11
Concurrency

After covering lazy evaluation and proxy objects in the last chapter, we will now explore how to write concurrent programs in C++ using threads with shared memory. We will look at ways to make concurrent programs correct by writing programs that are free from data races and deadlocks. This chapter will also contain advice on how to make concurrent programs run with low latency and high throughput.

Before we go any further, you should know that this chapter is not a complete introduction to concurrent programming, nor will it cover all the details of concurrency in C++. Instead, this chapter is an introduction to the core building blocks of writing concurrent programs in C++, mixed with some performance-related guidelines. If you haven't written concurrent programs before, it is wise to go through some introductory material to cover the theoretical aspects of concurrent programming. Concepts such as deadlocks, critical sections, condition variables, and mutexes will be very briefly discussed, but this will serve more as a refresher than a thorough introduction to the concepts.

The chapter covers the following:

- The fundamentals of concurrent programming, including parallel execution, shared memory, data races, and deadlocks
- An introduction to the C++ thread support library, the atomic library, and the C++ memory model
- A short example of lock-free programming

- Performance guidelines

Understanding the basics of concurrency

A concurrent program can execute multiple tasks at the same time. Concurrent programming is, in general, a lot harder than sequential programming, but there are several reasons why a program may benefit from being concurrent:

- **Efficiency**: The smartphones and desktop computers of today have multiple CPU cores that can execute multiple tasks in parallel. If you manage to split a big task into subtasks that can be run in parallel, it is theoretically possible to divide the running time of the big task by the number of CPU cores. For programs that run on machines with one single core, there can still be a gain in performance if a task is I/O bound. While one subtask is waiting for I/O, other subtasks can still perform useful work on the CPU.
- **Responsiveness and low latency contexts**: For applications with a graphical user interface, it is important to never block the UI so that the application becomes unresponsive. To prevent unresponsiveness, it is common to let long-running tasks (like loading a file from disk or fetching some data from the network) execute in separate background threads so that the thread responsible for the UI is never blocked by long-running tasks. Another example where low latency matters is real-time audio. The function responsible for producing buffers of audio data is executed in a separate high-priority thread, while the rest of the program can run in lower-priority threads to handle the UI and so on.
- **Simulation**: Concurrency can make it easier to simulate systems that are concurrent in the real world. After all, most things around us happen concurrently, and sometimes it is very hard to model concurrent flows with a sequential programming model. We will not focus on simulation in this book, but will instead focus on performance-related aspects of concurrency.

Concurrency solves many problems for us, but introduces new ones, as we will discuss next.

What makes concurrent programming hard?

There are a number of reasons why concurrent programming is hard, and, if you have written concurrent programs before, you have most likely already encountered the ones listed here:

- Sharing state between multiple threads in a safe manner is hard. Whenever we have data that can be read and written to at the same time, we need some way of protecting that data from data races. You will see many examples of this later on.
- Concurrent programs are usually more complicated to reason about because of the multiple parallel execution flows.
- Concurrency complicates debugging. Bugs that occur because of data races can be very hard to debug since they are dependent on how threads are scheduled. These kinds of bugs can be hard to reproduce and, in the worst-case scenario, they may even cease to exist when running the program using a debugger. Sometimes an innocent debug trace to the console can change the way a multithreaded program behaves and make the bug temporarily disappear. You have been warned!

Before we start looking at concurrent programming using C++, a few general concepts related to concurrent and parallel programming will be introduced.

Concurrency and parallelism

Concurrency and **parallelism** are two terms that are sometimes used interchangeably. However, they are not the same and it is important to understand the differences between them. A program is said to run

concurrently if it has multiple individual control flows running during overlapping time periods. In C++, each individual control flow is represented by a thread. The threads may or may not execute at the exact same time, though. If they do, they are said to execute in parallel. For a concurrent program to run in parallel, it needs to be executed on a machine that has support for parallel execution of instructions; that is, a machine with multiple CPU cores.

At first glance, it might seem obvious that we always want concurrent programs to run in parallel if possible, for efficiency reasons. However, that is not necessarily always true. A lot of synchronization primitives (such as mutex locks) covered in this chapter are required only to support the parallel execution of threads. Concurrent tasks that are not run in parallel do not require the same locking mechanisms and can be a lot easier to reason about.

Time slicing

You might ask, "How are concurrent threads executed on machines with only a single CPU core?" The answer is **time slicing**. It is the same mechanism that is used by the operating system to support the concurrent execution of processes. In order to understand time slicing, let's assume we have two separate sequences of instructions that should be executed concurrently, as shown in the following figure:

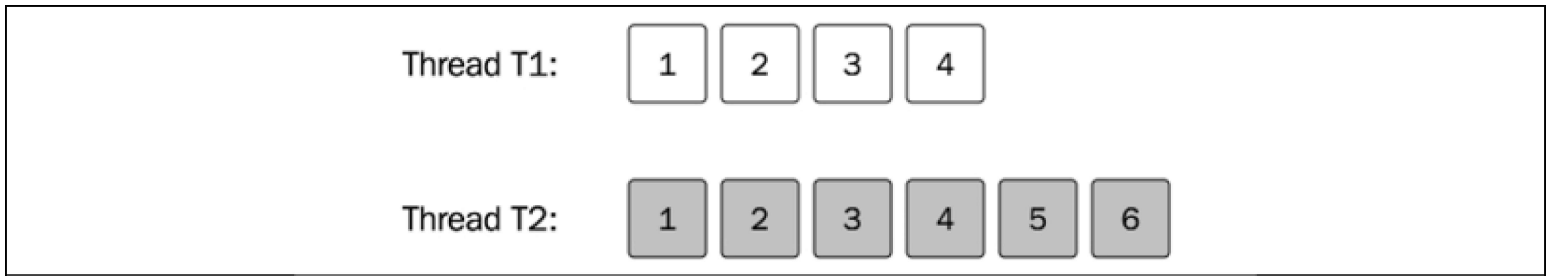

Figure 11.1: Two separate sequences of instructions

The numbered boxes represent the instructions. Each sequence of instructions is executed in a separate thread, labeled **T1** and **T2**. The operating system will schedule each thread to have some limited time on the CPU and then perform a context switch. The context switch will store the current state of the running thread and load the state of the thread that should be executed. This is done often enough so that it appears as if the threads are running at the same time. A context switch is time-consuming, though, and most likely will generate a lot of cache misses each time a new thread gets to execute on a CPU core. Therefore, we don't want context switches to happen too often.

