of `std::accumulate`

```

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return init;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;

```

```

unsigned long const hardware_threads=
    std::hardware_concurrency();
unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);
unsigned long const block_size=length/num_threads;
std::vector<std::future<T>> futures(num_threads-1);
std::vector<std::thread> threads(num_threads-1);
join_threads joiner(threads);                                #1
Iterator block_start=first;
for(unsigned long i=0;i<(num_threads-1);++i)
{
    Iterator block_end=block_start;
    std::advance(block_end,block_size);
    std::packaged_task<T(Iterator,Iterator)> task(
        accumulate_block<Iterator,T>());
    futures[i]=task.get_future();
    threads[i]=std::thread(std::move(task),block_start,block_end);
    block_start=block_end;
}
T last_result=accumulate_block<Iterator,T>()(block_start,last);
T result=init;
for(unsigned long i=0;i<(num_threads-1);++i)
{
    result+=futures[i].get();                                #2
}
result += last_result;
return result;
}

```

Once you've created your container of threads, you create an instance of your new class #1 to join with all the threads on exit. You can then remove your explicit join loop, safe in the knowledge that the threads will be joined however the function exits. Note that the calls to `futures[i].get()` #2 will block until the results are ready, so you don't need to have explicitly joined with the threads at this point. This is unlike the original from listing 8.2, where you needed to have joined with the threads to ensure that the `results` vector was correctly populated. Not only do you get exception-safe code, but your function is actually shorter because you've extracted the join code into your new (reusable) class.

EXCEPTION SAFETY WITH STD::ASYNC()

Now that you've seen what's required for exception safety when explicitly managing the threads, let's take a look at the same thing done with `std::async()`. As you've already seen, in this case the library takes care of managing the threads for you, and any threads spawned are completed when the future is *ready*. The key thing to note for exception safety is that if you destroy the future without waiting for it, the destructor will wait for the thread to complete. This neatly avoids the problem of leaked threads that are still executing and holding references to the data. The next listing shows an exception-safe implementation using `std::async()`.

Listing 8.5 An exception-safe parallel version of `std::accumulate` using `std::async`

```
template<typename Iterator,typename T>
```

```

T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);      #1
    unsigned long const max_chunk_size=25;
    if(length<=max_chunk_size)
    {
        return std::accumulate(first,last,init);                #2
    }
    else
    {
        Iterator mid_point=first;
        std::advance(mid_point,length/2);                      #3
        std::future<T> first_half_result=
            std::async(parallel_accumulate<Iterator,T>,
                       first,mid_point,init);                     #4
        T second_half_result=parallel_accumulate(mid_point,last,T());  #5
        return first_half_result.get()+second_half_result;        #6
    }
}

```

This version uses recursive division of the data rather than pre-calculating the division of the data into chunks, but it's a whole lot simpler than the previous version, and it's *still exception safe*. As before, you start by finding the length of the sequence #1, and if it's smaller than the maximum chunk size, you resort to calling `std::accumulate` directly #2. If there are more elements than your chunk size, you find the midpoint #3 and then spawn an asynchronous task to handle that half #4. The second half of the range is handled with a direct recursive call #5, and then the results from the two chunks are added together #6. The library ensures that the `std::async` calls make use of the hardware threads that are available without creating an overwhelming number of threads. Some of the "asynchronous" calls will actually be executed synchronously in the call to `get()` #6.

The beauty of this is that not only can it take advantage of the hardware concurrency, but it's also trivially exception safe. If an exception is thrown by the recursive call #5, the future created from the call to `std::async` #4 will be destroyed as the exception propagates. This will in turn wait for the asynchronous task to finish, thus avoiding a dangling thread. On the other hand, if the asynchronous call throws, this is captured by the future, and the call to `get()` #6 will rethrow the exception.

What other considerations do you need to take into account when designing concurrent code? Let's look at *scalability*. How much does the performance improve if you move your code to a system with more processors?

8.4.2 Scalability and Amdahl's law

Scalability is all about ensuring that your application can take advantage of additional processors in the system it's running on. At one extreme you have a single-threaded application that's completely unscalable; even if you add 100 processors to your system, the performance will remain unchanged. At the other extreme you have something like the SETI@Home³ project, which is designed to take advantage of thousands of additional

³ <http://setiathome.ssl.berkeley.edu/>

processors (in the form of individual computers added to the network by users) as they become available.

For any given multithreaded program, the number of threads that are performing useful work will vary as the program runs. Even if every thread is doing useful work for the entirety of its existence, the application may initially have only one thread, which will then have the task of spawning all the others. But even that's a highly unlikely scenario. Threads often spend time waiting for each other or waiting for I/O operations to complete.

Every time one thread has to wait for something (whatever that something is), unless there's another thread ready to take its place on the processor, you have a processor sitting idle that could be doing useful work.

A simplified way of looking at this is to divide the program into "serial" sections where only one thread is doing any useful work and "parallel" sections where all the available processors are doing useful work. If you run your application on a system with more processors, the "parallel" sections will theoretically be able to complete more quickly, because the work can be divided between more processors, whereas the "serial" sections will remain serial. Under such a simplified set of assumptions, you can therefore estimate the potential performance gain to be achieved by increasing the number of processors: if the "serial" sections constitute a fraction f_s of the program, then the performance gain P from using N processors can be estimated as

$$P = \frac{1}{f_s + \frac{1 - f_s}{N}}$$

This is *Amdahl's law*, which is often cited when talking about the performance of concurrent code. If everything can be parallelized, so the serial fraction is 0, the speedup is simply N . Alternatively, if the serial fraction is one third, even with an infinite number of processors you're not going to get a speedup of more than 3.

However, this paints a naive picture, because tasks are rarely infinitely divisible in the way that would be required for the equation to hold, and it's also rare for everything to be CPU bound in the way that's assumed. As you've just seen, threads may wait for many things while executing.

One thing that's clear from Amdahl's law is that when you're using concurrency for performance, it's worth looking at the overall design of the application to maximize the potential for concurrency and ensure that there's always useful work for the processors to be doing. If you can reduce the size of the "serial" sections or reduce the potential for threads to wait, you can improve the potential for performance gains on systems with more processors. Alternatively, if you can provide more data for the system to process, and thus keep the parallel sections primed with work, you can reduce the serial fraction and increase the performance gain P .

Essentially, scalability is about *reducing the time it takes to perform an action or increasing the amount of data that can be processed in a given time* as more processors are added. Sometimes these are equivalent (you can process more data if each element is processed faster) but not always. Before choosing the techniques to use for dividing work between threads, it's important to identify which of these aspects of scalability are important to you.

I mentioned at the beginning of this section that threads don't always have useful work to do. Sometimes they have to wait for other threads, or for I/O to complete, or for something else. If you give the system something useful to do during this wait, you can effectively "hide" the waiting.

8.4.3 Hiding latency with multiple threads

For lots of the discussions of the performance of multithreaded code, we've been assuming that the threads are running "flat out" and always have useful work to do when they're actually running on a processor. This is of course not true; in application code threads frequently block while waiting for something. For example, they may be waiting for some I/O to complete, waiting to acquire a mutex, waiting for another thread to complete some operation and notify a condition variable or populate a future, or even just sleeping for a period of time.

Whatever the reason for the waits, if you have only as many threads as there are physical processing units in the system, having blocked threads means you're wasting CPU time. The processor that would otherwise be running a blocked thread is instead doing nothing. Consequently, if you know that one of your threads is likely to spend a considerable portion of its time waiting around, you can make use of that spare CPU time by running one or more additional threads.

Consider a virus scanner application, which divides the work across threads using a pipeline. The first thread searches the filesystem for files to check and puts them on a queue. Meanwhile, another thread takes filenames from the queue, loads the files, and scans them for viruses. You know that the thread searching the filesystem for files to scan is definitely going to be I/O bound, so you make use of the "spare" CPU time by running an additional scanning thread. You'd then have one file-searching thread and as many scanning threads as there are physical cores or processors in the system. Since the scanning thread may also have to read significant portions of the files off the disk in order to scan them, it might make sense to have even more scanning threads. But at some point there'll be too many threads, and the system will slow down again as it spends more and more time task switching, as described in section 8.2.5.

As ever, this is an optimization, so it's important to measure performance before and after any change in the number of threads; the optimal number of threads will be highly dependent on the nature of the work being done and the percentage of time the thread spends waiting.

Depending on the application, it might be possible to use up this spare CPU time without running additional threads. For example, if a thread is blocked because it's waiting for an I/O operation to complete, it might make sense to use asynchronous I/O if that's available, and

then the thread can perform other useful work while the I/O is performed in the background. In other cases, if a thread is waiting for another thread to perform an operation, then rather than blocking, the waiting thread might be able to perform that operation itself, as you saw with the lock-free queue in chapter 7. In an extreme case, if a thread is waiting for a task to be completed and that task hasn't yet been started by any thread, the waiting thread might perform the task in entirety itself or another task that's incomplete. You saw an example of this in listing 8.1, where the sort function repeatedly tries to sort outstanding chunks as long as the chunks it needs are not yet sorted.

Rather than adding threads to ensure that all available processors are being used, sometimes it pays to add threads to ensure that external events are handled in a timely manner, to increase the *responsiveness* of the system.

8.4.4 Improving responsiveness with concurrency

Most modern graphical user interface frameworks are *event driven*; the user performs actions on the user interface by pressing keys or moving the mouse, which generate a series of events or messages that the application then handles. The system may also generate messages or events on its own. In order to ensure that all events and messages are correctly handled, the application typically has an event loop that looks like this:

```
while(true)
{
    event_data event=get_event();
    if(event.type==quit)
        break;
    process(event);
}
```

Obviously, the details of the API will vary, but the structure is generally the same: wait for an event, do whatever processing is necessary to handle it, and then wait for the next one. If you have a single-threaded application, this can make long-running tasks hard to write, as described in section 8.1.3. In order to ensure that user input is handled in a timely manner, `get_event()` and `process()` must be called with reasonable frequency, whatever the application is doing. This means that either the task must periodically suspend itself and return control to the event loop, or the `get_event()/process()` code must be called from within the code at convenient points. Either option complicates the implementation of the task.

By separating the concerns with concurrency, you can put the lengthy task on a whole new thread and leave a dedicated GUI thread to process the events. The threads can then communicate through simple mechanisms rather than having to somehow mix the event-handling code in with the task code. The following listing shows a simple outline for such a separation.

Listing 8.6 Separating GUI thread from task thread

```
std::thread task_thread;
std::atomic<bool> task_cancelled(false);
void gui_thread()
```

```

{
    while(true)
    {
        event_data event=get_event();
        if(event.type==quit)
            break;
        process(event);
    }
}
void task()
{
    while(!task_complete() && !task_cancelled)
    {
        do_next_operation();
    }
    if(task_cancelled)
    {
        perform_cleanup();
    }
    else
    {
        post_gui_event(task_complete);
    }
}
void process(event_data const& event)
{
    switch(event.type)
    {
    case start_task:
        task_cancelled=false;
        task_thread=std::thread(task);
        break;
    case stop_task:
        task_cancelled=true;
        task_thread.join();
        break;
    case task_complete:
        task_thread.join();
        display_results();
        break;
    default:
        //...
    }
}

```

By separating the concerns in this way, the user thread is always able to respond to the events in a timely fashion, even if the task takes a long time. This *responsiveness* is often key to the user experience when using an application; applications that completely lock up whenever a particular operation is being performed (whatever that may be) are inconvenient to use. By providing a dedicated event-handling thread, the GUI can handle GUI-specific messages (such as resizing or repainting the window) without interrupting the execution of the time-consuming processing, while still passing on the relevant messages where they *do* affect the long-running task.

So far in this chapter you've had a thorough look at the issues that need to be considered when designing concurrent code. Taken as a whole, these can be quite overwhelming, but as

you get used to working with your “multithreaded programming hat” on, most of them will become second nature. If these considerations are new to you, hopefully they’ll become clearer as you look at how they impact some concrete examples of multithreaded code.

8.5 Designing concurrent code in practice

When designing concurrent code for a particular task, the extent to which you’ll need to consider each of the issues described previously will depend on the task. To demonstrate how they apply, we’ll look at the implementation of parallel versions of three functions from the C++ Standard Library. This will give you a familiar basis on which to build, while providing a platform for looking at the issues. As a bonus, we’ll also have usable implementations of the functions, which could be used to help with parallelizing a larger task.

I’ve primarily selected these implementations to demonstrate particular techniques rather than to be state-of-the-art implementations; more advanced implementations that make better use of the available hardware concurrency may be found in the academic literature on parallel algorithms or in specialist multithreading libraries such as Intel’s Threading Building Blocks.⁴

The simplest parallel algorithm conceptually is a parallel version of `std::for_each`, so we’ll start with that.

8.5.1 A parallel implementation of `std::for_each`

`std::for_each` is simple in concept; it calls a user-supplied function on every element in a range in turn. The big difference between a parallel implementation and the sequential `std::for_each` is the order of the function calls. `std::for_each` calls the function with the first element in the range, then the second, and so on, whereas with a parallel implementation there’s no guarantee as to the order in which the elements will be processed, and they may (indeed we hope they *will*) be processed concurrently.

To implement a parallel version of this, you just need to divide the range into sets of elements to process on each thread. You know the number of elements in advance, so you can divide the data before processing begins (section 8.1.1). We’ll assume that this is the only parallel task running, so you can use `std::thread::hardware_concurrency()` to determine the number of threads. You also know that the elements can be processed entirely independently, so you can use contiguous blocks to avoid false sharing (section 8.2.3).

This algorithm is similar in concept to the parallel version of `std::accumulate` described in section 8.4.1, but rather than computing the sum of each element, you merely have to apply the specified function. Although you might imagine this would greatly simplify the code, because there’s no result to return, if you wish to pass on exceptions to the caller, you still need to use the `std::packaged_task` and `std::future` mechanisms to transfer the exception between threads. A sample implementation is shown here.

⁴ <http://threadingbuildingblocks.org/>

Listing 8.7 A parallel version of std::for_each

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;
    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();
    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;
    std::vector<std::future<void>> futures(num_threads-1);      #1
    std::vector<std::thread> threads(num_threads-1);
    join_threads joiner(threads);
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_task<void(void)> task();                      #2
        [=]()
        {
            std::for_each(block_start,block_end,f);
        });
        futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task));                     #3
        block_start=block_end;
    }
    std::for_each(block_start,last,f);
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        futures[i].get();                                         #4
    }
}
```

The basic structure of the code is identical to that of listing 8.4, which is unsurprising. The key difference is that the `futures` vector stores `std::future<void>` #1 because the worker threads don't return a value, and a simple lambda function that invokes the function `f` on the range from `block_start` to `block_end` is used for the task #2. This avoids having to pass the range into the thread constructor #3. Since the worker threads don't return a value, the calls to `futures[i].get()` #4 just provide a means of retrieving any exceptions thrown on the worker threads; if you don't wish to pass on the exceptions, you could omit this.

Just as your parallel implementation of `std::accumulate` could be simplified using `std::async`, so can your `parallel_for_each`. Such an implementation follows.

Listing 8.8 A parallel version of std::for_each using std::async

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
```

```

unsigned long const length=std::distance(first,last);
if(!length)
    return;
unsigned long const min_per_thread=25;
if(length<(2*min_per_thread))
{
    std::for_each(first,last,f);                                #1
}
else
{
    Iterator const mid_point=first+length/2;
    std::future<void> first_half=                           #2
        std::async(&parallel_for_each<Iterator,Func>,
                    first,mid_point,f);
    parallel_for_each(mid_point,last,f);                      #3
    first_half.get();                                         #4
}
}

```

As with your `std::async`-based `parallel_accumulate` from listing 8.5, you split the data recursively rather than before execution, because you don't know how many threads the library will use. As before, you divide the data in half at each stage, running one half asynchronously #2 and the other directly #3 until the remaining data is too small to be worth dividing, in which case you defer to `std::for_each` #1. Again, the use of `std::async` and the `get()` member function of `std::future` #4 provides the exception propagation semantics.

Let's move on from algorithms that must perform the same operation on each element (of which there are several: `std::count` and `std::replace` spring to mind for starters) to a slightly more complicated example in the shape of `std::find`.

8.5.2 A parallel implementation of `std::find`

`std::find` is a useful algorithm to consider next, because it's one of several algorithms that can complete without every element having been processed. For example, if the first element in the range matches the search criterion, there's no need to examine any other elements. As you'll see shortly, this is an important property for performance, and it has direct consequences for the design of the parallel implementation. It's a particular example of how data access patterns can affect the design of your code (section 8.3.2). Other algorithms in this category include `std::equal` and `std::any_of`.

If you were searching for an old photograph through the boxes of keepsakes in your attic with your wife or partner, you wouldn't let them continue searching if you found the photograph. Instead, you'd let them know you'd found the photograph (perhaps by shouting, "Found it!"), so that they could stop searching and move on to something else. The nature of many algorithms requires that they process every element, so they have no equivalent to shouting, "Found it!" For algorithms such as `std::find` the ability to complete "early" is an important property and not something to squander. You therefore need to design your code to make use of it—to interrupt the other tasks in some way when the answer is known, so that the code doesn't have to wait for the other worker threads to process the remaining elements.