The following figure shows a possible execution sequence of two threads that are being scheduled on a single CPU:

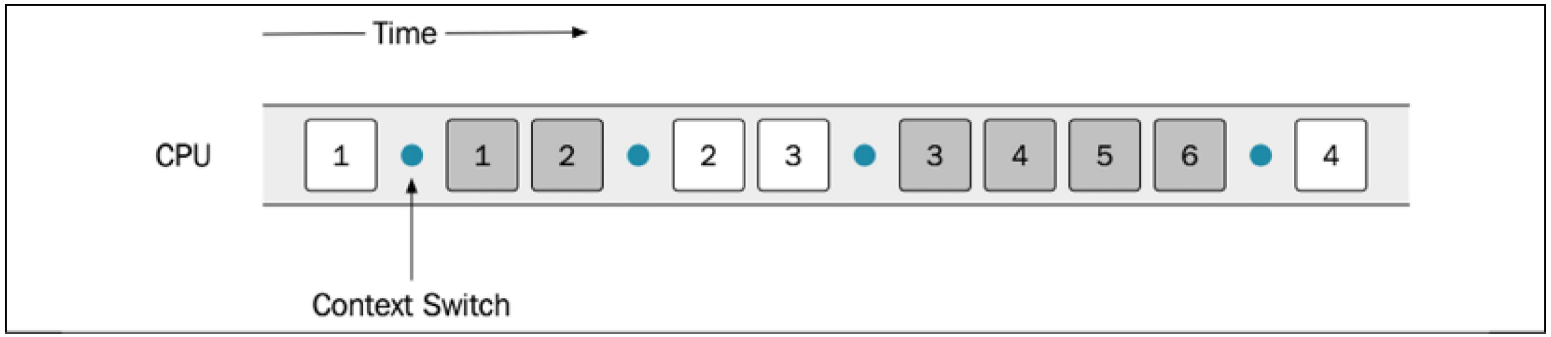

Figure 11.2: A possible execution of two threads. The dots indicate context switches

The first instruction of thread T1 starts, and is then followed by a context switch to let thread T2 execute the first two instructions. As programmers, we must make sure that the program can run as expected, regardless of how the operating system scheduler is scheduling the tasks. If a sequence, for some reason, is invalid, there are ways to control the order in which the instructions get executed by using locks, which will be covered later on.

If a machine has multiple CPU cores, it is possible to execute the two threads in parallel. However, there is no guarantee (it's even unlikely) that the two threads will execute on one core each throughout the lifetime of the program. The entire system shares time on the CPU, so the scheduler will let other processes execute as well. This is one of the reasons why the threads are not scheduled on dedicated cores.

Figure 11.3 shows the execution of the same two threads, but now they are running on a machine with two CPU cores. As you can see, the second and third instructions of the first thread (white boxes) are executing at the exact same time as the other thread is executing — the two threads are executing in parallel:

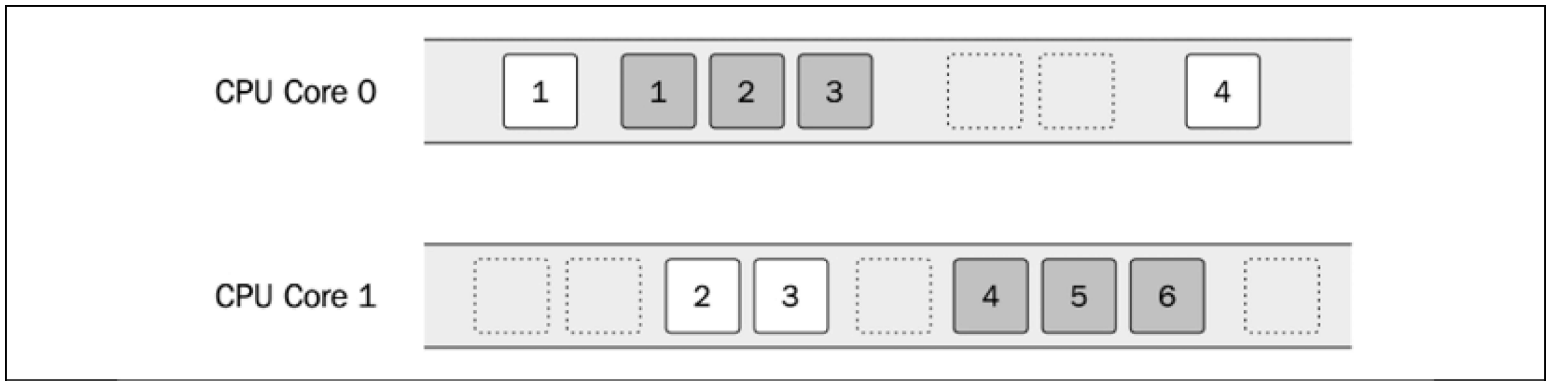

Figure 11.3: Two threads executing on a multicore machine. This makes it possible to execute the two threads in parallel.

Let's discuss shared memory next.

Shared memory

Threads created in the same process share the same virtual memory. This means that a thread can access any data that is addressable within the process. The operating system, which protects memory between

processes using virtual memory, does nothing to protect us from accidentally accessing memory inside a process that was not intended to be shared among different threads. Virtual memory only protects us from accessing memory allocated in different processes to our own.

Sharing memory between multiple threads can be a very efficient way to handle communication between threads. However, sharing memory in a safe way between threads is one of the major challenges when writing concurrent programs in C++. We should always strive to minimize the number of shared resources between threads.

Fortunately, not all memory is shared by default. Each thread has its own stack for storing local variables and other data necessary for handling function calls. Unless a thread passes references or pointers to local variables to other threads, no other thread will be able to access the stack from that thread. This is one more reason to use the stack as much as possible (if you are not already convinced that the stack is a good place for your data after reading *Chapter 7*, *Memory Management*).

There is also **thread local storage**, sometimes abbreviated to **TLS**, which can be used to store variables that are global in the context of a thread but which are not shared between threads. A thread local variable can be thought of as a global variable where each thread has its own copy.

Everything else is shared by default; that is, dynamic memory allocated on the heap, global variables, and static local variables. Whenever you have shared data that is mutated by some thread, you need to ensure that no other thread is accessing that data at the same time or you will have a data race.

Remember the figure from the *Process memory* section of *Chapter 7*, *Memory Management*, which illustrated the virtual address space of a process? Here it is again, but modified to show how it looks when a process contains multiple threads. As you can see in the following figure, each thread has its own stack memory, but there is only one heap for all threads:

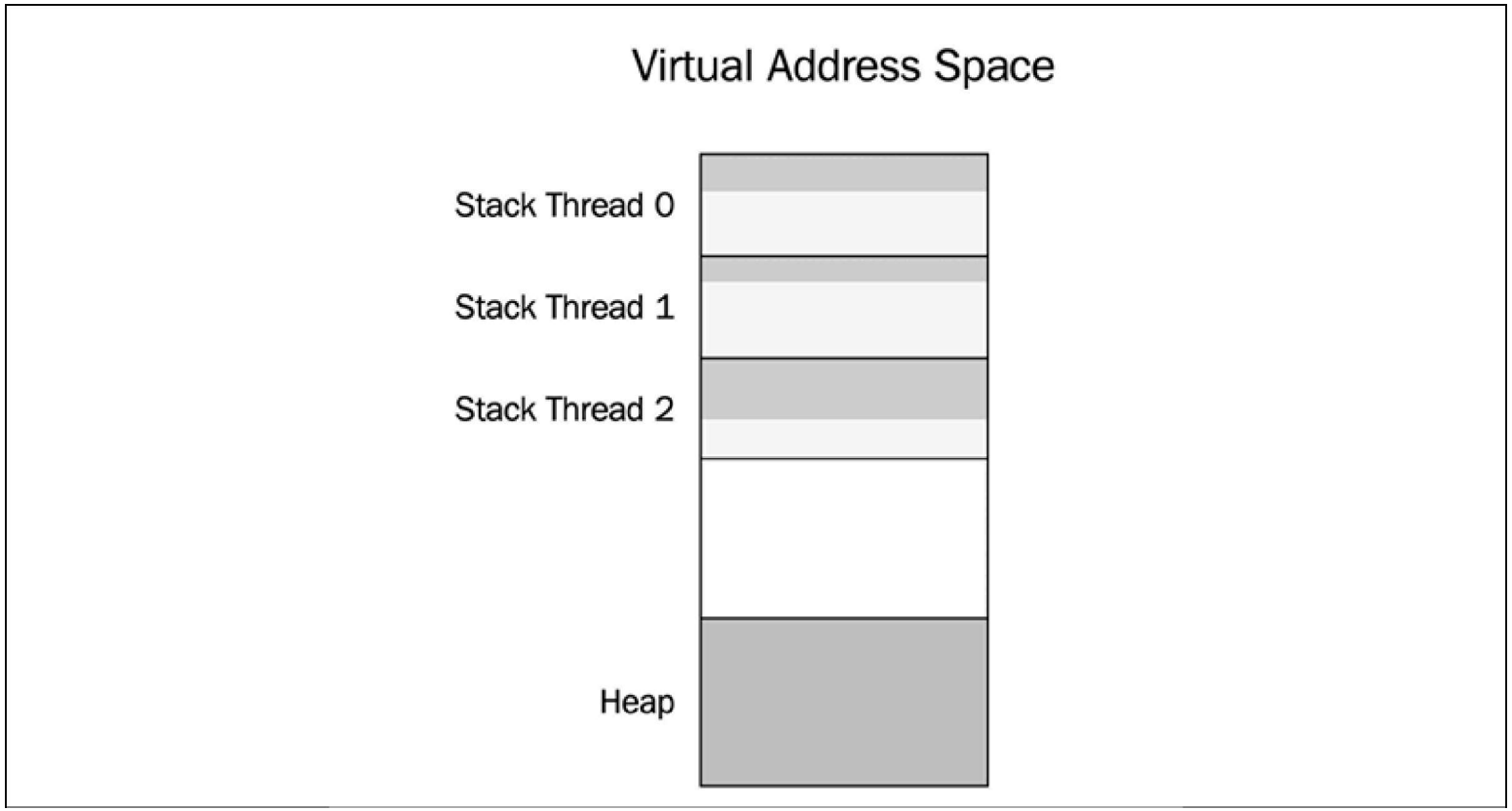

Figure 11.4: A possible layout of the virtual address space for a process

The process contains three threads in this example. The heap memory is by default shared by all threads.

Data races

A **data race** happens when two threads are accessing the same memory at the same time and at least one of the threads is mutating the data. If your program has a data race, it means that your program has un-defined behavior. The compiler and optimizer will *assume* that there are no data races in your code and optimize it under that assumption. This may result in crashes or other completely surprising behavior. In

other words, you can under no circumstances allow data races in your program. The compiler usually doesn't warn you about data races since they are hard to detect at compile time.

Debugging data races can be a real challenge and sometimes requires tools such as **ThreadSanitizer** (from Clang) or **Concurrency Visualizer** (a Visual Studio extension). These tools typically instrument the code so that a runtime library can detect, warn about, or visualize potential data races while running the program you are debugging.

Example: A data race

Figure 11.5 shows two threads that are going to update an integer called `counter`. Imagine that these threads are both incrementing a global counter variable with the instruction `++counter`. It turns out that incrementing an `int` might involve multiple CPU instructions. This can be done in different ways on different CPUs, but let's pretend that `++counter` generates the following made-up machine instructions:

- **R**: Read counter from memory
- **+1**: Increment counter
- **W**: Write new counter value to memory

Now, if we have two threads that are going to update the `counter` value that initially is 42, we would expect it to become 44 after both threads have run. However, as you can see in the following figure, there is no guarantee that the instructions will be executed sequentially to guarantee a correct increment of the `counter` variable.