If you don't interrupt the other threads, the serial version may well outperform your parallel implementation, because the serial algorithm can just stop searching and return once a match is found. If, for example, the system can support four concurrent threads, each thread will have to examine one quarter of the elements in the range, and our naïve parallel implementation would thus take approximately one quarter of the time a single thread would take to check every element. If the matching element lies in the first quarter of the range, the sequential algorithm will return first, because it doesn't need to check the remainder of the elements.

One way in which you can interrupt the other threads is by making use of an atomic variable as a flag and checking the flag after processing every element. If the flag is set, one of the other threads has found a match, so you can cease processing and return. By interrupting the threads in this way, you preserve the property that you don't have to process every value and thus improve the performance compared to the serial version in more circumstances. The downside to this is that atomic loads can be slow operations, so this can impede the progress of each thread.

Now you have two choices as to how to return the values and how to propagate any exceptions. You can use an array of futures, use `std::packaged_task` for transferring the values and exceptions, and then process the results back in the main thread; or you can use `std::promise` to set the final result directly from the worker threads. It all depends on how you wish to handle exceptions from the worker threads. If you want to stop on the first exception (even if you haven't processed all elements), you can use `std::promise` to set both the value and the exception. On the other hand, if you want to allow the other workers to keep searching, you can use `std::packaged_task`, store all the exceptions, and then rethrow one of them if a match isn't found.

In this case I've opted to use `std::promise` because the behavior matches that of `std::find` more closely. One thing to watch out for here is the case where the element being searched for isn't in the supplied range. You therefore need to wait for all the threads to finish *before* getting the result from the future. If you just block on the future, you'll be waiting forever if the value isn't there. The result is shown here.

Listing 8.9 An implementation of a parallel find algorithm

```
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    struct find_element                                #1
    {
        void operator()(Iterator begin,Iterator end,
                          MatchType match,
                          std::promise<Iterator>* result,
                          std::atomic<bool>* done_flag)
        {
            try
            {
                for(;(begin!=end) && !done_flag->load();++begin)      #2
                {
                    if(*begin==match)
```

```

    {
        result->set_value(begin);                      #3
        done_flag->store(true);                      #4
        return;
    }
}
catch(...)                                     #5
{
    try
    {
        result->set_exception(std::current_exception());  #6
        done_flag->store(true);
    }
    catch(...)                                     #7
    {}
}
};

unsigned long const length=std::distance(first,last);
if(!length)
    return last;
unsigned long const min_per_thread=25;
unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;
unsigned long const hardware_threads=
    std::thread::hardware_concurrency();
unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);
unsigned long const block_size=length/num_threads;
std::promise<Iterator> result;                  #8
std::atomic<bool> done_flag(false);              #9
std::vector<std::thread> threads(num_threads-1); #10
{
    join_threads joiner(threads);
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        threads[i]=std::thread(find_element(),
                               block_start,block_end,match,
                               &result,&done_flag);
        block_start=block_end;
    }
    find_element()(block_start,last,match,&result,&done_flag); #12
}
if(!done_flag.load())                           #13
{
    return last;
}
return result.get_future().get();                #14
}

```

The main body of listing 8.9 is similar to the previous examples. This time, the work is done in the function call operator of the local `find_element` class #1. This loops through the

elements in the block it's been given, checking the flag at each step #2. If a match is found, it sets the final result value in the promise #3 and then sets the `done_flag` #4 before returning.

If an exception is thrown, this is caught by the catchall handler #5, and you try to store the exception in the promise #6 before setting the `done_flag`. Setting the value on the promise might throw an exception if the promise is already set, so you catch and discard any exceptions that happen here #7.

This means that if a thread calling `find_element` either finds a match or throws an exception, all other threads will see `done_flag` set and will stop. If multiple threads find a match or throw at the same time, they'll race to set the result in the promise. But this is a benign race condition; whichever succeeds is therefore nominally "first" and is therefore an acceptable result.

Back in the main `parallel_find` function itself, you have the promise #8 and flag #9 used to stop the search, both of which are passed in to the new threads along with the range to search #11. The main thread also uses `find_element` to search the remaining elements #12. As already mentioned, you need to wait for all threads to finish before you check the result, because there might not be *any* matching elements. You do this by enclosing the thread launching-and-joining code in a block #10, so all threads are joined when you check the flag to see whether a match was found #13. If a match was found, you can get the result or throw the stored exception by calling `get()` on the `std::future<Iterator>` you can get from the promise #14.

Again, this implementation assumes that you're going to be using all available hardware threads or that you have some other mechanism to determine the number of threads to use for the up-front division of work between threads. Just as before, you can use `std::async` and recursive data division to simplify your implementation, while using the automatic scaling facility of the C++ Standard Library. An implementation of `parallel_find` using `std::async` is shown in the following listing.

Listing 8.10 An implementation of a parallel find algorithm using `std::async`

```
template<typename Iterator,typename MatchType> #1
Iterator parallel_find_Impl(Iterator first,Iterator last,MatchType match,
                           std::atomic<bool>& done)
{
    try
    {
        unsigned long const length=std::distance(first,last);
        unsigned long const min_per_thread=25;          #2
        if(length<(2*min_per_thread))                  #3
        {
            for(;(first!=last) && !done.load();++first) #4
            {
                if(*first==match)
                {
                    done=true;                         #5
                    return first;
                }
            }
        }
        return last;                                  #6
    }
```

```

    }
    else
    {
        Iterator const mid_point=first+(length/2);      #7
        std::future<Iterator> async_result=
            std::async(&parallel_find_impl<Iterator,MatchType>,
                      mid_point,last,match,std::ref(done));
        Iterator const direct_result=
            parallel_find_impl(first,mid_point,match,done);  #9
        return (direct_result==mid_point)?
            async_result.get():direct_result;                #10
    }
}
catch(...)
{
    done=true;                                         #11
    throw;
}
}
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    std::atomic<bool> done(false);
    return parallel_find_impl(first,last,match,done);      #12
}

```

The desire to finish early if you find a match means that you need to introduce a flag that is shared between all threads to indicate that a match has been found. This therefore needs to be passed in to all recursive calls. The simplest way to achieve this is by delegating to an implementation function #1 that takes an additional parameter—a reference to the done flag, which is passed in from the main entry point #12.

The core implementation then proceeds along familiar lines. In common with many of the implementations here, you set a minimum number of items to process on a single thread #2; if you can't cleanly divide into two halves of at least that size, you run everything on the current thread #3. The actual algorithm is a simple loop through the specified range, looping until you reach the end of the range or the done flag is set #4. If you do find a match, the done flag is set before returning #5. If you stop searching either because you got to the end of the list or because another thread set the done flag, you return `last` to indicate that no match was found here #6.

If the range can be divided, you first find the midpoint #7 before using `std::async` to run the search in the second half of the range #8, being careful to use `std::ref` to pass a reference to the done flag. In the meantime, you can search in the first half of the range by doing a direct recursive call #9. Both the asynchronous call and the direct recursion may result in further subdivisions if the original range is big enough.

If the direct search returned `mid_point`, then it failed to find a match, so you need to get the result of the asynchronous search. If no result was found in that half, the result will be `last`, which is the correct return value to indicate that the value was not found #10. If the “asynchronous” call was deferred rather than truly asynchronous, it will actually run here in the call to `get()`; in such circumstances the search of the top half of the range is skipped if

the search in the bottom half was successful. If the asynchronous search is really running on another thread, the destructor of the `async_result` variable will wait for the thread to complete, so you don't have any leaking threads.

As before, the use of `std::async` provides you with exception-safety and exception-propagation features. If the direct recursion throws an exception, the future's destructor will ensure that the thread running the asynchronous call has terminated before the function returns, and if the asynchronous call throws, the exception is propagated through the `get()` call #10. The use of a `try/catch` block around the whole thing is only there to set the `done` flag on an exception and ensure that all threads terminate quickly if an exception is thrown #11. The implementation would still be correct without it but would keep checking elements until every thread was finished.

A key feature that both implementations of this algorithm share with the other parallel algorithms you've seen is that there's no longer the guarantee that items are processed in the sequence that you get from `std::find`. This is essential if you're going to parallelize the algorithm. You can't process elements concurrently if the order matters. If the elements are independent, it doesn't matter for things like `parallel_for_each`, but it means that your `parallel_find` might return an element toward the end of the range even when there's a match toward the beginning, which might be surprising if you're not expecting it.

OK, so you've managed to parallelize `std::find`. As I stated at the beginning of this section, there are other similar algorithms that can complete without processing every data element, and the same techniques can be used for those. We'll also look further at the issue of interrupting threads in chapter 9.

To complete our trio of examples, we'll go in a different direction and look at `std::partial_sum`. This algorithm doesn't get a lot of press, but it's an interesting algorithm to parallelize and highlights some additional design choices.

8.5.3 A parallel implementation of `std::partial_sum`

`std::partial_sum` calculates the running totals in a range, so each element is replaced by the sum of that element and all the elements prior to it in the original sequence. Thus the sequence 1, 2, 3, 4, 5 becomes 1, $(1+2)=3$, $(1+2+3)=6$, $(1+2+3+4)=10$, $(1+2+3+4+5)=15$. This is interesting to parallelize because you can't just divide the range into chunks and calculate each chunk independently. For example, the initial value of the first element needs to be added to every other element.

One approach to determining the partial sum of a range is to calculate the partial sum of individual chunks and then add the resulting value of the last element in the first chunk onto the elements in the next chunk, and so forth. If you have the elements 1, 2, 3, 4, 5, 6, 7, 8, 9 and you're splitting into three chunks, you get {1, 3, 6}, {4, 9, 15}, {7, 15, 24} in the first instance. If you then add 6 (the sum for the last element in the first chunk) onto the elements in the second chunk, you get {1, 3, 6}, {10, 15, 21}, {7, 15, 24}. Then you add the last element of the second chunk (21) onto the elements in the third and final chunk to get the final result: {1, 3, 6}, {10, 15, 21}, {28, 36, 55}.

As well as the original division into chunks, the addition of the partial sum from the previous block can also be parallelized. If the last element of each block is updated first, the remaining elements in a block can be updated by one thread while a second thread updates the next block, and so forth. This works well when there are many more elements in the list than processing cores, because each core has a reasonable number of elements to process at each stage.

If you have a lot of processing cores (as many or more than the number of elements), this doesn't work so well. If you divide the work among the processors, you end up working in pairs of elements at the first step. Under these conditions, this forward propagation of results means that many processors are left waiting, so you need to find some work for them to do. You can then take a different approach to the problem. Rather than doing the full forward propagation of the sums from one chunk to the next, you do a partial propagation: first sum adjacent elements as before, but then add those sums to those two elements away, then add the next set of results to the results from four elements away, and so forth. If you start with the same initial nine elements, you get 1, 3, 5, 7, 9, 11, 13, 15, 17 after the first round, which gives you the final results for the first two elements. After the second you then have 1, 3, 6, 10, 14, 18, 22, 26, 30, which is correct for the first four elements. After round three you have 1, 3, 6, 10, 15, 21, 28, 36, 44, which is correct for the first eight elements, and finally after round four you have 1, 3, 6, 10, 15, 21, 28, 36, 45, which is the final answer. Although there are more total steps than in the first approach, there's greater scope for parallelism if you have many processors; each processor can update one entry with each step.

Overall, the second approach takes $\log_2(N)$ steps of around N operations (one per processor), where N is the number of elements in the list. This compares to the first algorithm where each thread has to perform N/k operations for the initial partial sum of the chunk allocated to it and then further N/k operations to do the forward propagation, where k is the number of threads. Thus the first approach is $O(N)$, whereas the second is $O(N \log(N))$ in terms of total number of operations. However, if you have as many processors as list elements, the second approach requires only $\log(N)$ operations *per processor*, whereas the first essentially serializes the operations when k gets large, because of the forward propagation. For small numbers of processing units, the first approach will therefore finish faster, whereas for massively parallel systems, the second will finish faster. This is an extreme example of the issues discussed in section 8.2.1.

Anyway, efficiency issues aside, let's look at some code. The following listing shows the first approach.

Listing 8.11 Calculating partial sums in parallel by dividing the problem

```
template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
    typedef typename Iterator::value_type value_type;

    struct process_chunk      #1
    {
```

```

void operator()(Iterator begin,Iterator last,
                 std::future<value_type>* previous_end_value,
                 std::promise<value_type>* end_value)
{
    try
    {
        Iterator end=last;
        ++end;
        std::partial_sum(begin,end,begin);      #2
        if(previous_end_value)                  #3
        {
            value_type& addend=previous_end_value->get();    #4
            *last+=addend;                                #5
            if(end_value)
            {
                end_value->set_value(*last);             #6
            }
            std::for_each(begin,last,[addend](value_type& item) #7
                          {
                              item+=addend;
                          });
        }
        else if(end_value)
        {
            end_value->set_value(*last);             #8
        }
    }
    catch(...)
    {
        if(end_value)
        {
            end_value->set_exception(std::current_exception()); #10
        }
        else
        {
            throw;                                #11
        }
    }
};

unsigned long const length=std::distance(first,last);
if(!length)
    return;
unsigned long const min_per_thread=25;                      #12
unsigned long const max_threads=
    (length+min_per_thread-1)/min_per_thread;
unsigned long const hardware_threads=
    std::thread::hardware_concurrency();
unsigned long const num_threads=
    std::min(hardware_threads!=0?hardware_threads:2,max_threads);
unsigned long const block_size=length/num_threads;
typedef typename Iterator::value_type value_type;
std::vector<std::thread> threads(num_threads-1);          #13
std::vector<std::promise<value_type>>
    end_values(num_threads-1);                                #14
std::vector<std::future<value_type>>
    previous_end_values;                                     #15
previous_end_values.reserve(num_threads-1);                 #16
join_threads joiner(threads);

```

```

Iterator block_start=first;
for(unsigned long i=0;i<(num_threads-1);++i)
{
    Iterator block_last=block_start;
    std::advance(block_last,block_size-1); #17
    threads[i]=std::thread(process_chunk(), #18
                           block_start,block_last,
                           (i!=0)?&previous_end_values[i-1]:0,
                           &end_values[i]);
    block_start=block_last;
    ++block_start; #19
    previous_end_values.push_back(end_values[i].get_future()); #20
}
Iterator final_element=block_start;
std::advance(final_element,std::distance(block_start,last)-1); #21
process_chunk()(block_start,final_element, #22
                (num_threads>1)?&previous_end_values.back():0,
                0);
}

```

In this instance, the general structure is the same as with the previous algorithms, dividing the problem into chunks, with a minimum chunk size per thread #12. In this case, as well as the vector of threads #13, you have a vector of promises #14, which is used to store the value of the last element in the chunk, and a vector of futures #15, which is used to retrieve the last value from the previous chunk. You can reserve the space for the futures #16 to avoid a reallocation while spawning threads, because you know how many you're going to have.

The main loop is the same as before, except this time you actually want the iterator that *points to* the last element in each block, rather than being the usual one past the end #17, so that you can do the forward propagation of the last element in each range. The actual processing is done in the `process_chunk` function object, which we'll look at shortly; the start and end iterators for this chunk are passed in as arguments alongside the future for the end value of the previous range (if any) and the promise to hold the end value of this range #18.

After you've spawned the thread, you can update the block start, remembering to advance it past that last element #19, and store the future for the last value in the current chunk into the vector of futures so it will be picked up next time around the loop #20.

Before you process the final chunk, you need to get an iterator for the last element #21, which you can pass in to `process_chunk` #22. `std::partial_sum` doesn't return a value, so you don't need to do anything once the final chunk has been processed. The operation is complete once all the threads have finished.

OK, now it's time to look at the `process_chunk` function object that actually does all the work #1. You start by calling `std::partial_sum` for the entire chunk, including the final element #2, but then you need to know if you're the first chunk or not #3. If you are *not* the first chunk, then there was a `previous_end_value` from the previous chunk, so you need to wait for that #4. In order to maximize the parallelism of the algorithm, you then update the last element first #5, so you can pass the value on to the next chunk (if there is one) #6. Once you've done that, you can just use `std::for_each` and a simple lambda function #7 to update all the remaining elements in the range.

If there was *not* a `previous_end_value`, you're the first chunk, so you can just update the `end_value` for the next chunk (again, if there is one—you might be the only chunk) #8.

Finally, if any of the operations threw an exception, you catch it #9 and store it in the promise #10 so it will propagate to the next chunk when it tries to get the previous end value #4. This will propagate all exceptions into the final chunk, which then just rethrows #11, because you know you're running on the main thread.

Because of the synchronization between the threads, this code isn't readily amenable to rewriting with `std::async`. The tasks wait on results made available partway through the execution of other tasks, so all tasks must be running concurrently.

With the block-based, forward-propagation approach out of the way, let's look at the second approach to computing the partial sums of a range.

IMPLEMENTING THE INCREMENTAL PAIRWISE ALGORITHM FOR PARTIAL SUMS

This second approach to calculating the partial sums by adding elements increasingly further away works best where your processors can execute the additions in lockstep. In this case, no further synchronization is necessary because all the intermediate results can be propagated directly to the next processor that needs them. But in practice you rarely have such systems to work with except for those cases where a single processor can execute the same instruction across a small number of data elements simultaneously with so-called Single-Instruction/Multiple-Data (SIMD) instructions. Therefore, you must design your code for the general case and explicitly synchronize the threads at each step.