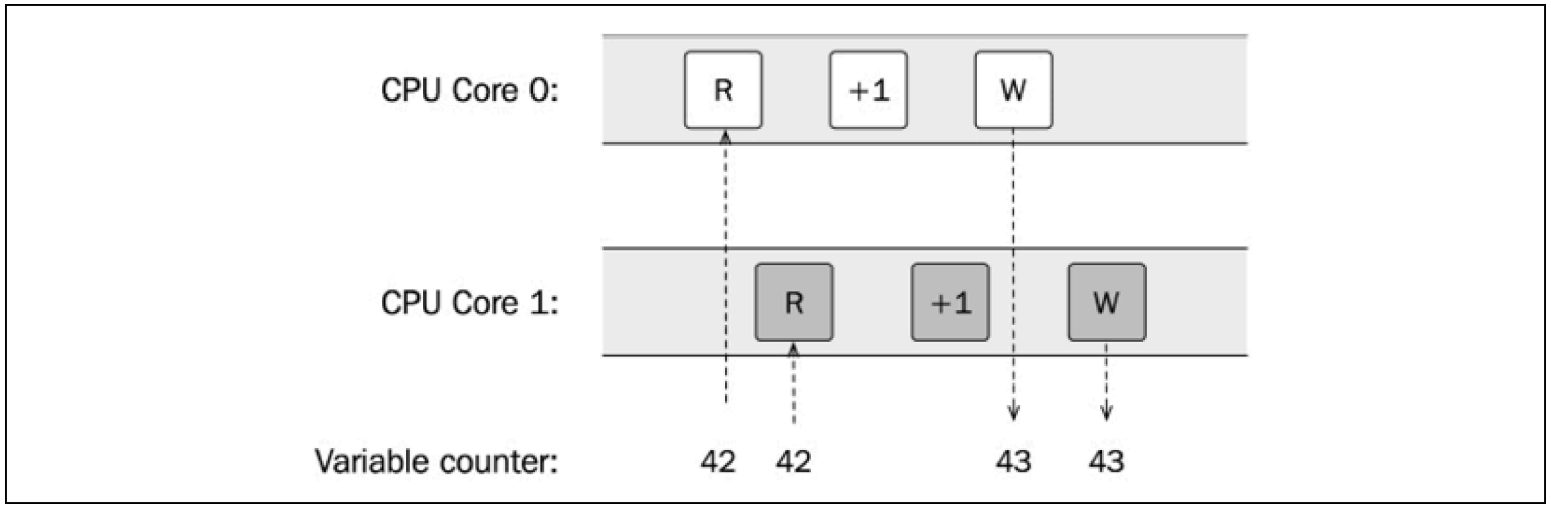

Figure 11.5: The two threads are both incrementing the same shared variable

Without a data race, the counter would have reached the value 44, but instead, it only reaches 43.

In this example, both threads read the value 42 and increment that value to 43. Then, they both write the new value, 43, which means that we never reach the correct answer of 44. Had the first thread been able to write the value 43 before the next thread started to read, we would have ended up with 44 instead. Note also that this would have been possible even if there was only one CPU core. The scheduler could have scheduled the two threads in a similar way so that both read instructions were executed before any writes.

Again, this is one possible scenario, but the important thing is that the behavior is undefined. Anything could happen when your program has a data race. One such example is **tearing**, which is the common term for **torn reads** and **torn writes**. This happens when a thread writes parts of a value to memory while another thread reads the value at the same time and therefore ends up with a corrupt value.

Avoiding data races

How can we avoid data races? There are two main options:

- Use an atomic data type instead of the `int`. This will tell the compiler to execute the read, increment, and write atomically. We will spend more time discussing atomic data types later in this chapter.
- Use a mutually exclusive lock (mutex) that guarantees that multiple threads never execute a critical section at the same time. A **critical section** is a place in the code that must not be executed simultaneously since it updates or reads shared memory that potentially could generate data races.

It is also worth emphasizing that immutable data structures — data structures that are never changed — can be accessed by multiple threads without any risk of data races. Minimizing the use of mutable objects is good for many reasons, but it becomes even more important when writing concurrent programs. A common pattern is to always create new immutable objects instead of mutating existing objects. When the new object is fully constructed and represents the new state, it can be swapped with the old object. In that way, we can minimize the critical sections of our code. Only the swap is a critical section, and hence needs to be protected by an atomic operation or a mutex.

Mutex

A **mutex**, short for **mutual exclusion lock**, is a synchronization primitive for avoiding data races. A thread that needs to enter a critical section first needs to lock the mutex (locking is sometimes also called acquiring a mutex lock). This means that no other thread can lock the same mutex until the first thread that holds the lock has unlocked the mutex. In that way, the mutex guarantees that only one thread at a time is inside a critical section.

In *Figure 11.6*, you can see how the race condition demonstrated in the section *A data race example* can be avoided by using a mutex. The instruction labeled **L** is a lock instruction and the instruction labeled **U** is an unlock instruction. The first thread executing on Core 0 reaches the critical section first and locks the mutex before reading the value of the counter. It then adds 1 to the counter and writes it back to memory. After that, it releases the lock.