One way to do this is to use a *barrier*—a synchronization mechanism that causes threads to wait until the required number of threads has reached the barrier. Once all the threads have reached the barrier, they're all unblocked and may proceed. The C++11 Thread Library doesn't offer such a facility directly, so you have to design one yourself.

Imagine a roller coaster at the fairground. If there's a reasonable number of people waiting, the fairground staff will ensure that every seat is filled before the roller coaster leaves the platform. A barrier works the same way: you specify up front the number of "seats," and threads have to wait until all the "seats" are filled. Once there are enough waiting threads, they can all proceed; the barrier is reset and starts waiting for the next batch of threads. Often, such a construct is used in a loop, where the same threads come around and wait next time. The idea is to keep the threads in lockstep, so one thread doesn't run away in front of the others and get out of step. For an algorithm such as this one, that would be disastrous, because the runaway thread would potentially modify data that was still being used by other threads or use data that hadn't been correctly updated yet.

Anyway, the following listing shows a simple implementation of a barrier.

Listing 8.12 A simple barrier class

```
class barrier
{
    unsigned const count;
    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;
```

```

public:
    explicit barrier(unsigned count_):           #1
        count(count_),spaces(count_),generation(0)
    {}
    void wait()
    {
        unsigned const my_generation=generation;   #2
        if(!--spaces)                            #3
        {
            spaces=count;                      #4
            ++generation;                     #5
        }
        else
        {
            while(generation==my_generation)    #6
                std::this_thread::yield();       #7
        }
    }
};

```

With this implementation, you construct a barrier with the number of “seats” #1, which is stored in the `count` variable. Initially, the number of `spaces` at the barrier is equal to this count. As each thread waits, the number of `spaces` is decremented #3. When it reaches zero, the number of `spaces` is reset back to `count` #4, and the `generation` is increased to signal to the other threads that they can continue #5. If the number of free `spaces` does not reach zero, you have to wait. This implementation uses a simple spin lock #6, checking the `generation` against the value you retrieved at the beginning of `wait()` #2. Because the `generation` is only updated when all the threads have reached the barrier #5, you `yield()` while waiting #7 so the waiting thread doesn’t hog the CPU in a busy wait.

When I said this implementation was simple, I meant it: it uses a spin wait, so it’s not ideal for cases where threads are likely to be waiting a long time, and it doesn’t work if there’s more than `count` threads that can potentially call `wait()` at any one time. If you need to handle either of those scenarios, you must use a more robust (but more complex) implementation instead. I’ve also stuck to sequentially consistent operations on the atomic variables, because that makes everything easier to reason about, but you could potentially relax some of the ordering constraints. Such global synchronization is expensive on massively parallel architectures, because the cache line holding the barrier state must be shuttled between all the processors involved (see the discussion of cache ping-pong in section 8.2.2), so you must take great care to ensure that this really is the best choice here. If your C++ Standard Library provides the facilities from the Concurrency TS, you could use `std::experimental::barrier` here. See chapter 4 for details.

Anyway, this is just what you need here; you have a fixed number of threads that need to run in a lockstep loop. Well, it’s *almost* a fixed number of threads. As you may remember, the items at the beginning of the list acquire their final values after a couple of steps. This means that either you have to keep those threads looping until the entire range has been processed, or you need to allow your barrier to handle threads dropping out, and thus decreasing `count`. I

opted for the latter option, because it avoids having threads doing unnecessary work just looping until the final step is done.

This means you have to change `count` to be an atomic variable, so you can update it from multiple threads without external synchronization:

```
std::atomic<unsigned> count;
```

The initialization remains the same, but now you have to explicitly `load()` from `count` when you reset the number of `spaces`:

```
spaces=count.load();
```

These are all the changes that you need on the `wait()` front; now you need a new member function to decrement `count`. Let's call it `done_waiting()`, because a thread is declaring that it is done with waiting:

```
void done_waiting()
{
    --count;                      #1
    if(!--spaces)                  #2
    {
        spaces=count.load();      #3
        ++generation;
    }
}
```

The first thing you do is decrement the `count` #1 so that the next time `spaces` is reset it reflects the new lower number of waiting threads. Then you need to decrease the number of free `spaces` #2. If you don't do this, the other threads will be waiting forever, because `spaces` was initialized to the old, larger value. If you're the last thread through on this batch, you need to reset the counter and increase the generation #3, just as you do in `wait()`. The key difference here is that if you're the last thread in the batch, you don't have to wait. You're finished with waiting after all!

You're now ready to write your second implementation of partial sum. At each step, every thread calls `wait()` on the barrier to ensure the threads step through together, and once each thread is done, it calls `done_waiting()` on the barrier to decrement the count. If you use a second buffer alongside the original range, the barrier provides all the synchronization you need. At each step the threads read from either the original range or the buffer and write the new value to the corresponding element of the other. If the threads read from the original range on one step, they read from the buffer on the next, and vice versa. This ensures there are no race conditions between the reads and writes by separate threads. Once a thread has finished looping, it must ensure that the correct final value has been written to the original range. The following listing pulls this all together.

Listing 8.13 A parallel implementation of `partial_sum` by pairwise updates

```
struct barrier
{
    std::atomic<unsigned> count;
```

```

    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;
    barrier(unsigned count_):
        count(count_),spaces(count_),generation(0)
    {}
    void wait()
    {
        unsigned const gen=generation.load();
        if(!--spaces)
        {
            spaces=count.load();
            ++generation;
        }
        else
        {
            while(generation.load()==gen)
            {
                std::this_thread::yield();
            }
        }
    }
    void done_waiting()
    {
        --count;
        if(!--spaces)
        {
            spaces=count.load();
            ++generation;
        }
    }
};

template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
    typedef typename Iterator::value_type value_type;
    struct process_element #1
    {
        void operator()(Iterator first,Iterator last,
                         std::vector<value_type>& buffer,
                         unsigned i,barrier& b)
        {
            value_type& ith_element=*(first+i);
            bool update_source=false;

            for(unsigned step=0,stride=1;stride<=i;++step,stride*=2)
            {
                value_type const& source=(step%2)? #2
                    buffer[i]:ith_element;
                value_type& dest=(step%2)?
                    ith_element:buffer[i];
                value_type const& addend=(step%2)? #3
                    buffer[i-stride]:*(first+i-stride);
                dest=source+addend; #4
                update_source!=!(step%2);
                b.wait(); #5
            }
            if(update_source) #6
            {
                ith_element=buffer[i];
            }
        }
    };
}

```

```

        }
        b.done_waiting();                                #7
    }
};

unsigned long const length=std::distance(first,last);
if(length<=1)
    return;
std::vector<value_type> buffer(length);
barrier b(length);
std::vector<std::thread> threads(length-1);          #8
join_threads joiner(threads);
Iterator block_start=first;
for(unsigned long i=0;i<(length-1);++i)
{
    threads[i]=std::thread(process_element(),first,last,
                           std::ref(buffer),i,std::ref(b));      #9
}
process_element()(first,last,buffer,length-1,b);        #10
}

```

The overall structure of this code is probably becoming familiar by now. You have a class with a function call operator (`process_element`) for doing the work #1, which you run on a bunch of threads #9 stored in a vector #8 and which you also call from the main thread #10. The key difference this time is that the number of threads is dependent on the number of items in the list rather than on `std::thread::hardware_concurrency`. As I said already, unless you're on a massively parallel machine where threads are cheap, this is probably a bad idea, but it shows the overall structure. It would be possible to have fewer threads, with each thread handling several values from the source range, but there will come a point where there are sufficiently few threads that this is less efficient than the forward-propagation algorithm.

Anyway, the key work is done in the function call operator of `process_element`. At each step you either take the *i*th element from the original range or the *i*th element from the buffer #2 and add it to the value `stride` elements prior #3, storing it in the buffer if you started in the original range or back in the original range if you started in the buffer #4. You then wait on the barrier #5 before starting the next step. You've finished when the `stride` takes you off the start of the range, in which case you need to update the element in the original range if your final result was stored in the buffer #6. Finally, you tell the barrier that you're `done_waiting()` #7.

Note that this solution isn't exception safe. If an exception is thrown in `process_element` on one of the worker threads, it will terminate the application. You could deal with this by using a `std::promise` to store the exception, as you did for the `parallel_find` implementation from listing 8.9, or even just using a `std::exception_ptr` protected by a mutex.

That concludes our three examples. Hopefully, they've helped to crystallize some of the design considerations highlighted in sections 8.1, 8.2, 8.3, and 8.4 and have demonstrated how these techniques can be brought to bear in real code.

8.6 Summary

We've covered quite a lot of ground in this chapter. We started with various techniques for dividing work between threads, such as dividing the data beforehand or using a number of threads to form a pipeline. We then looked at the issues surrounding the performance of multithreaded code from a low-level perspective, with a look at false sharing and data contention before moving on to how the patterns of data access can affect the performance of a bit of code. We then looked at additional considerations in the design of concurrent code, such as exception safety and scalability. Finally, we ended with a number of examples of parallel algorithm implementations, each of which highlighted particular issues that can occur when designing multithreaded code.

One item that has cropped up a couple of times in this chapter is the idea of a thread pool—a preconfigured group of threads that run tasks assigned to the pool. Quite a lot of thought goes into the design of a good thread pool, so we'll look at some of the issues in the next chapter, along with other aspects of advanced thread management.

9

Advanced thread management

This chapter covers

- Thread pools
- Handling dependencies between pool tasks
- Work stealing for pool threads
- Interrupting threads

In earlier chapters, we've been explicitly managing threads by creating `std::thread` objects for every thread. In a couple of places you've seen how this can be undesirable, because you then have to manage the lifetime of the thread objects, determine the number of threads appropriate to the problem and to the current hardware, and so forth. The ideal scenario would be that you could just divide the code into the smallest pieces that can be executed concurrently, pass them over to the compiler and library, and say, "Parallelize this for optimal performance." As we'll see in chapter 10, there are cases where you can do this: if your code that requires parallelization can be expressed as a call to a standard library algorithm, then you can ask the library to do the parallelization for you in a large number of cases.

Another recurring theme in several of the examples is that you might use several threads to solve a problem but require that they finish early if some condition is met. This might be because the result has already been determined, or because an error has occurred, or even because the user has explicitly requested that the operation be aborted. Whatever the reason, the threads need to be sent a "Please stop" request so that they can give up on the task they were given, tidy up, and finish as soon as possible.

In this chapter, we'll look at mechanisms for managing threads and tasks, starting with the automatic management of the number of threads and the division of tasks between them.

9.1 Thread pools

In many companies, employees who would normally spend their time in the office are occasionally required to visit clients or suppliers or attend a trade show or conference. Although these trips might be necessary, and on any given day there might be several people making such a trip, it may well be months or even years between such trips for any particular employee. Since it would therefore be rather expensive and impractical for each employee to have a company car, companies often offer a *car pool* instead; they have a limited number of cars that are available to all employees. When an employee needs to make an off-site trip, they book one of the pool cars for the appropriate time and return it for others to use when they return to the office. If there are no pool cars free on a given day, the employee will have to reschedule their trip for a subsequent date.

A *thread pool* is a similar idea, except that *threads* are being shared rather than cars. On most systems, it's impractical to have a separate thread for every task that can potentially be done in parallel with other tasks, but you'd still like to take advantage of the available concurrency where possible. A thread pool allows you to accomplish this; tasks that can be executed concurrently are submitted to the pool, which puts them on a queue of pending work. Each task is then taken from the queue by one of the *worker threads*, which executes the task before looping back to take another from the queue.

There are several key design issues when building a thread pool, such as how many threads to use, the most efficient way to allocate tasks to threads, and whether or not you can wait for a task to complete. In this section we'll look at some thread pool implementations that address these design issues, starting with the simplest possible thread pool.

9.1.1 The simplest possible thread pool

At its simplest, a thread pool is a fixed number of *worker threads* (typically the same number as the value returned by `std::thread::hardware_concurrency()`) that process work. When you have work to do, you call a function to put it on the queue of pending work. Each worker thread takes work off the queue, runs the specified task, and then goes back to the queue for more work. In the simplest case there's no way to wait for the task to complete. If you need to do this, you have to manage the synchronization yourself.

The following listing shows a sample implementation of such a thread pool.

Listing 9.1 Simple thread pool

```
class thread_pool
{
    std::atomic_bool done;
    thread_safe_queue<std::function<void()> > work_queue;      #1
    std::vector<std::thread> threads;                            #2
    join_threads joiner;                                         #3
    void worker_thread()
    {
        while(!done)                                              #4
        {
            std::function<void()> task;
```

```

        if(work_queue.try_pop(task))                      #5
        {
            task();                                     #6
        }
        else
        {
            std::this_thread::yield();                  #7
        }
    }
public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency(); #8
        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this)); #9
            }
        }
        catch(...)
        {
            done=true;                                #10
            throw;
        }
    }
    ~thread_pool()
    {
        done=true;                                #11
    }
    template<typename FunctionType>
    void submit(FunctionType f)
    {
        work_queue.push(std::function<void()>(f));   #12
    }
};

```

This implementation has a vector of worker threads #2 and uses one of the thread-safe queues from chapter 6 #1 to manage the queue of work. In this case, users can't wait for the tasks, and they can't return any values, so you can use `std::function<void()>` to encapsulate your tasks. The `submit()` function then wraps whatever function or callable object is supplied inside a `std::function<void()>` instance and pushes it on the queue #12.

The threads are started in the constructor: you use `std::thread::hardwareConcurrency()` to tell you how many concurrent threads the hardware can support #8, and you create that many threads running your `worker_thread()` member function #9.

Starting a thread can fail by throwing an exception, so you need to ensure that any threads you've already started are stopped and cleaned up nicely in this case. This is achieved with a try-catch block that sets the `done` flag when an exception is thrown #10, alongside an instance of the `join_threads` class from chapter 8 #3 to join all the threads. This also works with the destructor: you can just set the `done` flag #11, and the `join_threads` instance will

ensure that all the threads have completed before the pool is destroyed. Note that the order of declaration of the members is important: both the `done` flag and the `worker_queue` must be declared before the `threads` vector, which must in turn be declared before the `joiner`. This ensures that the members are destroyed in the right order; you can't destroy the queue safely until all the threads have stopped, for example.

The `worker_thread` function itself is quite simple: it sits in a loop waiting until the `done` flag is set #4, pulling tasks off the queue #5 and executing them #6 in the meantime. If there are no tasks on the queue, the function calls `std::this_thread::yield()` to take a small break #7 and give another thread a chance to put some work on the queue before it tries to take some off again the next time around.

For many purposes such a simple thread pool will suffice, especially if the tasks are entirely independent and don't return any values or perform any blocking operations. But there are also many circumstances where such a simple thread pool may not adequately address your needs and yet others where it can cause problems such as deadlock. Also, in the simple cases you may well be better served using `std::async` as in many of the examples in chapter 8. Throughout this chapter, we'll look at more complex thread pool implementations that have additional features either to address user needs or reduce the potential for problems. First up: waiting for the tasks we've submitted.

9.1.2 Waiting for tasks submitted to a thread pool

In the examples in chapter 8 that explicitly spawned threads, after dividing the work between threads, the master thread always waited for the newly spawned threads to finish, to ensure that the overall task was complete before returning to the caller. With thread pools, you'd need to wait for the tasks submitted to the thread pool to complete, rather than the worker threads themselves. This is similar to the way that the `std::async`-based examples in chapter 8 waited for the futures. With the simple thread pool from listing 9.1, you'd have to do this manually using the techniques from chapter 4: condition variables and futures. This adds complexity to the code; it would be better if you could wait for the tasks directly.

By moving that complexity into the thread pool itself, you *can* wait for the tasks directly. You can have the `submit()` function return a task handle of some description that you can then use to wait for the task to complete. This task handle would wrap the use of condition variables or futures, thus simplifying the code that uses the thread pool.

A special case of having to wait for the spawned task to finish occurs when the main thread needs a result computed by the task. You've seen this in examples throughout the book, such as the `parallel_accumulate()` function from chapter 2. In this case, you can combine the waiting with the result transfer through the use of futures. Listing 9.2 shows the changes required to the simple thread pool that allows you to wait for tasks to complete and then pass return values from the task to the waiting thread. Since `std::packaged_task<>` instances are not *copyable*, just *movable*, you can no longer use `std::function<>` for the queue entries, because `std::function<>` requires that the stored function objects are copy-constructible. Instead, you must use a custom function wrapper that can handle move-only types. This is a

simple type-erasure class with a function call operator. You only need to handle functions that take no parameters and return `void`, so this is a straightforward virtual call in the implementation.

Listing 9.2 A thread pool with waitable tasks

```
class function_wrapper
{
    struct impl_base {
        virtual void call()=0;
        virtual ~impl_base() {}
    };
    std::unique_ptr<impl_base> impl;
    template<typename F>
    struct impl_type: impl_base
    {
        F f;
        impl_type(F&& f_): f(std::move(f_)) {}
        void call() { f(); }
    };
public:
    template<typename F>
    function_wrapper(F&& f):
        impl(new impl_type<F>(std::move(f)))
    {}
    void operator()() { impl->call(); }
    function_wrapper() = default;
    function_wrapper(function_wrapper&& other):
        impl(std::move(other.impl))
    {}
    function_wrapper& operator=(function_wrapper&& other)
    {
        impl=std::move(other.impl);
        return *this;
    }
    function_wrapper(const function_wrapper&)=delete;
    function_wrapper(function_wrapper&)=delete;
    function_wrapper& operator=(const function_wrapper&)=delete;
};
class thread_pool
{
    thread_safe_queue<function_wrapper> work_queue; #A
    void worker_thread()
    {
        while(!done)
        {
            function_wrapper task; #A
            if(work_queue.try_pop(task))
            {
                task();
            }
            else
            {
                std::this_thread::yield();
            }
        }
    }
public:
```

```

template<typename FunctionType>
std::future<typename std::result_of<FunctionType()>::type>      #1
    submit(FunctionType f)
{
    typedef typename std::result_of<FunctionType()>::type
        result_type;                                              #2
    std::packaged_task<result_type()> task(std::move(f));       #3
    std::future<result_type> res(task.get_future());             #4
    work_queue.push(std::move(task));                            #5
    return res;                                                 #6
}
// rest as before
};

```

#A Use function_wrapper rather than std::function

First, the modified `submit()` function #1 returns a `std::future<>` to hold the return value of the task and allow the caller to wait for the task to complete. This requires that you know the return type of the supplied function `f`, which is where `std::result_of<>` comes in: `std::result_of<FunctionType()>::type` is the type of the result of invoking an instance of type `FunctionType` (such as `f`) with no arguments. You use the same `std::result_of<>` expression for the `result_type` typedef #2 inside the function.

You then wrap the function `f` in a `std::packaged_task<result_type()>` #3, because `f` is a function or callable object that takes no parameters and returns an instance of type `result_type`, as we just deduced. You can now get your future from the `std::packaged_task<>` #4, before pushing the task onto the queue #5 and returning the future #6. Note that you have to use `std::move()` when pushing the task onto the queue, because `std::packaged_task<>` isn't copyable. The queue now stores `function_wrapper` objects rather than `std::function<void()>` objects in order to handle this.

This pool thus allows you to wait for your tasks and have them return results. The next listing shows what the `parallel_accumulate` function looks like with such a thread pool.

Listing 9.3 parallel_accumulate using a thread pool with waitable tasks

```

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return init;
    unsigned long const block_size=25;
    unsigned long const num_blocks=(length+block_size-1)/block_size;  #1
    std::vector<std::future<T>> futures(num_blocks-1);
    thread_pool pool;
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        futures[i]=pool.submit([=]{
            accumulate_block<Iterator,T>()(block_start,block_end);
        });      #2
    }
}

```

```

        block_start=block_end;
    }
    T last_result=accumulate_block<Iterator,T>()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        result+=futures[i].get();
    }
    result += last_result;
    return result;
}

```

When you compare this against listing 8.4, there are a couple of things to notice. First, you're working in terms of the number of blocks to use (`num_blocks #1`) rather than the number of threads. In order to make the most use of the scalability of your thread pool, you need to divide the work into the smallest blocks that it's worth working with concurrently. When there are only a few threads in the pool, each thread will process many blocks, but as the number of threads grows with the hardware, the number of blocks processed in parallel will also grow.

You need to be careful when choosing the “smallest blocks that it's worth working with concurrently.” There's an inherent overhead to submitting a task to a thread pool, having the worker thread run it, and passing the return value through a `std::future<>`, and for small tasks it's not worth the payoff. *If you choose too small a task size, the code may run more slowly with a thread pool than with one thread.*

Assuming the block size is sensible, you don't have to worry about packaging the tasks, obtaining the futures, or storing the `std::thread` objects so you can join with the threads later; the thread pool takes care of that. All you need to do is call `submit()` with your task #2.

The thread pool takes care of the exception safety too. Any exception thrown by the task gets propagated through the future returned from `submit()`, and if the function exits with an exception, the thread pool destructor abandons any not-yet-completed tasks and waits for the pool threads to finish.

This works well for simple cases like this, where the tasks are independent. But it's not so good for situations where the tasks depend on other tasks also submitted to the thread pool.

9.1.3 Tasks that wait for other tasks

The Quicksort algorithm is an example that I've used throughout this book. It's simple in concept: the data to be sorted is partitioned into those items that go before a pivot item and those that go after it in the sorted sequence. These two sets of items are recursively sorted and then stitched back together to form a fully sorted set. When parallelizing this algorithm, you need to ensure that these recursive calls make use of the available concurrency.

Back in chapter 4, when I first introduced this example, we used `std::async` to run one of the recursive calls at each stage, letting the library choose between running it on a new thread and running it synchronously when the relevant `get()` was called. This works well, because each task is either running on its own thread or will be invoked when required.

When we revisited the implementation in chapter 8, you saw an alternative structure that used a fixed number of threads related to the available hardware concurrency. In this case, you used a stack of pending chunks that needed sorting. As each thread partitioned the data it was sorting, it added a new chunk to the stack for one of the sets of data and then sorted the other one directly. At this point, a straightforward wait for the sorting of the other chunk to complete would potentially deadlock, because you'd be consuming one of your limited number of threads waiting. It would be very easy to end up in a situation where all of the threads were waiting for chunks to be sorted and no threads were actually doing any sorting. We addressed this issue by having the threads pull chunks off the stack and sort them while the particular chunk they were waiting for was unsorted.

You'd get the same problem if you substituted a simple thread pool like the ones you've seen so far in this chapter instead of `std::async` in the example from chapter 4. There are now only a limited number of threads, and they might end up all waiting for tasks that haven't been scheduled because there are no free threads. You therefore need to use a solution similar to the one you used in chapter 8: process outstanding chunks while you're waiting for your chunk to complete. If you're using the thread pool to manage the list of tasks and their association with threads—which is, after all, the whole point of using a thread pool—you don't have access to the task list to do this. What you need to do is modify the thread pool to do this automatically.

The simplest way to do this is to add a new function on `thread_pool` to run a task from the queue and manage the loop yourself, so we'll go with that. Advanced thread pool implementations might add logic into the `wait` function or additional `wait` functions to handle this case, possibly prioritizing the task being waited for. The following listing shows the new `run_pending_task()` function, and a modified Quicksort to make use of it is shown in listing 9.5.

Listing 9.4 An implementation of `run_pending_task()`

```
void thread_pool::run_pending_task()
{
    function_wrapper task;
    if(work_queue.try_pop(task))
    {
        task();
    }
    else
    {
        std::this_thread::yield();
    }
}
```

This implementation of `run_pending_task()` is lifted straight out of the main loop of the `worker_thread()` function, which can now be modified to call the extracted `run_pending_task()`. This tries to take a task of the queue and run it if there is one; otherwise, it yields to allow the OS to reschedule the thread. The Quicksort implementation

next is a lot simpler than the corresponding version from listing 8.1, because all the thread-management logic has been moved to the thread pool.

Listing 9.5 A thread pool-based implementation of Quicksort

```
template<typename T>
struct sorter          #1
{
    thread_pool pool;      #2

    std::list<T> do_sort(std::list<T>& chunk_data)
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }
        std::list<T> result;
        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();
        typename std::list<T>::iterator divide_point=
            std::partition(chunk_data.begin(),chunk_data.end(),
                           [&](T const& val){return val<partition_val;});
        std::list<T> new_lower_chunk;
        new_lower_chunk.splice(new_lower_chunk.end(),
                               chunk_data,chunk_data.begin(),
                               divide_point);
        std::future<std::list<T> > new_lower=           #3
            pool.submit(std::bind(&sorter::do_sort,this,
                                  std::move(new_lower_chunk)));
        std::list<T> new_higher(do_sort(chunk_data));
        result.splice(result.end(),new_higher);
        while(new_lower.wait_for(std::chrono::seconds(0)) ==
              std::future_status::timeout)
        {
            pool.run_pending_task();                      #4
        }
        result.splice(result.begin(),new_lower.get());
        return result;
    }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);
}
```

Just as in listing 8.1, you've delegated the real work to the `do_sort()` member function of the `sorter` class template #1, although in this case the class is only there to wrap the `thread_pool` instance #2.

Your thread and task management is now reduced to submitting a task to the pool #3 and running pending tasks while waiting #4. This is much simpler than in listing 8.1, where you

had to explicitly manage the threads and the stack of chunks to sort. When submitting the task to the pool, you use `std::bind()` to bind the `this` pointer to `do_sort()` and to supply the chunk to sort. In this case, you call `std::move()` on the `new_lower_chunk` as you pass it in, to ensure that the data is moved rather than copied.

Although this has now addressed the crucial deadlock-causing problem with tasks that wait for other tasks, this thread pool is still far from ideal. For starters, every call to `submit()` and every call to `run_pending_task()` accesses the same queue. You saw in chapter 8 how having a single set of data modified by multiple threads can have a detrimental effect on performance, so you need to somehow address this problem.

9.1.4 Avoiding contention on the work queue

Every time a thread calls `submit()` on a particular instance of the thread pool, it has to push a new item onto the single shared work queue. Likewise, the worker threads are continually popping items off the queue in order to run the tasks. This means that as the number of processors increases, there's increasing contention on the queue. This can be a real performance drain; even if you use a lock-free queue so there's no explicit waiting, cache ping-pong can be a substantial time sink.

One way to avoid cache ping-pong is to use a separate work queue per thread. Each thread then posts new items to its own queue and takes work from the global work queue only if there's no work on its own individual queue. The following listing shows an implementation that makes use of a `thread_local` variable to ensure that each thread has its own work queue, as well as the global one.

Listing 9.6 A thread pool with thread-local work queues

```
class thread_pool
{
    thread_safe_queue<function_wrapper> pool_work_queue;
    typedef std::queue<function_wrapper> local_queue_type;      #1
    static thread_local std::unique_ptr<local_queue_type>
        local_work_queue;                                         #2
    void worker_thread()
    {
        local_work_queue.reset(new local_queue_type);           #3
        while(!done)
        {
            run_pending_task();
        }
    }
public:
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type>
    submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;
        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)                                       #4
        {
```

```

        local_work_queue->push(std::move(task));
    }
    else
    {
        pool_work_queue.push(std::move(task));           #5
    }
    return res;
}
void run_pending_task()
{
    function_wrapper task;
    if(local_work_queue && !local_work_queue->empty())      #6
    {
        task=std::move(local_work_queue->front());
        local_work_queue->pop();
        task();
    }
    else if(pool_work_queue.try_pop(task))                 #7
    {
        task();
    }
    else
    {
        std::this_thread::yield();
    }
}
// rest as before
};

```

We've used a `std::unique_ptr<>` to hold the thread-local work queue #2 because we don't want other threads that aren't part of our thread pool to have one; this is initialized in the `worker_thread()` function before the processing loop #3. The destructor of `std::unique_ptr<>` will ensure that the work queue is destroyed when the thread exits.

`submit()` then checks to see if the current thread has a work queue #4. If it does, it's a pool thread, and you can put the task on the local queue; otherwise, you need to put the task on the pool queue as before #5.

There's a similar check in `run_pending_task()` #6, except this time you also need to check to see if there are any items on the local queue. If there are, you can take the front one and process it; notice that the local queue can be a plain `std::queue< #1` because it's only ever accessed by the one thread. If there are no tasks on the local queue, you try the pool queue as before #7.

This works fine for reducing contention, but when the distribution of work is uneven, it can easily result in one thread having a lot of work on its queue while the others have no work do to. For example, with the Quicksort example, only the topmost chunk would make it to the pool queue, because the remaining chunks would end up on the local queue of the worker thread that processed that one. This defeats the purpose of using a thread pool.

Thankfully, there is a solution to this: allow the threads to *steal* work from each other's queues if there's no work in their queue and no work in the global queue.

9.1.5 Work stealing

In order to allow a thread with no work to do to take work from another thread with a full queue, the queue must be accessible to the thread doing the stealing from `run_pending_tasks()`. This requires that each thread register its queue with the thread pool or be given one by the thread pool. Also, you must ensure that the data in the work queue is suitably synchronized and protected, so that your invariants are protected.

It's possible to write a lock-free queue that allows the owner thread to push and pop at one end while other threads can steal entries from the other, but the implementation of such a queue is beyond the scope of this book. In order to demonstrate the idea, we'll stick to using a mutex to protect the queue's data. We hope work stealing is a rare event, so there should be little contention on the mutex, and such a simple queue should therefore have minimal overhead. A simple lock-based implementation is shown here.

Listing 9.7 Lock-based queue for work stealing

```
class work_stealing_queue
{
private:
    typedef function_wrapper data_type;
    std::deque<data_type> the_queue;      #1
    mutable std::mutex the_mutex;
public:
    work_stealing_queue()
    {}
    work_stealing_queue(const work_stealing_queue& other)=delete;
    work_stealing_queue& operator=(const work_stealing_queue& other)=delete;
    void push(data_type data)      #2
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        the_queue.push_front(std::move(data));
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        return the_queue.empty();
    }
    bool try_pop(data_type& res)      #3
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }
        res=std::move(the_queue.front());
        the_queue.pop_front();
        return true;
    }
    bool try_steal(data_type& res)     #4
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }
        res=std::move(the_queue.back());
        the_queue.pop_back();
        return true;
    }
};
```

```

        }
        res=std::move(the_queue.back());
        the_queue.pop_back();
        return true;
    }
};

```

This queue is a simple wrapper around a `std::deque<function_wrapper>` #1 that protects all accesses with a mutex lock. Both `push()` #2 and `try_pop()` #3 work on the front of the queue, while `try_steal()` #4 works on the back.

This actually means that this “queue” is a last-in-first-out stack for its own thread; the task most recently pushed on is the first one off again. This can help improve performance from a cache perspective, because the data related to that task is more likely to still be in the cache than the data related to a task pushed on the queue previously. Also, it maps nicely to algorithms such as Quicksort. In the previous implementation, each call to `do_sort()` pushes one item on the stack and then waits for it. By processing the most recent item first, you ensure that the chunk needed for the current call to complete is processed before the chunks needed for the other branches, thus reducing the number of active tasks and the total stack usage. `try_steal()` takes items from the opposite end of the queue to `try_pop()` in order to minimize contention; you could potentially use the techniques discussed in chapters 6 and 7 to enable concurrent calls to `try_pop()` and `try_steal()`.

OK, so you have your nice sparkly work queue that permits stealing; how do you use it in your thread pool? Here’s one potential implementation.

Listing 9.8 A thread pool that uses work stealing

```

class thread_pool
{
    typedef function_wrapper task_type;
    std::atomic_bool done;
    thread_safe_queue<task_type> pool_work_queue;
    std::vector<std::unique_ptr<work_stealing_queue>> queues;      #1
    std::vector<std::thread> threads;
    join_threads joiner;
    static thread_local work_stealing_queue* local_work_queue;      #2
    static thread_local unsigned my_index;
    void worker_thread(unsigned my_index_)
    {
        my_index=my_index_;
        local_work_queue=queues[my_index].get();                      #3
        while(!done)
        {
            run_pending_task();
        }
    }
    bool pop_task_from_local_queue(task_type& task)
    {
        return local_work_queue && local_work_queue->try_pop(task);
    }
    bool pop_task_from_pool_queue(task_type& task)
    {
        return pool_work_queue.try_pop(task);
    }
};

```

```

    }
    bool pop_task_from_other_thread_queue(task_type& task)           #4
    {
        for(unsigned i=0;i<queues.size();++i)
        {
            unsigned const index=(my_index+i+1)%queues.size();          #5
            if(queues[index]->try_steal(task))
            {
                return true;
            }
        }
        return false;
    }
public:
    thread_pool():
        done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency();
        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                queues.push_back(std::unique_ptr<work_stealing_queue>(  #6
                    new work_stealing_queue));
            }
            for(unsigned i=0;i<thread_count;++i)
            {
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this,i));
            }
        }
        catch(...)
        {
            done=true;
            throw;
        }
    }
    ~thread_pool()
    {
        done=true;
    }
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type> submit(
        FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;
        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)
        {
            local_work_queue->push(std::move(task));
        }
        else
        {
            pool_work_queue.push(std::move(task));
        }
        return res;
    }
    void run_pending_task()

```

```

    {
        task_type task;
        if(pop_task_from_local_queue(task) || #7
            pop_task_from_pool_queue(task) || #8
            pop_task_from_other_thread_queue(task)) #9
        {
            task();
        }
        else
        {
            std::this_thread::yield();
        }
    }
};

```

This code is very similar to listing 9.6. The first difference is that each thread has a `work_stealing_queue` rather than a plain `std::queue<>` #2. When each thread is created, rather than allocating its own work queue, the pool constructor allocates one #6, which is then stored in the list of work queues for this pool #1. The index of the queue in the list is then passed in to the thread function and used to retrieve the pointer to the queue #3. This means that the thread pool can access the queue when trying to steal a task for a thread that has no work to do. `run_pending_task()` will now try to take a task from its thread's own queue #7, take a task from the pool queue #8, or take a task from the queue of another thread #9.

`pop_task_from_other_thread_queue()` #4 iterates through the queues belonging to all the threads in the pool, trying to steal a task from each in turn. In order to avoid every thread trying to steal from the first thread in the list, each thread starts at the next thread in the list, by offsetting the index of the queue to check by its own index #5.