The second thread, executing on Core 1, reaches the critical section just after the first thread has acquired the mutex lock. Since the mutex is already locked, the thread is blocked until the first thread has updated the counter undisturbed and released the mutex:

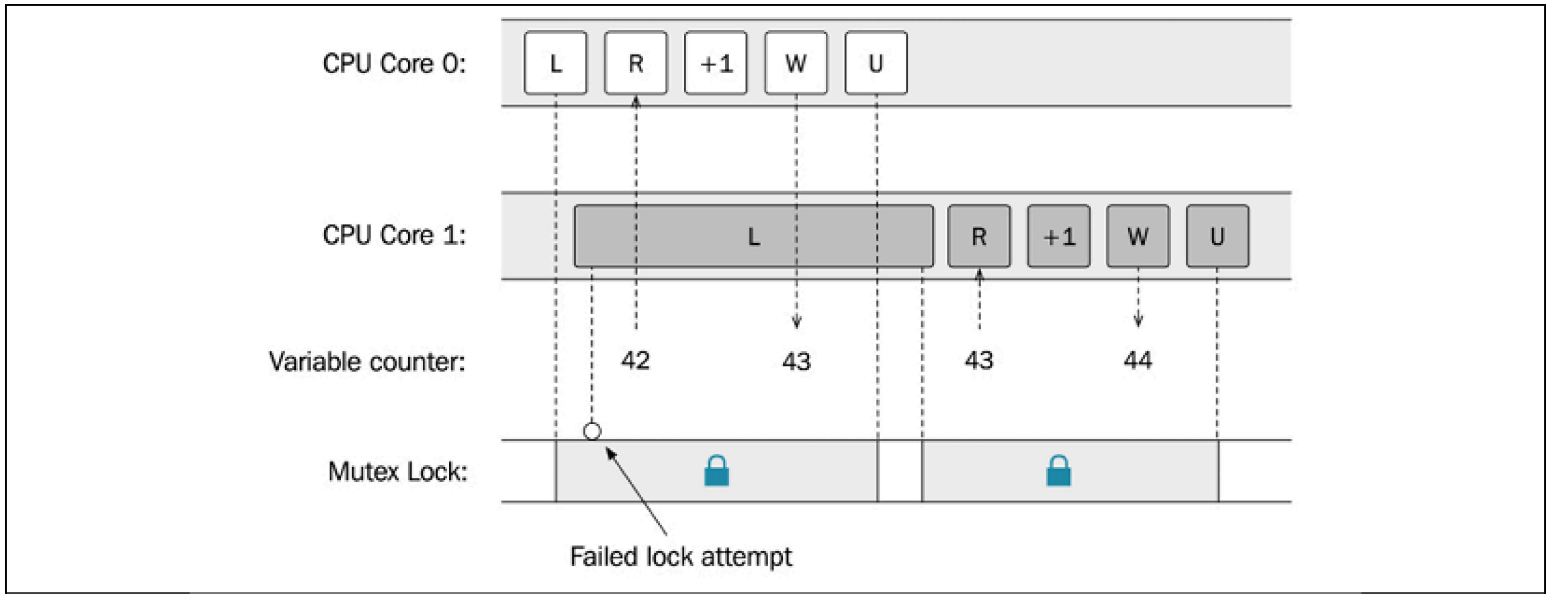

Figure 11.6: The mutex lock is protecting the critical section and avoids data races on the counter variable

The net result is that the two threads can update the mutable shared variable in a safe and correct way. However, it also means that the two threads can no longer be run in parallel. If most of the work a thread

does cannot be done without serializing the work, there is, from a performance perspective, no point in using threads.

The state where the second thread is blocked by the first thread is called **contention**. This is something we strive to minimize, because it hurts the scalability of a concurrent program. Adding more CPU cores will not improve performance if the degree of contention is high.

Deadlock

When using mutex locks to protect shared resources, there is a risk of getting stuck in a state called **deadlock**. A deadlock can happen when two threads are waiting for each other to release their locks. Neither of the threads can proceed and they are stuck in a deadlock state. One condition that needs to be fulfilled for a deadlock to occur is that one thread that already holds a lock tries to acquire an additional lock. When a system grows and gets larger, it becomes more and more difficult to track all locks that might be used by all threads running in a system. This is one reason for always trying to minimize the use of shared resources, and this demonstrates the need for exclusive locking.

Figure 11.7 shows two threads in a waiting state, trying to acquire the lock held by the other thread:

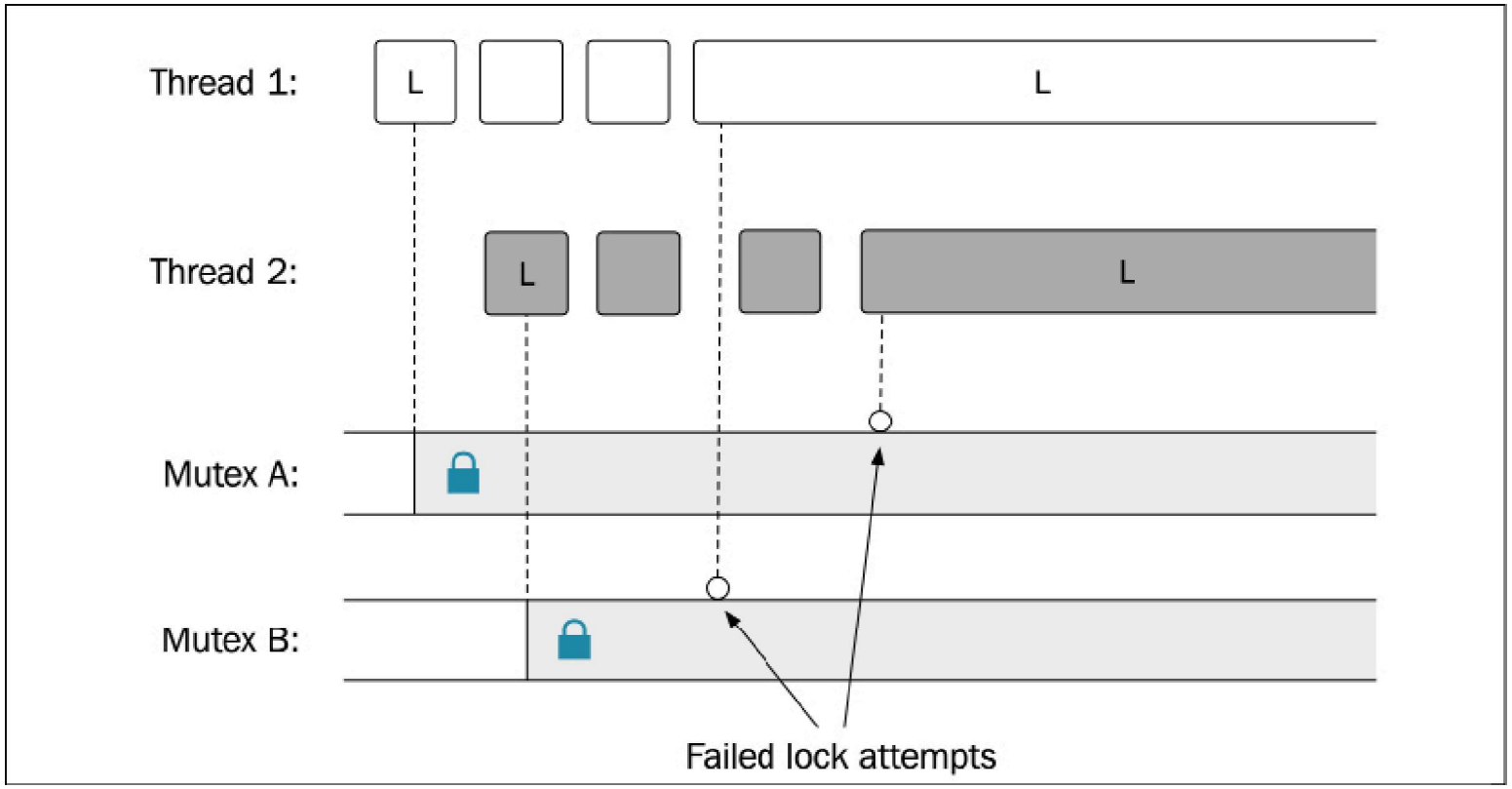

Figure 11.7: An example of a deadlock state

Let's discuss synchronous and asynchronous tasks next.

Synchronous and asynchronous tasks

I will refer to **synchronous tasks** and **asynchronous tasks** in this chapter. Synchronous tasks are like ordinary C++ functions. When a synchronous task is finished doing whatever it is supposed to do, it will return the control to the caller of the task. The caller of the task is waiting or blocked until the synchronous task has finished.

An asynchronous task, on the other hand, will return the control back to the caller immediately and instead perform its work concurrently.

The sequence in *Figure 11.8* shows the difference between calling a synchronous and asynchronous task, respectively:

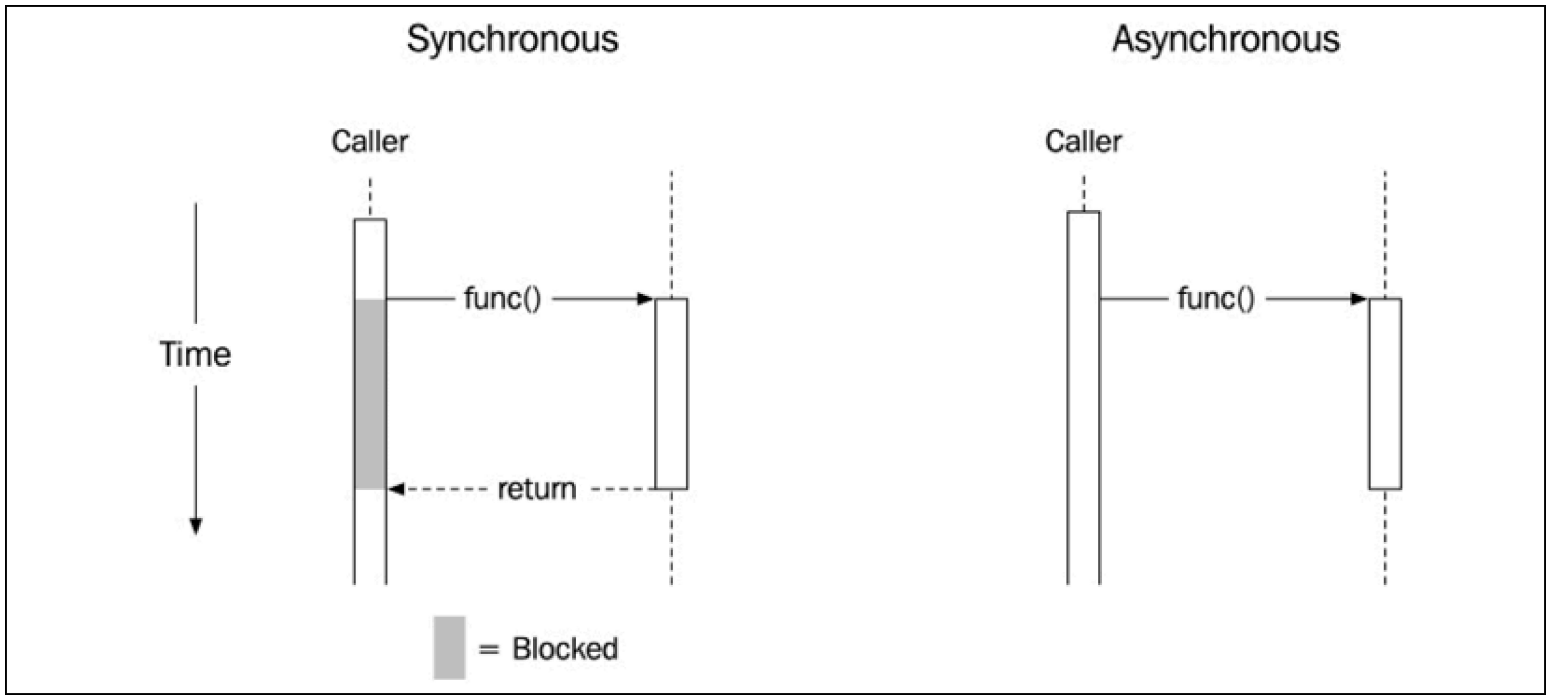

Figure 11.8: Synchronous versus asynchronous calls. The asynchronous task returns immediately but continues to work after the caller has regained control.

If you haven't seen asynchronous tasks before, they might look strange at first, since ordinary functions in C++ always stop executing when they encounter a return statement or reach the end of the function

body. Asynchronous APIs are getting more and more common, though, and it is likely that you have encountered them before, for example, when working with asynchronous JavaScript.