Now you have a working thread pool that's good for many potential uses. Of course, there are still a myriad of ways to improve it for any particular usage, but that's left as an exercise for the reader. One aspect in particular that hasn't been explored is the idea of dynamically resizing the thread pool to ensure that there's optimal CPU usage even when threads are blocked waiting for something such as I/O or a mutex lock.

Next on the list of "advanced" thread-management techniques is interrupting threads.

9.2 Interrupting threads

In many situations it's desirable to signal to a long-running thread that it's time to stop. This might be because it's a worker thread for a thread pool and the pool is now being destroyed, or it might be because the work being done by the thread has been explicitly canceled by the user, or a myriad of other reasons. Whatever the reason, the idea is the same: you need to signal from one thread that another should stop before it reaches the natural end of its processing, and you need to do this in a way that allows that thread to terminate nicely rather than abruptly pulling the rug from under it.

You could potentially design a separate mechanism for every case where you need to do this, but that would be overkill. Not only does a common mechanism make it easier to write the code on subsequent occasions, but it can allow you to write code that can be interrupted,

without having to worry about where that code is being used. The C++11 Standard doesn't provide such a mechanism, but it's relatively straightforward to build one. Let's look at how you can do that, starting from the point of view of the interface for launching and interrupting a thread rather than that of the thread being interrupted.

9.2.1 Launching and interrupting another thread

To start with, let's look at the external interface. What do you need from an interruptible thread? At the basic level, all you need is the same interface as you have for `std::thread`, with an additional `interrupt()` function:

```
class interruptible_thread
{
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f);
    void join();
    void detach();
    bool joinable() const;
    void interrupt();
};
```

Internally, you can use `std::thread` to manage the thread itself and use some custom data structure to handle the interruption. Now, what about from the point of view of the thread itself? At the most basic level you want to be able to say, "I can be interrupted here"—you want an *interruption point*. For this to be usable without having to pass down additional data, it needs to be a simple function that can be called without any parameters: `interruption_point()`. This implies that the interruption-specific data structure needs to be accessible through a `thread_local` variable that's set when the thread is started, so that when a thread calls your `interruption_point()` function, it checks the data structure for the currently executing thread. We'll look at the implementation of `interruption_point()` later.

This `thread_local` flag is the primary reason you can't just use plain `std::thread` to manage the thread; it needs to be allocated in a way that the `interruptible_thread` instance can access, as well as the newly started thread. You can do this by wrapping the supplied function before you pass it to `std::thread` to actually launch the thread in the constructor, as shown in the next listing.

Listing 9.9 Basic implementation of `interruptible_thread`

```
class interrupt_flag
{
public:
    void set();
    bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;      #1
class interruptible_thread
{
    std::thread internal_thread;
    interrupt_flag* flag;
```

```

public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f)
    {
        std::promise<interrupt_flag*> p;                                #2
        internal_thread=std::thread([f,&p]{
            p.set_value(&this_thread_interrupt_flag);                  #3
            f();                                                       #4
        });
        flag=p.get_future().get();                                       #5
    }
    void interrupt()
    {
        if(flag)
        {
            flag->set();                                              #6
        }
    }
};

```

The supplied function `f` is wrapped in a lambda function #3 that holds a copy of `f` and a reference to the local promise `p` #2. The lambda sets the value of the promise to the address of the `this_thread_interrupt_flag` (which is declared `thread_local` #1) for the new thread before invoking the copy of the supplied function #4. The calling thread then waits for the future associated with the promise to become ready and stores the result in the `flag` member variable #5. Note that even though the lambda is running on the new thread and has a dangling reference to the local variable `p`, this is OK because the `interruptible_thread` constructor waits until `p` is no longer referenced by the new thread before returning. Note that this implementation doesn't take account of handling joining with the thread, or detaching it. You need to ensure that the `flag` variable is cleared when the thread exits, or is detached, to avoid a dangling pointer.

The `interrupt()` function is then relatively straightforward: if you have a valid pointer to an interrupt flag, you have a thread to interrupt, so you can just set the flag #6. It's then up to the interrupted thread what it does with the interruption. Let's explore that next.

9.2.2 Detecting that a thread has been interrupted

You can now set the interruption flag, but that doesn't do you any good if the thread doesn't actually check whether it's being interrupted. In the simplest case you can do this with an `interruption_point()` function; you can call this function at a point where it's safe to be interrupted, and it throws a `thread_interrupted` exception if the flag is set:

```

void interruption_point()
{
    if(this_thread_interrupt_flag.is_set())
    {
        throw thread_interrupted();
    }
}

```

You can use such a function by calling it at convenient points within your code:

```

void foo()
{
    while(!done)
    {
        interruption_point();
        process_next_item();
    }
}

```

Although this works, it's not ideal. Some of the best places for interrupting a thread are where it's blocked waiting for something, which means that the thread isn't running in order to call `interruption_point()`! What you need here is a means for waiting for something in an interruptible fashion.

9.2.3 Interrupting a condition variable wait

OK, so you can detect interruptions at carefully chosen places in your code, with explicit calls to `interruption_point()`, but that doesn't help when you want to do a blocking wait, such as waiting for a condition variable to be notified. You need a new function—`interruptible_wait()`—which you can then overload for the various things you might want to wait for, and you can work out how to interrupt the waiting. I've already mentioned that one thing you might be waiting for is a condition variable, so let's start there: what do you need to do in order to be able to interrupt a wait on a condition variable? The simplest thing that would work is to notify the condition variable once you've set the interrupt flag, and put an interruption point immediately after the wait. But for this to work, you'd have to notify all threads waiting on the condition variable in order to ensure that your thread of interest wakes up. Waiters have to handle spurious wake-ups anyway, so other threads would handle this the same as a spurious wake-up—they wouldn't be able to tell the difference. The `interrupt_flag` structure would need to be able to store a pointer to a condition variable so that it can be notified in a call to `set()`. One possible implementation of `interruptible_wait()` for condition variables might look like the following listing.

Listing 9.10 A broken version of `interruptible_wait` for `std::condition_variable`

```

void interruptible_wait(std::condition_variable& cv,
                       std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);      #1
    cv.wait(lk);                                                 #2
    this_thread_interrupt_flag.clear_condition_variable();       #3
    interruption_point();
}

```

Assuming the presence of some functions for setting and clearing an association of a condition variable with an interrupt flag, this code is nice and simple. It checks for interruption, associates the condition variable with the `interrupt_flag` for the current thread #1, waits on the condition variable #2, clears the association with the condition variable #3, and checks for interruption again. If the thread is interrupted during the wait on the condition

variable, the interrupting thread will broadcast the condition variable and wake you from the wait, so you can check for interruption. Unfortunately, this code is *broken*: there are two problems with it. The first problem is relatively obvious if you have your exception safety hat on: `std::condition_variable::wait()` can throw an exception, so you might exit the function without removing the association of the interrupt flag with the condition variable. This is easily fixed with a structure that removes the association in its destructor.

The second, less obvious problem is that there's a race condition. If the thread is interrupted after the initial call to `interruption_point()`, but before the call to `wait()`, then it doesn't matter whether the condition variable has been associated with the interrupt flag, because *the thread isn't waiting and so can't be woken by a notify on the condition variable*. You need to ensure that the thread can't be notified between the last check for interruption and the call to `wait()`. Without delving into the internals of `std::condition_variable`, you have only one way of doing that: use the mutex held by `lk` to protect this too, which requires passing it in on the call to `set_condition_variable()`. Unfortunately, this creates its own problems: you'd be passing a reference to a mutex whose lifetime you don't know to another thread (the thread doing the interrupting) for that thread to lock (in the call to `interrupt()`), without knowing whether that thread has locked the mutex already when it makes the call. This has the potential for deadlock *and* the potential to access a mutex after it has already been destroyed, so it's a nonstarter. It would be rather too restrictive if you couldn't *reliably* interrupt a condition variable wait—you can do almost as well without a special `interruptible_wait()`—so what other options do you have? One option is to put a timeout on the wait; use `wait_for()` rather than `wait()` with a small timeout value (such as 1 ms). This puts an upper limit on how long the thread will have to wait before it sees the interruption (subject to the tick granularity of the clock). If you do this, the waiting thread will see rather more “spurious” wakes resulting from the timeout, but it can't easily be helped. Such an implementation is shown in the next listing, along with the corresponding implementation of `interrupt_flag`.

Listing 9.11 Using a timeout in `interruptible_wait` for `std::condition_variable`

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::mutex set_clear_mutex;
public:
    interrupt_flag():
        thread_cond(0)
    {}
    void set()
    {
        flag.store(true,std::memory_order_relaxed);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        if(thread_cond)
        {
            thread_cond->notify_all();
        }
    }
}
```

```

    }
    bool is_set() const
    {
        return flag.load(std::memory_order_relaxed);
    }
    void set_condition_variable(std::condition_variable& cv)
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=&cv;
    }
    void clear_condition_variable()
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=0;
    }
    struct clear_cv_on_destruct
    {
        ~clear_cv_on_destruct()
        {
            this_thread_interrupt_flag.clear_condition_variable();
        }
    };
};

void interruptible_wait(std::condition_variable& cv,
                      std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct guard;
    interruption_point();
    cv.wait_for(lk,std::chrono::milliseconds(1));
    interruption_point();
}

```

If you have the predicate that's being waited for, then the 1 ms timeout can be completely hidden inside the predicate loop:

```

template<typename Predicate>
void interruptible_wait(std::condition_variable& cv,
                      std::unique_lock<std::mutex>& lk,
                      Predicate pred)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct guard;
    while(!this_thread_interrupt_flag.is_set() && !pred())
    {
        cv.wait_for(lk,std::chrono::milliseconds(1));
    }
    interruption_point();
}

```

This will result in the predicate being checked more often than it might otherwise be, but it's easily used in place of a plain call to `wait()`. The variants with timeouts are easily implemented: wait either for the time specified, or 1 ms, whichever is shortest. OK, so

`std::condition_variable` waits are now taken care of; what about `std::condition_variable_any`? Is this the same, or can you do better?

9.2.4 Interrupting a wait on `std::condition_variable_any`

`std::condition_variable_any` differs from `std::condition_variable` in that it works with *any* lock type rather than just `std::unique_lock<std::mutex>`. It turns out that this makes things much easier, and you *can* do better with `std::condition_variable_any` than you could with `std::condition_variable`. Because it works with *any* lock type, you can build your own lock type that locks/unlocks both the internal `set_clear_mutex` in your `interrupt_flag` and the lock supplied to the wait call, as shown here.

Listing 9.12 `interruptible_wait` for `std::condition_variable_any`

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::condition_variable_any* thread_cond_any;
    std::mutex set_clear_mutex;
public:
    interrupt_flag():
        thread_cond(0),thread_cond_any(0)
    {}
    void set()
    {
        flag.store(true,std::memory_order_relaxed);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        if(thread_cond)
        {
            thread_cond->notify_all();
        }
        else if(thread_cond_any)
        {
            thread_cond_any->notify_all();
        }
    }
    template<typename Lockable>
    void wait(std::condition_variable_any& cv,Lockable& lk)
    {
        struct custom_lock
        {
            interrupt_flag* self;
            Lockable& lk;
            custom_lock(interrupt_flag* self_,
                        std::condition_variable_any& cond,
                        Lockable& lk_):
                self(self_),lk(lk_)
            {
                self->set_clear_mutex.lock();           #1
                self->thread_cond_any=&cond;           #2
            }
            void unlock()                         #3
            {
                lk.unlock();
                self->set_clear_mutex.unlock();
            }
        };
        cv.wait(custom_lock());
    }
};
```

```

    }
    void lock()
    {
        std::lock(self->set_clear_mutex,lk);      #4
    }
    ~custom_lock()
    {
        self->thread_cond_any=0;                  #5
        self->set_clear_mutex.unlock();
    }
};

custom_lock cl(this,cv,lk);
interruption_point();
cv.wait(cl);
interruption_point();
}
// rest as before
};

template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv,
Lockable& lk)
{
    this_thread_interrupt_flag.wait(cv,lk);
}

```

Your custom lock type acquires the lock on the internal `set_clear_mutex` when it's constructed #1 and then sets the `thread_cond_any` pointer to refer to the `std::condition_variable_any` passed in to the constructor #2. The `Lockable` reference is stored for later; this must already be locked. You can now check for an interruption without worrying about races. If the interrupt flag is set at this point, it was set before you acquired the lock on `set_clear_mutex`. When the condition variable calls your `unlock()` function inside `wait()`, you unlock the `Lockable` object *and the internal `set_clear_mutex`* #3. This allows threads that are trying to interrupt you to acquire the lock on `set_clear_mutex` and check the `thread_cond_any` pointer *once you're inside the wait() call* but not before. This is exactly what you were after (but couldn't manage) with `std::condition_variable`. Once `wait()` has finished waiting (either because it was notified or because of a spurious wake), it will call your `lock()` function, which again acquires the lock on the internal `set_clear_mutex` and the lock on the `Lockable` object #4. You can now check again for interruptions that happened during the `wait()` call before clearing the `thread_cond_any` pointer in your `custom_lock` destructor #5, where you also unlock the `set_clear_mutex`.

9.2.5 Interrupting other blocking calls

That rounds up interrupting condition variable waits, but what about other blocking waits: mutex locks, waiting for futures, and the like? In general you have to go for the timeout option you used for `std::condition_variable` because there's no way to interrupt the wait short of actually fulfilling the condition being waited for, without access to the internals of the mutex or future. But with those other things you do know what you're waiting for, so you can loop within the `interruptible_wait()` function. As an example, here's an overload of `interruptible_wait()` for a `std::future<>`:

```

template<typename T>
void interruptible_wait(std::future<T>& uf)
{
    while(!this_thread_interrupt_flag.is_set())
    {
        if(uf.wait_for(lk,std::chrono::milliseconds(1))==std::future_status::ready)
            break;
    }
    interruption_point();
}

```

This waits until either the interrupt flag is set or the future is ready but does a blocking wait on the future for 1 ms at a time. This means that on average it will be around 0.5 ms before an interrupt request is acknowledged, assuming a high-resolution clock. The `wait_for` will typically wait at least a whole clock tick, so if your clock ticks every 15 ms, you'll end up waiting around 15 ms rather than 1 ms. This may or may not be acceptable, depending on the circumstances. You can always reduce the timeout if necessary (and the clock supports it). The downside of reducing the timeout is that the thread will wake more often to check the flag, and this will increase the task-switching overhead.

OK, so we've looked at how you might detect interruption, with the `interruption_point()` and `interruptible_wait()` functions, but how do you handle that?

9.2.6 Handling interruptions

From the point of view of the thread being interrupted, an interruption is just a `thread_interrupted` exception, which can therefore be handled just like any other exception. In particular, you can catch it in a standard `catch` block:

```

try
{
    do_something();
}
catch(thread_interrupted&)
{
    handle_interruption();
}

```

This means that you could catch the interruption, handle it in some way, and then carry on regardless. If you do this, and another thread calls `interrupt()` again, your thread will be interrupted again the next time it calls an interruption point. You might want to do this if your thread is performing a series of independent tasks; interrupting one task will cause that task to be abandoned, and the thread can then move on to performing the next task in the list.

Because `thread_interrupted` is an exception, all the usual exception-safety precautions must also be taken when calling code that can be interrupted, in order to ensure that resources aren't leaked, and your data structures are left in a coherent state. Often, it will be desirable to let the interruption terminate the thread, so you can just let the exception propagate up. But if you let exceptions propagate out of the thread function passed to the `std::thread` constructor, `std::terminate()` will be called, and the whole program will be

terminated. In order to avoid having to remember to put a catch (thread_interrupted) handler in every function you pass to interruptible_thread, you can instead put that catch block inside the wrapper you use for initializing the interrupt_flag. This makes it safe to allow the interruption exception to propagate unhandled, because it will then terminate just that individual thread. The initialization of the thread in the interruptible_thread constructor now looks like this:

```
internal_thread=std::thread([f,&p]{  
    p.set_value(&this_thread_interrupt_flag);  
    try  
    {  
        f();  
    }  
    catch(thread_interrupted const&)  
    {}  
});
```

Let's now look at a concrete example where interruption is useful.

9.2.7 Interrupting background tasks on application exit

Consider for a moment a desktop search application. As well as interacting with the user, the application needs to monitor the state of the filesystem, identifying any changes and updating its index. Such processing is typically left to a background thread, in order to avoid affecting the responsiveness of the GUI. This background thread needs to run for the entire lifetime of the application; it will be started as part of the application initialization and left to run until the application is shut down. For such an application this is typically only when the machine itself is being shut down, because the application needs to run the whole time in order to maintain an up-to-date index. In any case, when the application is being shut down, you need to close down the background threads in an orderly manner; one way to do this is by interrupting them.

The following listing shows a sample implementation of the thread-management parts of such a system.

Listing 9.13 Monitoring the filesystem in the background

```
std::mutex config_mutex;  
std::vector<interruptible_thread> background_threads;  
void background_thread(int disk_id)  
{  
    while(true)  
    {  
        interruption_point(); #1  
        fs_change fsc=get_fs_changes(disk_id); #2  
        if(fsc.has_changes())  
        {  
            update_index(fsc); #3  
        }  
    }  
}  
void start_background_processing()
```

```

{
    background_threads.push_back(
        interruptible_thread(background_thread,disk_1));
    background_threads.push_back(
        interruptible_thread(background_thread,disk_2));
}
int main()
{
    start_background_processing();                      #4
    process_gui_until_exit();                         #5
    std::unique_lock<std::mutex> lk(config_mutex);
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].interrupt();            #6
    }
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].join();                 #7
    }
}

```

At startup, the background threads are launched #4. The main thread then proceeds with handling the GUI #5. When the user has requested that the application exit, the background threads are interrupted #6, and then the main thread waits for each background thread to complete before exiting #7. The background threads sit in a loop, checking for disk changes #8 and updating the index #3. Every time around the loop they check for interruption by calling `interruption_point()` #1.

Why do you interrupt all the threads before waiting for any? Why not interrupt each and then wait for it before moving on to the next? The answer is *concurrency*. Threads will likely not finish immediately when they're interrupted, because they have to proceed to the next interruption point and then run any destructor calls and exception-handling code necessary before they exit. By joining with each thread immediately, you therefore cause the interrupting thread to wait, *even though it still has useful work it could do*—interrupt the other threads. Only when you have no more work to do (all the threads have been interrupted) do you wait. This also allows all the threads being interrupted to process their interruptions in parallel and potentially finish sooner.

This interruption mechanism could easily be extended to add further interruptible calls or to disable interruptions across a specific block of code, but this is left as an exercise for the reader.

9.3 Summary

In this chapter, we've looked at various “advanced” thread-management techniques: thread pools and interrupting threads. You've seen how the use of local work queues and work stealing can reduce the synchronization overhead and potentially improve the throughput of the thread pool and how running other tasks from the queue while waiting for a subtask to complete can eliminate the potential for deadlock.

We've also looked at various ways of allowing one thread to interrupt the processing of another, such as the use of specific interruption points and functions that perform what would otherwise be a blocking wait in a way that can be interrupted.

10

Parallel Algorithms

This chapter covers

- using the C++17 parallel algorithms

In the last chapter we looked at advanced thread management and thread pools, and in chapter 8 we looked at designing concurrent code, using parallel versions of some algorithms as examples. In this chapter, we're going to look at the parallel algorithms provided by the C++17 standard, so let's start, without further ado.

10.1 Parallelizing the Standard Library Algorithms

The C++17 standard added the concept of “parallel algorithms” to the C++ Standard Library. These are additional overloads of many of the functions that operate on ranges, such as `std::find`, `std::transform` or `std::reduce`. The parallel versions have the same signature as the “normal” single-threaded versions, except for the addition of a new first parameter which specifies the *execution policy* to use. e.g.

```
std::vector<int> my_data;  
std::sort(std::execution::par,my_data.begin(),my_data.end());
```

The execution policy of `std::execution::par` indicates to the standard library that it is *allowed* to perform this call as a parallel algorithm, using multiple threads. Note that this is *permission*, not a *requirement* — the library may still execute the code on a single thread if it wishes. It is also important to note that by specifying an execution policy the requirements on the algorithm complexity have changed, and are usually slacker than the requirements for the normal serial algorithm. This is because parallel algorithms often do more total work in order to take advantage of the parallelism of the system — if you can divide the work across 100 processors, then you can still get an overall speed up of 50, even if the implementation does twice as much total work.

Before we get onto the algorithms themselves, let's take a look at the execution policies.

10.2 Execution Policies

The standard specifies 3 execution policies:

- `std::execution::sequenced_policy`,
- `std::execution::parallel_policy`, and
- `std::execution::parallel_unsequenced_policy`.

These are classes defined in the `<execution>` header. The header also defines 3 corresponding policy objects to pass to the algorithms:

- `std::execution::seq`,
- `std::execution::par`, and
- `std::execution::par_unseq`

You cannot rely on being able to construct objects from these policy classes yourself, except by copying these 3 objects, because they might have special initialization requirements. Implementations may also define additional execution policies that have implementation-specific behaviour. You cannot define your own execution policies.

The consequences of these policies on the behavior of the algorithms is described below. Any given implementation is also allowed to provide any additional execution policies, with whatever semantics they wish. Let's now take a look at the consequences of using one of the standard execution policies, starting with the general changes for all algorithm overloads that take an exception policy.

10.2.1 General effects of specifying an execution policy

If you pass an execution policy to one of the standard library algorithms, then the behaviour of that algorithm is now governed by the execution policy. This affects several aspects of the behaviour:

- The algorithm's complexity,
- The behaviour when an exception is thrown, and
- Where, how and when the steps of the algorithm are executed.

EFFECTS ON ALGORITHM COMPLEXITY

If an execution policy is supplied to an algorithm, then that algorithm's complexity may be changed: in addition to the scheduling overhead of managing the parallel execution, many parallel algorithms will perform more of the core operations of the algorithm (whether that is *swaps*, *comparisons*, or *applications of a supplied function object*), with the intention that this provides an overall improvement in the performance in terms of total elapsed time.

The precise details of the complexity change will vary with each algorithm, but the general policy is that if an algorithm specifies something will happen exactly *some-expression* times, or at most *some-expression* times, then the overload with an execution policy will slacken that requirement to $O(\text{some-expression})$. This means that the overload with an execution policy may perform some multiple of the number of operations performed by its counterpart without an execution policy, where that multiple will depend on the internals of the library and the platform, rather than the data supplied to the algorithm.

EXCEPTIONAL BEHAVIOUR

If an exception is thrown during execution of an algorithm with an execution policy, then the consequences are determined by the execution policy. All the standard-supplied execution policies will call `std::terminate` if there are any uncaught exceptions. The only exception that may be thrown by a call to a standard library algorithm with one of the standard execution policies is `std::bad_alloc`, which is thrown if the library cannot obtain sufficient memory resources for its internal operations. For example, the following call to `std::for_each`, **without** an execution policy, will propagate the exception:

```
std::for_each(v.begin(),v.end(),[](auto x){ throw my_exception(); });
```

whereas the corresponding call **with** an execution policy will terminate the program:

```
std::for_each(
    std::execution::seq,v.begin(),v.end(),
    [] (auto x){ throw my_exception();});
```

This is one of the key differences between using `std::execution::seq` and not providing an execution policy.

WHERE AND WHEN ALGORITHM STEPS ARE EXECUTED

This is the fundamental aspect of an execution policy, and is the only aspect that differs between the standard execution policies. The policy specifies which execution agents are used to perform the steps of the algorithm, be these “normal” threads, vector streams, GPU threads, or anything else. The execution policy will also specify whether there are any ordering constraints on how the algorithm steps are run: whether or not they are run in any particular order, whether or not parts of separate algorithm steps may be interleaved with each other, or run in parallel with each other, and so forth.

The details for each of the standard execution policies are given below, starting with the most basic policy: `std::execution::sequenced_policy`.

10.2.2 `std::execution::sequenced_policy`

The sequenced policy is not a policy for parallelism: using it forces the implementation to perform all operations on the thread that called the function, so there is no parallelism at all. However, it is still an execution policy, and therefore has the same consequences on algorithmic complexity and the effect of exceptions as the other standard policies.

Not only must all operations be performed on the same thread, but they must be performed in some definite order, so they are not interleaved. The precise order is unspecified, and may be different between different invocations of the function. In particular, **the order of execution of the operations is not guaranteed to be the same as that of the corresponding overload without an execution policy**. For example, the following call to `std::for_each` will populate the vector with the numbers 1-1000, in an unspecified order. This is in contrast to the overload without an execution policy which will store the numbers in order:

```
std::vector<int> v(1000);
```

```
int count=0;
std::for_each(std::execution::seq,v.begin(),v.end(),
[&](int& x){ x=++count; });
```

Of course, the numbers **may** be stored in order, but you cannot rely on it.

This means that the sequenced policy therefore imposes very little requirements on the on the iterators, values and callable objects used with the algorithm: they may freely use synchronization mechanisms, and may rely on all operations being invoked on the same thread, though they cannot rely on the order of such operations.

10.2.3 std::execution::parallel_policy

The parallel policy provides basic parallel execution across some number of threads. Operations may be performed either on the thread that invoked the algorithm, or on threads created by the library. Operations performed on a given thread must be performed in a definite order, and not interleaved, but the precise order is unspecified, and may vary between invocations. A given operation will run on a fixed thread for its entire duration.

This imposes additional requirements on the iterators, values and callable objects used with the algorithm over the sequenced policy: they must not cause data races if invoked in parallel, and must not rely on being run on the same thread as any other operation, or indeed rely on **not** being run on the same thread as any other operation.

You can use the parallel execution policy for the vast majority of cases where you would have used a standard library algorithm without an execution policy. It is only where there is specific ordering between elements that is required, or unsynchronized access to shared data that there is a problem. Incrementing all the values in a vector can thus be done in parallel:

```
std::for_each(std::execution::par,v.begin(),v.end(),[](auto& x){++x;});
```

but the previous example of populating a vector is **not OK** if done with the parallel execution policy; specifically, it is **undefined behaviour**:

```
std::for_each(std::execution::par,v.begin(),v.end(),
[&](int& x){ x=++count; });
```

Here, the variable `count` is **modified** from every invocation of the lambda, so if the library was to execute the lambdas across multiple threads, this would be a data race, and thus undefined behaviour. The requirements for `std::execution::parallel_policy` preempt this: it is undefined behaviour to make the call above, **even if the library doesn't use multiple threads for this call**. Whether or not something exhibits undefined behaviour is thus a static property of the call, rather than dependent on implementation details of the library. Synchronization between the function invocations **is** permitted, however, so we could make this defined behaviour again either by making `count` an `std::atomic<int>` rather than a plain `int`, or by using a mutex. In this case, that would likely defeat the point of using the parallel execution policy, since that would serialize all the calls, but in the general case it would allow for synchronized access to shared state.

10.2.4 `std::execution::parallel_unsequenced_policy`

The parallel unsequenced policy provides the library with the most scope for parallelizing the algorithm in exchange for imposing the strictest requirements on the iterators, values and callable objects used with the algorithm.

An algorithm invoked with the parallel unsequenced policy may perform the algorithm steps on unspecified threads of execution, unordered and unsequenced with respect to one-another. This means that **operations may now be interleaved with each other on a single thread**, such that a second operation is started on the same thread before the first has finished, and **may be migrated between threads**, so a given operation may start on one thread, run further on a second thread, and complete on a third.

If you use the parallel unsequenced policy, then the operations invoked on the iterators, values and callable objects supplied to the algorithm must not use any form of synchronization or call any function that *synchronizes-with* another, or any function such that some other code *synchronizes-with* it.

This basically means that the operations must only operate on the relevant element, or any data that can be accessed based on that element, and must **not** modify any state that is shared between threads, or between elements.

We'll flesh these out with some examples later. For now, let's take a look at the parallel algorithms themselves.

10.3 The Parallel Algorithms from the C++ Standard Library

Most of the algorithms from the `<algorithm>` and `<numeric>` headers have overloads that take an execution policy. This comprises: `all_of`, `any_of`, `none_of`, `for_each`, `for_each_n`, `find`, `find_if`, `find_end`, `find_first_of`, `adjacent_find`, `count`, `count_if`, `mismatch`, `equal`, `search`, `search_n`, `copy`, `copy_n`, `copy_if`, `move`, `swap_ranges`, `transform`, `replace`, `replace_if`, `replace_copy`, `replace_copy_if`, `fill`, `fill_n`, `generate`, `generate_n`, `remove`, `remove_if`, `remove_copy`, `remove_copy_if`, `unique`, `unique_copy`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `is_partitioned`, `partition`, `stable_partition`, `partition_copy`, `sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`, `is_sorted`, `is_sorted_until`, `nth_element`, `merge`, `inplace_merge`, `includes`, `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`, `is_heap`, `is_heap_until`, `min_element`, `max_element`, `minmax_element`, `lexicographical_compare`, `reduce`, `transform_reduce`, `exclusive_scan`, `inclusive_scan`, `transform_exclusive_scan`, `transform_inclusive_scan`, and `adjacent_difference`.

That's quite a list: pretty much every algorithm in the C++ Standard Library that could be parallelized is in the list. Notable exceptions are things like `std::accumulate`, which is strictly a serial accumulation, but its generalized counterpart in `std::reduce` **does** appear in the list — with a suitable warning in the standard that if the reduction operation is not both associative and commutative, then the result may be non-deterministic due to the unspecified order of operations.

For each of the algorithms in the list, every “normal” overload has a new variant which takes an execution policy as the first argument — the corresponding arguments for the “normal” overload then come after this execution policy. For example, `std::sort` has two “normal” overloads without an execution policy:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(
    RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

It therefore also has 2 overloads **with** an execution policy:

```
template<class ExecutionPolicy, class RandomAccessIterator>
void sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

There is one important difference between the signatures with and without the execution policy argument, which only impacts some algorithms: if the “normal” algorithm allows *Input Iterators* or *Output Iterators*, then the overloads with an execution policy require *Forward Iterators* instead. This is because *Input Iterators* are fundamentally single-pass: you can only access the current element, and you cannot store iterators to previous elements. Similarly, *Output Iterators* only allow writing to the current element: you cannot advance them to write a later element, and then back-track to write a previous one.

Iterator Categories in the C++ Standard Library

The C++ Standard Library defines 5 categories of iterators: *Input Iterators*, *Output Iterators*, *Forward Iterators*, *Bidirectional Iterators*, and *Random Access Iterators*.

Input Iterators are single-pass iterators for retrieving values. They are typically used for things like input from a console or network, or generated sequences. Advancing an *Input Iterator* invalidates any copies of that iterator.

Output Iterators are single-pass iterators for writing values. They are typically used for output to files, or adding values to a container. Advancing an *Output Iterator* invalidates any copies of that iterator.

Forward Iterators are multi-pass iterators for one-way iteration through persistent data. Though you can't make an iterator go back to a previous element, you can store copies and use them to reference earlier elements. *Forward Iterators* return real references to the elements, and so can be used for both reading and writing (if the target is non-const).

Bidirectional Iterators are multi-pass iterators like *Forward Iterators*, but they can also be made to go backwards to access previous elements.

Random Access Iterators are multi-pass iterators that can forward and backwards like *Bidirectional Iterators*, but they can go forward and backward in steps larger than a single element, and you can directly access elements at an offset, using the array index operator.

Thus, given the “normal” signature for `std::copy`:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator first, InputIterator last, OutputIterator result);
```

The overload with an execution policy is:

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(
    ExecutionPolicy&& policy,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result);
```

Though the naming of the template parameters doesn't carry any direct consequence from the compiler's perspective, it does from the C++ Standard's perspective: the names of the template parameters for Standard Library algorithms denote semantic constraints on the types, and the algorithms will rely on the operations implied by those constraints existing, with the specified semantics. In the case of *Input Iterators* vs *Forward Iterators*, the former allows dereferencing the iterator to return a proxy type which is convertible to the value type of the iterator, whereas the latter requires that dereferencing the iterator returns a real reference to the value, and that all equal iterators return a reference to the same value.

This is important for parallelism: it means that the iterators can be freely copied around, and used equivalently. Also, the requirement that incrementing a *Forward Iterator* does not invalidate other copies is important, as it means that separate threads can operate on their own copies of the iterators, incrementing them when required, without concern about invalidating the iterators held by the other threads. If the overload with an execution policy allowed use of *Input Iterators*, this would force any threads to serialize access to the one and only iterator that was used for reading from the source sequence, which obviously limits the potential for parallelism.

Let's have a look at some concrete examples.

10.3.1 Examples of using parallel algorithms

The simplest possible example surely has to be the parallel loop: do something for each element of a container. This is the classic example of an embarrassingly parallel scenario: each item is independent, so we have the maximum possibility of parallelism. With a compiler that supports OpenMP, we might write:

```
#pragma omp parallel for
for(unsigned i=0;i<v.size();++i){
    do_stuff(v[i]);
}
```

With the C++ Standard Library algorithms, we can instead write:

```
std::for_each(std::execution::par,v.begin(),v.end(),do_stuff);
```

This will divide up the elements of the range between the internal threads created by the library, and invoke `do_stuff(x)` on each element `x` in the range. Quite how those elements are divided between the threads is an implementation detail.

CHOICE OF EXECUTION POLICY

`std::execution::par` is the policy that you will likely want to use most often, unless your implementation provides a non-standard policy better suited to your needs. If your code is suitable for parallelization, then it should work with `std::execution::par`. In some circumstances, you may be able to use `std::execution::par_unseq` instead. This may do nothing at all (none of the standard execution policies make a guarantee about the level of parallelism that will be attained), but it may give the library additional scope to improve the performance of the code by reordering and interleaving the tasks, in exchange for the tighter requirements on your code. Most notable of these tighter requirements is that there is no synchronization used in accessing the elements, or performing the operations on the elements. This means that you cannot use mutexes or atomic variables, or any of the other mechanisms described in previous chapters, to ensure that accesses from multiple threads are safe; instead, you must rely on the algorithm itself not accessing the same element from multiple threads, and use external synchronization outside the call to the parallel algorithm to prevent other threads accessing the data.

The example from listing 10.1 shows some code that can be used with `std::execution::par`, but not `std::execution::par_unseq`. The use of the internal mutex for synchronization means that attempting to use `std::execution::par_unseq` would be undefined behavior.

Listing 10.1 Parallel algorithms on a class with internal synchronization