Sometimes, we use the term **blocking** for operations that block the caller; that is, make the caller wait until the operation has finished.

With a general introduction to concurrency behind us, it's time to explore the support for threaded programming in C++.

Concurrent programming in C++

The concurrency support in C++ makes it possible for a program to execute multiple tasks concurrently. As mentioned earlier, writing a correct concurrent C++ program is, in general, a lot harder than writing a program that executes all tasks sequentially in one thread. This section will also demonstrate some common pitfalls to make you aware of all the difficulties involved in writing concurrent programs.

Concurrency support was first introduced in C++11 and has since been extended into C++14, C++17, and C++20. Before concurrency was part of the language, it was implemented with native concurrency support from the operating system, **POSIX Threads** (**pthreads**), or some other library.

With concurrency support directly in the C++ language, we can write cross-platform concurrent programs, which is great! Sometimes, however, you have to reach for platform-specific functionality when dealing with concurrency on your platform. For example, there is no support in the C++ standard library for setting thread priorities, configuring CPU affinity (CPU pinning), or setting the stack size of new threads.

It should also be said that the thread support library has been extended quite a bit with the release of C++20, and more features are likely to be added in future versions of the language. The need for good concurrency support is increasing because of the way hardware is being developed, and there is a lot yet to be discovered when it comes to the efficiency, scalability, and correctness of highly concurrent programs.

The thread support library

We will now take a tour through the C++ thread support library and cover its most important components.

Threads

A running program contains at least one thread. When your main function is called, it is executed on a thread usually referred to as the **main thread**. Each thread has an identifier, which can be useful when debugging a concurrent program. The following program prints the thread identifier of the main thread:

```
int main() {
  std::cout << "Thread ID: " <<  std::this_thread::get_id() << '\n';
}
```

Running the preceding program might produce something like this:

```
 Thread ID: 0x1001553c0
```

It is possible to make a thread sleep. Sleep is rarely used in production code but can be very useful during debugging. For example, if you have a data race that only occurs under rare circumstances, adding sleep to your code might make it appear more often. This is how to make the currently running thread sleep for a second:

```
std::this_thread::sleep_for(std::chrono::seconds{1});
```

Your program should never expose any data races after inserting random sleeps in your code. Your program may not work satisfactorily after adding sleeps; buffers may become full, the UI may lag, and so on, but it should always behave in a predictable and defined way. We don't have control over the scheduling of the threads, and random sleeps simulate unlikely but possible scheduling scenarios.

Now, let's create an additional thread using the `std::thread` class from the `<thread>` header. It represents a single thread of execution and is usually a wrapper around an operating system thread. The `print()` function will be invoked from a thread created by us explicitly:

```
void print() {
  std::this_thread::sleep_for(std::chrono::seconds{1});
  std::cout << "Thread ID: "<<  std::this_thread::get_id() << '\n';
}

int main() {
  auto t1 = std::thread{print};
  t1.join();
```

```
  std::cout << "Thread ID: "<<  std::this_thread::get_id() << '\n';
}
```

When creating the thread, we pass in a callable object (a function, lambda, or a function object) that the thread will begin to execute whenever it gets scheduled time on the CPU. I have added a call to sleep to make it obvious why we need to call `join()` on the thread. When a `std::thread` object is destructed, it must have been *joined* or *detached* or it will cause the program to call `std::terminate()`, which by default will call `std::abort()` if we haven't installed a custom `std::terminate_handler`.

In the preceding example, the `join()` function is blocking — it waits until the thread has finished running. So, in the preceding example, the `main()` function will not return until thread `t1` has finished running. Consider the following line:

```
t1.join();
```

Suppose we detach the thread `t1` by replacing the preceding line with the following line:

```
t1.detach();
```

In such a case, our main function will end before thread `t1` wakes up to print the message, and, as a result, the program will (most likely) only output the thread ID of the main thread. Remember, we have no control of the scheduling of the threads and it is possible, but very unlikely, that the main thread will output its message *after* the `print()` function has had time to sleep, wake up, and print its thread ID.

Using `detach()` instead of `join()` in this example also introduces another problem. We are using `std::cout` from both threads without any synchronization, and since `main()` is no longer waiting for thread `t1` to finish, they both could theoretically use `std::cout` in parallel. Fortunately, `std::cout` is thread-safe and can be used from multiple threads without introducing data races, so there is no undefined behavior. However, it is still possible that the output generated by the threads is interleaved, resulting in something like the following:

```
Thread ID: Thread ID: 0x1003a93400x700004fd4000
```

If we want to avoid the interleaved output, we need to treat the outputting of characters as a critical section and synchronize access to `std::cout`. We will talk more about critical sections and race conditions in a while, but first, let's cover some details about `std::thread`.

Thread states

Before we go any further, you should have a good understanding of what a `std::thread` object really represents and in what states it can be. We haven't yet talked about what sort of threads there normally are in a system executing a C++ program.

In the following figure, you can see a snapshot of a hypothetical running system.