```
class X{
    mutable std::mutex m;
    int data;
public:
    X():data(0){}
    int get_value() const{
        std::lock_guard guard(m);
        return data;
    }
    void increment(){
        std::lock_guard guard(m);
        ++data;
    }
};
void increment_all(std::vector<X>& v){
    std::for_each(std::execution::par,v.begin(),v.end(),
        [](&x){
            x.increment();
        });
}
```

Listing 10.2 shows an alternative that can be used with `std::execution::par_unseq`. In this case, the internal per-element mutex has been replaced with a whole-container mutex.

Listing 10.2 Parallel algorithms on a class without internal synchronization

```
class Y{
    int data;
public:
    Y():data(0){}
    int get_value() const{
        return data;
    }
    void increment(){
        ++data;
    }
};
class ProtectedY{
    std::mutex m;
    std::vector<Y> v;
public:
    void lock(){
        m.lock();
    }
    void unlock(){
        m.unlock();
    }
    std::vector<Y>& get_vec(){
        return v;
    }
};
void increment_all(ProtectedY& data){
    std::lock_guard guard(data);
    auto& v=data.get_vec();
    std::for_each(std::execution::par_unseq,v.begin(),v.end(),
        [](&y){
            y.increment();
        });
}
```

The element accesses in listing 10.2 now have no synchronization, and it is thus safe to use `std::execution::par_unseq`. The downside is that concurrent accesses from other threads outside the parallel algorithm invocation must now wait for the entire operation to complete, rather than the per-element granularity of listing 10.1.

Let's now take a look at a more realistic example of how the parallel algorithms might be used: counting visits to a website.

10.3.2 Counting visits

Suppose you run a busy website, such that the logs contain millions of entries, and you want to process those logs to see aggregate data: how many visits per page, where do those visits come from, which browsers were used to access the website, and so forth. Analysing these logs essentially has two parts: processing each line to extract the relevant information, and aggregating the results together. This is an ideal scenario for using parallel algorithms, since processing each individual line is entirely independent of everything else, and aggregating the results can be done piecemeal, provided the final totals are correct.

In particular, this is the sort of task that `transform_reduce` is designed for. Listing 10.3 shows how this could be used for this task.

Listing 10.3: Using transform_reduce to count visits to pages of a website

```
#include <vector>
#include <string>
#include <unordered_map>
#include <numeric>

struct log_info {
    std::string page;
    time_t visit_time;
    std::string browser;
    // any other fields
};

extern log_info parse_log_line(std::string const &line); #1

using visit_map_type= std::unordered_map<std::string, unsigned long long>;

visit_map_type
count_visits_per_page(std::vector<std::string> const &log_lines) {

    struct combine_visits {
        visit_map_type
        operator()(visit_map_type lhs, visit_map_type rhs) const { #3
            if(lhs.size() < rhs.size())
                std::swap(lhs, rhs);
            for(auto const &entry : rhs) {
                lhs[entry.first] += entry.second;
            }
            return lhs;
        }

        visit_map_type operator()(log_info log,visit_map_type map) const{#4
            ++map[log.page];
            return map;
        }
        visit_map_type operator()(visit_map_type map,log_info log) const{#5
            ++map[log.page];
            return map;
        }
        visit_map_type operator()(log_info log1,log_info log2) const{#6
            visit_map_type map;
            ++map[log1.page];
            ++map[log2.page];
            return map;
        }
    };
}

return std::transform_reduce(                                     #2
    std::execution::par, log_lines.begin(), log_lines.end(),
    visit_map_type(), combine_visits(), parse_log_line);
}
```

Assuming we've got some function `parse_log_line` to extract the relevant information from a log entry (#1), our `count_visits_per_page` function is actually a very simple wrapper around a call to `std::transform_reduce` (#2). The complexity comes from the *reduction* operation: we need to be able to combine two `log_info` structures to produce a map, a `log_info` structure and a map (either way round), and two maps. This therefore

means that our `combine_visits` function object needs 4 overloads of the function call operator (#3, #4, #5, #6), which precludes doing it with a simple lambda, even though the implementation of these 4 overloads is actually very simple.

The implementation of `std::transform_reduce` will therefore use the available hardware to perform this calculation in parallel (since we passed `std::execution::par`). Writing such an algorithm manually is non-trivial, as we saw in the previous chapter, so this allows you to delegate the hard work of implementing the parallelism to the Standard Library implementors, so you can focus on the required outcome.

10.4 Summary

In this chapter we've looked at the parallel algorithms available in the C++ Standard Library, and how to use them. We've looked at the various execution policies, and the impact your choice of execution policy has on the behaviour of the algorithm, and the restrictions it imposes on your code. We've then seen an example of how such an algorithm might be used in real code.

11

Testing and debugging multithreaded applications

This chapter covers

- Concurrency-related bugs
- Locating bugs through testing and code review
- Designing multithreaded tests
- Testing the performance of multithreaded code

Up to now, I've focused on what's involved in writing concurrent code—the tools that are available, how to use them, and the overall design and structure of the code. But there's a crucial part of software development that I haven't addressed yet: testing and debugging. If you're reading this chapter hoping for an easy way to test concurrent code, you're going to be sorely disappointed. Testing and debugging concurrent code is *hard*. What I *am* going to give you are some techniques that will make things easier, alongside issues that are important to think about.

Testing and debugging are like two sides of a coin—you subject your code to tests in order to find any bugs that might be there, and you debug it to remove those bugs. With luck, you only have to remove the bugs found by your own tests rather than bugs found by the end users of your application. Before we look at either testing or debugging, it's important to understand the problems that might arise, so let's look at those.

11.1 Types of concurrency-related bugs

You can get just about any sort of bug in concurrent code; it's not special in that regard. But some types of bugs are directly related to the use of concurrency and therefore of

particular relevance to this book. Typically, these concurrency-related bugs fall into two primary categories:

- Unwanted blocking
- Race conditions

These are huge categories, so let's divide them up a bit. First, let's look at unwanted blocking.

11.1.1 Unwanted blocking

What do I mean by unwanted blocking? First, a thread is *blocked* when it's unable to proceed because it's waiting for something. This is typically something like a mutex, a condition variable, or a future, but it could be waiting for I/O. This is a natural part of multithreaded code, but it's not always desirable—hence the problem of unwanted blocking. This leads us to the next question: why is this blocking unwanted? Typically, this is because some other thread is also waiting for the blocked thread to perform some action, and so that thread in turn is blocked. There are several variations on this theme:

- *Deadlock*—As you saw in chapter 3, in the case of deadlock one thread is waiting for another, which is in turn waiting for the first. If your threads deadlock, the tasks they're supposed to be doing won't get done. In the most visible cases, one of the threads involved is the thread responsible for the user interface, in which case the interface will cease to respond. In other cases, the interface will remain responsive, but some required task won't complete, such as a search not returning or a document not printing.
- *Livelock*—Livelock is similar to deadlock in that one thread is waiting for another, which is in turn waiting for the first. The key difference here is that the wait is not a blocking wait but an active checking loop, such as a spin lock. In serious cases, the symptoms are the same as deadlock (the app doesn't make any progress), except that the CPU usage is high because threads are still running but blocking each other. In not-so-serious cases, the livelock will eventually resolve because of the random scheduling, but there will be a long delay in the task that got livelocked, with a high CPU usage during that delay.
- *Blocking on I/O or other external input*—If your thread is blocked waiting for external input, it can't proceed, even if the waited-for input is never going to come. It's therefore undesirable to block on external input from a thread that also performs tasks that other threads may be waiting for.

That briefly covers unwanted blocking. What about race conditions?

11.1.2 Race conditions

Race conditions are the most common cause of problems in multithreaded code—many deadlocks and livelocks only actually manifest because of a race condition. Not all race conditions are problematic—a race condition occurs anytime the behavior depends on the

relative scheduling of operations in separate threads. A large number of race conditions are entirely benign; for example, which worker thread processes the next task in the task queue is largely irrelevant. However, many concurrency bugs are due to race conditions. In particular, race conditions often cause the following types of problems:

- *Data races*—A data race is the specific type of race condition that results in undefined behavior because of unsynchronized concurrent access to a shared memory location. I introduced data races in chapter 5 when we looked at the C++ memory model. Data races usually occur through incorrect usage of atomic operations to synchronize threads or through access to shared data without locking the appropriate mutex.
- *Broken invariants*—These can manifest as dangling pointers (because another thread deleted the data being accessed), random memory corruption (due to a thread reading inconsistent values resulting from partial updates), and double-free (such as when two threads pop the same value from a queue, and so both delete some associated data), among others. The invariants being broken can be temporal- as well as value-based. If operations on separate threads are required to execute in a particular order, incorrect synchronization can lead to a race condition in which the required order is sometimes violated.
- *Lifetime issues*—Although you could bundle these problems in with broken invariants, this really is a separate category. The basic problem with bugs in this category is that the thread outlives the data that it accesses, so it is accessing data that has been deleted or otherwise destroyed, and potentially the storage is even reused for another object. You typically get lifetime issues where a thread references local variables that go out of scope before the thread function has completed, but they aren't limited to that scenario. Whenever the lifetime of the thread and the data it operates on aren't tied together in some way, there's the potential for the data to be destroyed before the thread has finished and for the thread function to have the rug pulled out from under its feet. If you manually call `join()` in order to wait for the thread to complete, you need to ensure that the call to `join()` can't be skipped if an exception is thrown. This is basic exception safety applied to threads.

It's the problematic race conditions that are the killers. With deadlock and livelock, the application appears to hang and become completely unresponsive or takes too long to complete a task. Often, you can attach a debugger to the running process to identify which threads are involved in the deadlock or livelock and which synchronization objects they're fighting over. With data races, broken invariants, and lifetime issues, the visible symptoms of the problem (such as random crashes or incorrect output) can manifest anywhere in the code—the code may overwrite memory used by another part of the system that isn't touched until much later. The fault will then manifest in code completely unrelated to the location of the buggy code, possibly much later in the execution of the program. This is the true curse of shared memory systems—however much you try to limit which data is accessible by which thread and try to ensure that correct synchronization is used, any thread can overwrite the data being used by any other thread in the application.

Now that we've briefly identified the sorts of problems we're looking for, let's look at what you can do to locate any instances in your code so you can fix them.

11.2 Techniques for locating concurrency-related bugs

In the previous section we looked at the types of concurrency-related bugs you might see and how they might manifest in your code. With that information in mind, you can then look at your code to see where bugs might lie and how you can attempt to determine whether there are any bugs in a particular section.

Perhaps the most obvious and straightforward thing to do is *look at the code*. Although this might seem obvious, it's actually difficult to do in a thorough way. When you read code you've just written, it's all too easy to read what you intended to write rather than what's actually there. Likewise, when reviewing code that others have written, it's tempting to just give it a quick read-through, check it off against your local coding standards, and highlight any glaringly obvious problems. What's needed is to spend the time really going through the code with a fine-tooth comb, thinking about the concurrency issues—and the non-concurrency issues as well. (You might as well, while you're doing it. After all, a bug is a bug.) We'll cover specific things to think about when reviewing code shortly.

Even after thoroughly reviewing your code, you still might have missed some bugs, and in any case you need to confirm that it does indeed work, for peace of mind if nothing else. Consequently, we'll continue on from reviewing the code to a few techniques to employ when testing multithreaded code.

11.2.1 Reviewing code to locate potential bugs

As I've already mentioned, when reviewing multithreaded code to check for con-currency-related bugs, it's important to review it thoroughly, with a fine-tooth comb. If possible, get someone else to review it. Because they haven't written the code, they'll have to think through how it works, and this will help uncover any bugs that may be there. It's important that the reviewer have the time to do the review properly—not a casual two-minute quick glance, but a proper, considered review. Most concurrency bugs require more than a quick glance to spot—they usually rely on subtle timing issues to actually manifest.

If you get one of your colleagues to review the code, they'll be coming at it fresh. They'll therefore see things from a different point of view and may well spot things that you don't. If you don't have colleagues you can ask, ask a friend, or even post the code on the internet (taking care not to upset your company lawyers). If you can't get anybody to review your code for you, or they don't find anything, don't worry—there's still more you can do. For starters, it might be worth leaving the code alone for a while—work on another part of the application, read a book, or go for a walk. If you take a break, your subconscious can work on the problem in the background while you're consciously focused on something else. Also, the code will be less familiar when you come back to it—you might manage to look at it from a different perspective yourself.

An alternative to getting someone else to review your code is to do it yourself. One useful technique is to try to explain how it works *in detail* to someone else. They don't even have to be physically there—many teams have a bear or rubber chicken for this purpose, and I personally find that writing detailed notes can be hugely beneficial. As you explain, think about each line, what could happen, which data it accesses, and so forth. Ask yourself questions about the code, and explain the answers. I find this to be an incredibly powerful technique—by asking myself these questions and thinking carefully about the answers, the problem often reveals itself. These questions can be helpful for *any* code review, not just when reviewing your own code.

QUESTIONS TO THINK ABOUT WHEN REVIEWING MULTITHREADED CODE

As I've already mentioned, it can be useful for a reviewer (whether the code's author or someone else) to think about specific questions relating to the code being reviewed. These questions can focus the reviewer's mind on the relevant details of the code and can help identify potential problems. The questions I like to ask include the following, though this is most definitely not a comprehensive list. You might find other questions that help you to focus better. Anyway, here are the questions:

- Which data needs to be protected from concurrent access?
- How do you ensure that the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this thread hold?
- Which mutexes might other threads hold?
- Are there any ordering requirements between the operations done in this thread and those done in another? How are those requirements enforced?
- Is the data loaded by this thread still valid? Could it have been modified by other threads?
- If you assume that another thread could be modifying the data, what would that mean and how could you ensure that this never happens?

This last question is my favorite, because it really makes me think about the relationships between the threads. By assuming the existence of a bug related to a particular line of code, you can then act as a detective and track down the cause. In order to convince yourself that there's no bug, you have to consider every corner case and possible ordering. This is particularly useful where the data is protected by more than one mutex over its lifetime, such as with the thread-safe queue from chapter 6 where we had separate mutexes for the head and tail of the queue: in order to be sure that an access is safe while holding one mutex, you have to be certain that a thread holding the *other* mutex can't also access the same element. It also makes it obvious that public data, or data for which other code can readily obtain a pointer or reference, has to come under particular scrutiny.

The penultimate question in the list is also important, because it addresses what's an easy mistake to make: if you release and then reacquire a mutex, you must assume that other threads may have modified the shared data. Although this is obvious, if the mutex locks aren't

immediately visible—perhaps because they're internal to an object—you may unwittingly be doing exactly that. In chapter 6 you saw how this can lead to race conditions and bugs where the functions provided on a thread-safe data structure are too fine-grained. Whereas for a non-thread-safe stack it makes sense to have separate `top()` and `pop()` operations, for a stack that may be accessed by multiple threads concurrently, this is no longer the case because the lock on the internal mutex is released between the two calls, and so another thread can modify the stack. As you saw in chapter 6, the solution is to combine the two operations so they are both performed under the protection of the same mutex lock, thus eliminating the potential race condition.

OK, so you've reviewed your code (or got someone else to review it). You're sure there are no bugs. The proof of the pudding is, as they say, in the eating—how can you test your code to confirm or deny your belief in its lack of bugs?

11.2.2 Locating concurrency-related bugs by testing

When developing single-threaded applications, testing your applications is relatively straightforward, if time consuming. You could, in principle, identify all the possible sets of input data (or at least all the interesting cases) and run them through the application. If the application produced the correct behavior and output, you'd know it works for that given set of input. Testing for error states such as the handling of disk-full errors is more complicated than that, but the idea is the same—set up the initial conditions and allow the application to run.

Testing multithreaded code is an order of magnitude harder, because the precise scheduling of the threads is indeterminate and may vary from run to run. Consequently, even if you run the application with the same input data, it might work correctly some times and fail at other times if there's a race condition lurking in the code. Just because there's a potential race condition doesn't mean the code will fail *always*, just that it *might* fail *sometimes*.

Given the inherent difficulty of reproducing concurrency-related bugs, it pays to design your tests carefully. You want each test to run the smallest amount of code that could potentially demonstrate a problem, so that you can best isolate the code that's faulty if the test fails—it's better to test a concurrent queue directly to verify that concurrent pushes and pops work rather than testing it through a whole chunk of code that uses the queue. It can help if you think about how code should be tested when designing it—see the section on designing for testability later in this chapter.

It's also worth eliminating the concurrency from the test in order to verify that the problem is concurrency-related. If you have a problem when everything is running in a single thread, it's just a plain common or garden-variety bug rather than a concurrency-related bug. This is particularly important when trying to track down a bug that occurs "in the wild" as opposed to being detected in your test harness. Just because a bug occurs in the multithreaded portion of your application doesn't mean it's automatically concurrency-related. If you're using thread pools to manage the level of concurrency, there's usually a configuration parameter you can set to specify the number of worker threads. If you're managing threads manually, you'll have to modify the code to use a single thread for the test. Either way, if you can reduce your

application to a single thread, you can eliminate concurrency as a cause. On the flip side, if the problem goes away on a *single-core* system (even with multiple threads running) but is present on *multicore* systems or *multiprocessor* systems, you have a race condition and possibly a synchronization or memory-ordering issue.

There's more to testing concurrent code than the structure of the code being tested; the structure of the test is just as important, as is the test environment. If you continue on with the example of testing a concurrent queue, you have to think about various scenarios:

- One thread calling `push()` or `pop()` on its own to verify that the queue does work at a basic level
- One thread calling `push()` on an empty queue while another thread calls `pop()`
- Multiple threads calling `push()` on an empty queue
- Multiple threads calling `push()` on a full queue
- Multiple threads calling `pop()` on an empty queue
- Multiple threads calling `pop()` on a full queue
- Multiple threads calling `pop()` on a partially full queue with insufficient items for all threads
- Multiple threads calling `push()` while one thread calls `pop()` on an empty queue
- Multiple threads calling `push()` while one thread calls `pop()` on a full queue
- Multiple threads calling `push()` while multiple threads call `pop()` on an empty queue
- Multiple threads calling `push()` while multiple threads call `pop()` on a full queue

Having thought about all these scenarios and more, you then need to consider additional factors about the test environment:

- What you mean by "multiple threads" in each case (3, 4, 1024?)
- Whether there are enough processing cores in the system for each thread to run on its own core
- Which processor architectures the tests should be run on
- How you ensure suitable scheduling for the "while" parts of your tests

There are additional factors to think about specific to your particular situation. Of these four environmental considerations, the first and last affect the structure of the test itself (and are covered in section 11.2.5), whereas the other two are related to the physical test system being used. The number of threads to use relates to the particular code being tested, but there are various ways of structuring tests to obtain suitable scheduling. Before we look at these techniques, let's look at how you can design your application code to be easier to test.

11.2.3 Designing for testability

Testing multithreaded code is difficult, so you want to do what you can to make it easier. One of the most important things you can do is *design* the code for testability. A lot has been written about designing single-threaded code for testability, and much of the advice still applies. In general, code is easier to test if the following factors apply:

- The responsibilities of each function and class are clear.

- The functions are short and to the point.
- Your tests can take complete control of the environment surrounding the code being tested.
- The code that performs the particular operation being tested is close together rather than spread throughout the system.
- You thought about how to test the code before you wrote it.

All of these are still true for multithreaded code. In fact, I'd argue that it's even more important to pay attention to the testability of multithreaded code than for single-threaded code, because it's inherently that much harder to test. That last point is important: even if you don't go as far as writing your tests before the code, it's well worth thinking about how you can test the code before you write it—what inputs to use, which conditions are likely to be problematic, how to stimulate the code in potentially problematic ways, and so on.

One of the best ways to design concurrent code for testing is to eliminate the concurrency. If you can break down the code into those parts that are responsible for the communication paths between threads and those parts that operate on the communicated data within a single thread, then you've greatly reduced the problem. Those parts of the application that operate on data that's being accessed by only that one thread can then be tested using the normal single-threaded techniques. The hard-to-test concurrent code that deals with communicating between threads and ensuring that only one thread at a time *is* accessing a particular block of data is now much smaller and the testing more tractable.

For example, if your application is designed as a multithreaded state machine, you could split it into several parts. The state logic for each thread, which ensures that the transitions and operations are correct for each possible set of input events, can be tested independently with single-threaded techniques, with the test harness providing the input events that would be coming from other threads. Then, the core state machine and message routing code that ensures that events are correctly delivered to the right thread in the right order can be tested independently, but with multiple concurrent threads and simple state logic designed specifically for the tests.

Alternatively, if you can divide your code into multiple blocks of *read shared data/transform data/update shared data*, you can test the *transform data* portions using all the usual single-threaded techniques, because this is now just single-threaded code. The hard problem of testing a multithreaded transformation will be reduced to testing the reading and updating of the shared data, which is much simpler.

One thing to watch out for is that library calls can use internal variables to store state, which then becomes shared if multiple threads use the same set of library calls. This can be a problem because it's not immediately apparent that the code accesses shared data. However, with time you learn which library calls these are, and they stick out like sore thumbs. You can then either add appropriate protection and synchronization or use an alternate function that's safe for concurrent access from multiple threads.

There's more to designing multithreaded code for testability than structuring your code to minimize the amount of code that needs to deal with concurrency-related issues and paying

attention to the use of non-thread-safe library calls. It's also helpful to bear in mind the same set of questions you ask yourself when reviewing the code, from section 11.2.1. Although these questions aren't directly about testing and testability, if you think about the issues with your "testing hat" on and consider how to test the code, it will affect which design choices you make and will make testing easier.

Now that we've looked at designing code to make testing easier, and potentially modified the code to separate the "concurrent" parts (such as the thread-safe containers or state machine event logic) from the "single-threaded" parts (which may still interact with other threads through the concurrent chunks), let's look at the techniques for testing concurrency-aware code.

11.2.4 Multithreaded testing techniques

So, you've thought through the scenario you wish to test and written a small amount of code that exercises the functions being tested. How do you ensure that any potentially problematic scheduling sequences are exercised in order to flush out the bugs?

Well, there are a few ways of approaching this, starting with brute-force testing, or stress testing.

BRUTE-FORCE TESTING

The idea behind brute-force testing is to stress the code to see if it breaks. This typically means running the code many times, possibly with many threads running at once. If there's a bug that manifests only when the threads are scheduled in a particular fashion, then the more times the code is run, the more likely the bug is to appear. If you run the test once and it passes, you might feel a bit of confidence that the code works. If you run it ten times in a row and it passes every time, you'll likely feel more confident. If you run the test a billion times and it passes every time, you'll feel more confident still.

The confidence you have in the results does depend on the amount of code being tested by each test. If your tests are quite fine-grained, like the tests outlined previously for a thread-safe queue, such brute-force testing can give you a high degree of confidence in your code. On the other hand, if the code being tested is considerably larger, the number of possible scheduling permutations is so vast that even a billion test runs might yield a low level of confidence.

The downside to brute-force testing is that it might give you false confidence. If the way you've written the test means that the problematic circumstances can't occur, you can run the test as many times as you like and it won't fail, even if it would fail every time in slightly different circumstances. The worst example is where the problematic circumstances can't occur on your test system because of the way the particular system you're testing on happens to run. Unless your code is to run only on systems identical to the one being tested, the particular hardware and operating system combination may not allow the circumstances that would cause a problem to arise.

The classic example here is testing a multithreaded application on a single-processor system. Because every thread has to run on the same processor, everything is automatically serialized, and many race conditions and cache ping-pong problems that you may get with a true multiprocessor system evaporate. This isn't the only variable though; different processor architectures provide different synchronization and ordering facilities. For example, on x86 and x86-64 architectures, atomic load operations are always the same, whether tagged `memory_order_relaxed` or `memory_order_seq_cst` (see section 5.3.3). This means that code written using relaxed memory ordering may work on systems with an x86 architecture, where it would fail on a system with a finer-grained set of memory-ordering instructions such as SPARC.

If you need your application to be portable across a range of target systems, it's important to test it on representative instances of those systems. This is why I listed the processor architectures being used for testing as a consideration in section 11.2.2.

Avoiding the potential for false confidence is crucial to successful brute-force testing. This requires careful thought over test design, not just with respect to the choice of unit for the code being tested but also with respect to the design of the test harness and the choice of testing environment. You need to ensure that as many of the code paths as possible are tested and as many of the possible thread interactions as feasible. Not only that, but you need to know *which* options are covered and *which are left untested*.

Although brute-force testing does give you some degree of confidence in your code, it's not guaranteed to find all the problems. There's one technique that *is* guaranteed to find the problems, if you have the time to apply it to your code and the appropriate software. I call it *combination simulation testing*.

COMBINATION SIMULATION TESTING

That's a bit of a mouthful, so I'd best explain what I mean. The idea is that you run your code with a special piece of software that *simulates* the real runtime environment of the code. You may be aware of software that allows you to run multiple virtual machines on a single physical computer, where the characteristics of the virtual machine and its hardware are emulated by the supervisor software. The idea here is similar, except rather than just emulating the system, the simulation software records the sequences of data accesses, locks, and atomic operations from each thread. It then uses the rules of the C++ memory model to repeat the run with every permitted *combination* of operations and thus identify race conditions and deadlocks.

Although such exhaustive combination testing is guaranteed to find all the problems the system is designed to detect, for anything but the most trivial of programs it will take a huge amount of time, because the number of combinations increases exponentially with the number of threads and the number of operations performed by each thread. This technique is thus best reserved for fine-grained tests of individual pieces of code rather than an entire application. The other obvious downside is that it relies on the availability of simulation software that can handle the operations used in your code.

So, you have a technique that involves running your test many times under normal conditions but that might miss problems, and you have a technique that involves running your test many times under special conditions but that's more likely to find any problems that exist. Are there any other options?

A third option is to use a library that detects problems as they occur in the running of the tests.

DETECTING PROBLEMS EXPOSED BY TESTS WITH A SPECIAL LIBRARY

Although this option doesn't provide the exhaustive checking of a combination simulation test, you can identify many problems by using a special implementation of the library synchronization primitives such as mutexes, locks, and condition variables. For example, it's common to require that all accesses to a piece of shared data be done with a particular mutex locked. If you could check which mutexes were locked when the data was accessed, you could verify that the appropriate mutex was indeed locked by the calling thread when the data was accessed and report a failure if this was not the case. By marking your shared data in some way, you can allow the library to check this for you.

Such a library implementation can also record the sequence of locks if more than one mutex is held by a particular thread at once. If another thread locks the same mutexes in a different order, this could be recorded as a *potential* deadlock even if the test didn't actually deadlock while running.

Another type of special library that could be used when testing multithreaded code is one where the implementations of the threading primitives such as mutexes and condition variables give the test writer control over which thread gets the lock when multiple threads are waiting or which thread is notified by a `notify_one()` call on a condition variable. This would allow you to set up particular scenarios and verify that your code works as expected in those scenarios.

Some of these testing facilities would have to be supplied as part of the C++ Standard Library implementation, whereas others can be built on top of the Standard Library as part of your test harness.

Having looked at various ways of executing test code, let's now look at ways of structuring the code to achieve the scheduling you want.

11.2.5 Structuring multithreaded test code

Back in section 11.2.2 I said that you need to find ways of providing suitable scheduling for the "while" part of your tests. Now it's time to look at the issues involved in that.

The basic issue is that you need to arrange for a set of threads to each be executing a chosen piece of code at a time that you specify. In the most basic case you have two threads, but this could easily be extended to more. In the first step, you need to identify the distinct parts of each test:

- The general setup code that must be executed before anything else
- The thread-specific setup code that must run on each thread

- The actual code for each thread that you desire to run concurrently
- The code to be run after the concurrent execution has finished, possibly including assertions on the state of the code

To explain further, let's consider a specific example from the test list in section 11.2.2: one thread calling `push()` on an empty queue while another thread calls `pop()`.

The *general* setup code is simple: you must create the queue. The thread executing `pop()` has no *thread-specific* setup code. The thread-specific setup code for the thread executing `push()` depends on the interface to the queue and the type of object being stored. If the object being stored is expensive to construct or must be heap allocated, you want to do this as part of the thread-specific setup, so that it doesn't affect the test. On the other hand, if the queue is just storing plain `ints`, there's nothing to be gained by constructing an `int` in the setup code. The actual code being tested is relatively straightforward—a call to `push()` from one thread and a call to `pop()` from another—but what about the “after completion” code?

In this case, it depends on what you want `pop()` to do. If it's supposed to block until there is data, then clearly you want to see that the returned data is what was supplied to the `push()` call and that the queue is empty afterward. If `pop()` is *not* blocking and may complete even when the queue is empty, you need to test for two possibilities: either the `pop()` returned the data item supplied to the `push()` and the queue is empty or the `pop()` signaled that there was no data and the queue has one element. One or the other must be true; what you want to avoid is the scenario that `pop()` signaled “no data” but the queue is empty, or that `pop()` returned the value and the queue is *still* not empty. In order to simplify the test, assume you have a blocking `pop()`. The final code is therefore an assertion that the popped value is the pushed value and that the queue is empty.

Now, having identified the various chunks of code, you need to do the best you can to ensure that everything runs as planned. One way to do this is to use a set of `std::promise`s to indicate when everything is ready. Each thread sets a promise to indicate that it's ready and then waits on a (copy of a) `std::shared_future` obtained from a third `std::promise`; the main thread waits for all the promises from all the threads to be set and then triggers the threads to go. This ensures that each thread has started and is just before the chunk of code that should be run concurrently; any thread-specific setup should be done before setting that thread's promise. Finally, the main thread waits for the threads to complete and checks the final state. You also need to be aware of exceptions and make sure you don't have any threads left waiting for the go signal when that's not going to happen. The following listing shows one way of structuring this test.

Listing 11.1 An example test for concurrent push() and pop() calls on a queue

```
void test_concurrent_push_and_pop_on_empty_queue()
{
    threadsafe_queue<int> q;                                #1
    std::promise<void> go, push_ready, pop_ready;           #2
    std::shared_future<void> ready(go.get_future());          #3
    std::future<void> push_done;                            #4
    std::future<int> pop_done;
```

```

try
{
    push_done=std::async(std::launch::async,           #5
                         [&q,ready,&push_ready]())
    {
        push_ready.set_value();
        ready.wait();
        q.push(42);
    }
);
pop_done=std::async(std::launch::async,           #6
                    [&q,ready,&pop_ready]())
{
    pop_ready.set_value();
    ready.wait();
    return q.pop();          #7
}
);
push_ready.get_future().wait();
pop_ready.get_future().wait();
go.set_value();                                #9
push_done.get();                               #10
assert(pop_done.get()==42);                   #11
assert(q.empty());
}
catch(...)
{
    go.set_value();                            #12
    throw;
}
}

```

The structure is pretty much as described previously. First, you create your empty queue as part of the general setup #1. Then, you create all your promises for the “ready” signals #2 and get a `std::shared_future` for the go signal #3. Then, you create the futures you’ll use to indicate that the threads have finished #4. These have to go outside the `try` block so that you can set the go signal on an exception without waiting for the test threads to complete (which would deadlock—a deadlock in the test code would be rather less than ideal).

Inside the `try` block you can then start the threads #5, #6—you use `std::launch::async` to guarantee that the tasks are each running on their own thread. Note that the use of `std::async` makes your exception-safety task easier than it would be with plain `std::thread` because the destructor for the future will join with the thread. The lambda captures specify that each task will reference the queue and the relevant promise for signaling readiness, while taking a copy of the `ready` future you got from the `go` promise.

As described previously, each task sets its own `ready` signal and then waits for the general `ready` signal before running the actual test code. The main thread does the reverse—waiting for the signals from both threads #8 before signaling them to start the real test #9.

Finally, the main thread calls `get()` on the futures from the `async` calls to wait for the tasks to finish #10, #11 and checks the results. Note that the `pop` task returns the retrieved value through the future #7, so you can use that to get the result for the assert #11.

If an exception is thrown, you set the go signal to avoid any chance of a dangling thread and rethrow the exception #12. The futures corresponding to the tasks #4 were declared last, so they'll be destroyed first, and their destructors will wait for the tasks to complete if they haven't already.

Although this seems like quite a lot of boilerplate just to test two simple calls, it's necessary to use something similar in order to have the best chance of testing what you actually want to test. For example, actually starting a thread can be quite a time-consuming process, so if you didn't have the threads wait for the go signal, then the push thread may have completed before the pop thread even started, which would completely defeat the point of the test. Using the futures in this way ensures that both threads are running and blocked on the same future. Unblocking the future then allows both threads to run. Once you're familiar with the structure, it should be relatively straightforward to create new tests in the same pattern. For tests that require more than two threads, this pattern is readily extended to additional threads.

So far, we've just been looking at the *correctness* of multithreaded code. Although this is the most important issue, it's not the only reason you test: it's also important to test the *performance* of multithreaded code, so let's look at that next.

11.2.6 Testing the performance of multithreaded code

One of the main reasons you might choose to use concurrency in an application is to make use of the increasing prevalence of multicore processors to improve the performance of your applications. It's therefore important to actually test your code to confirm that the performance does indeed improve, just as you'd do with any other attempt at optimization.

The particular issue with using concurrency for performance is the *scalability*—you want code that runs approximately 24 times faster or processes 24 times as much data on a 24-core machine than on a single-core machine, all else being equal. You don't want code that runs twice as fast on a dual-core machine but is actually slower on a 24-core machine. As you saw in section 8.4.2, if a significant section of your code runs on only one thread, this can limit the potential performance gain. It's therefore worth looking at the overall design of the code before you start testing, so you know whether you're hoping for a factor-of-24 improvement, or whether the serial portion of your code means you're limited to a maximum of a factor of 3.

As you've already seen in previous chapters, contention between processors for access to a data structure can have a big performance impact. Something that scales nicely with the number of processors when that number is small may actually perform badly when the number of processors is much larger because of the huge increase in contention.

Consequently, when testing for the performance of multithreaded code, it's best to check the performance on systems with as many different configurations as possible, so you get a picture of the scalability graph. At the very least, you ought to test on a single-processor system *and* a system with as many processing cores as are available to you.

11.3 Summary

In this chapter we looked at various types of concurrency-related bugs that you might encounter, from deadlocks and livelocks to data races and other problematic race conditions. We followed that with techniques for locating bugs. These included issues to think about during code reviews, guidelines for writing testable code, and how to structure tests for concurrent code. Finally, we looked at some utility components that can help with testing.