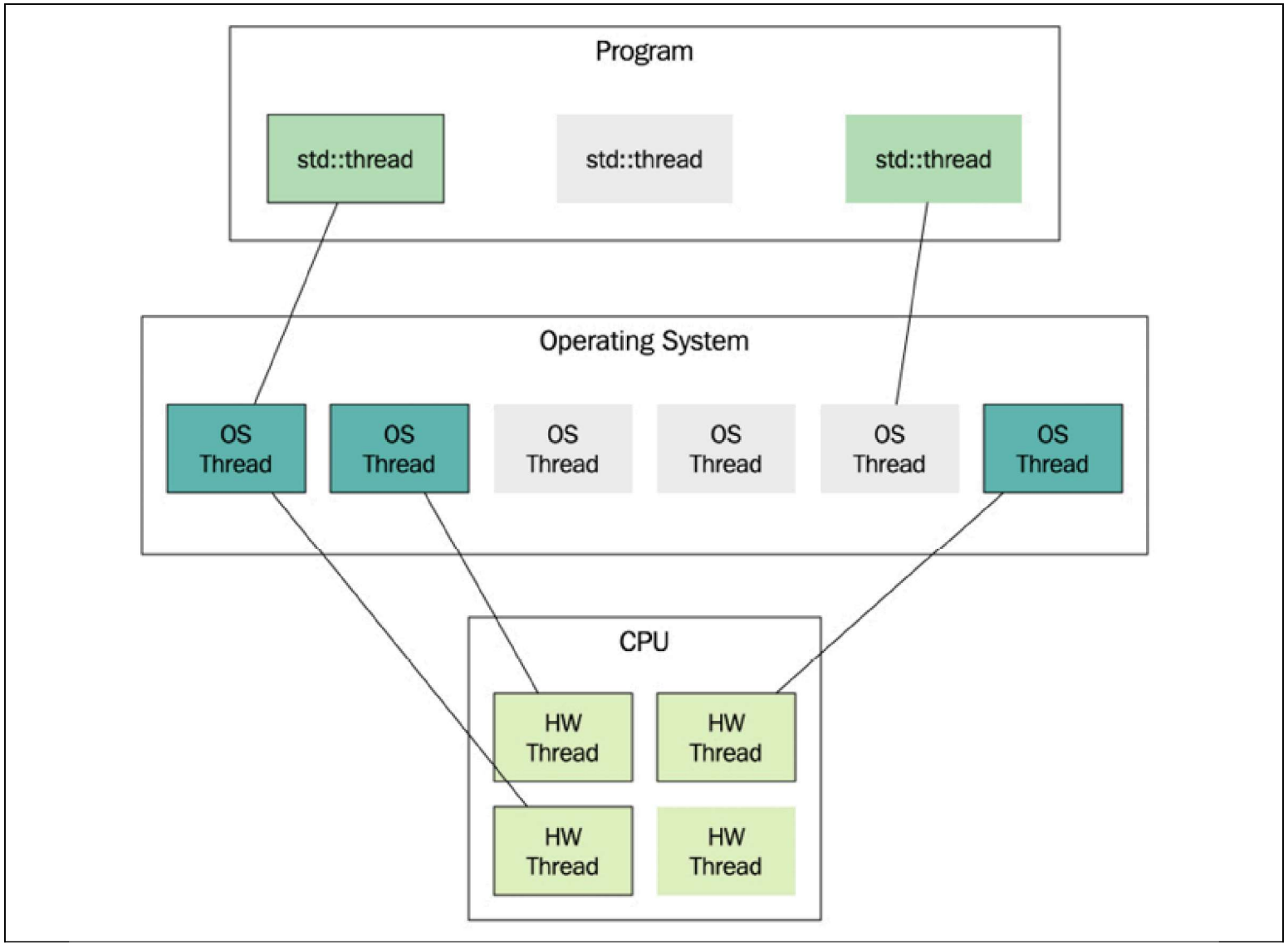

Figure 11.9: Snapshot of a hypothetical running system

Starting from the bottom, the figure shows the CPU and its **hardware threads**. Those are the execution units on the CPU. In this example, the CPU provides four hardware threads. Usually that means it has four

cores, but it could be some other configuration; for example, some cores can execute two hardware threads. This is usually called **hyperthreading**. The total number of hardware threads can be printed at runtime with this:

```
std::cout << std::thread::hardware_concurrency() << '\n';
// Possible output: 4
```

The preceding code might also output `0` if the number of hardware threads cannot be determined on the running platform.

The layer above the hardware threads contains the **operating system threads**. These are the actual software threads. The operating system scheduler determines when and for how long an operating system thread is executed by a hardware thread. In *Figure 11.9*, there are currently three out of six software threads executing.

The topmost layer in the figure contains the `std::thread` objects. A `std::thread` object is nothing more than an ordinary C++ object that may or may not be associated with an underlying operating system thread. Two instances of `std::thread` cannot be associated with the same underlying thread. In the figure, you can see that the program currently has three instances of `std::thread`; two are associated with threads and one is not. It's possible to use the `std::thread::joinable` property to find out what state a `std::thread` object is in. A thread is *not* joinable if it has been:

- Default constructed; that is, if it has nothing to execute
- Moved from (its associated running thread has been transferred to another `std::thread` object)
- Detached by a call to `detach()`
- Already joined by a call to `join()`

Otherwise, the `std::thread` object is in the joinable state. Remember, when a `std::thread` object is destructed, it must no longer be in the joinable state or the program will terminate.

Joinable thread

C++20 introduced a new thread class named `std::jthread`. It is very similar to `std::thread`, but with a couple of important additions:

- `std::jthread` has support for stopping a thread using a stop token. This is something that we had to implement manually before C++20 when using `std::thread`.
- Instead of terminating the app when it is being destructed in a non-joinable state, the destructor of `std::jthread` will send a stop request and join the thread on destruction.

I will illustrate the latter point next. First, we will use the `print()` function, which is defined like this:

```
void print() {
  std::this_thread::sleep_for(std::chrono::seconds{1});
  std::cout << "Thread ID: "<<  std::this_thread::get_id() << '\n';
}
```

It sleeps for a second, and then prints the current thread identifier:

```
int main() {
  std::cout << "main begin\n";
  auto joinable_thread = std::jthread{print};
```

```
  std::cout << "main end\n";
} // OK: jthread will join automatically
```

The following output was produced when running the code on my machine:

```
main begin
main end
Thread ID: 0x1004553c0
```

Now let's change our `print()` function so that it output messages continuously in a loop. We then need some way to communicate to the `print()` function when to stop. The `std::jthread` (as opposed to `std::thread`) has built-in support for this by using a stop token. When `std::jthread` invokes the `print()` function, it can pass an instance of a `std::stop_token` if the `print()` function accepts such an argument. Here is an example of how we could implement this new `print()` function using a stop token:

```
void print(std::stop_token stoken) {
  while (!stoken.stop_requested()) {
    std::cout << std::this_thread::get_id() << '\n';
    std::this_thread::sleep_for(std::chrono::seconds{1});
  }
  std::cout << "Stop requested\n";
}
```

The `while`-loop checks at each iteration whether the function has been requested to stop by calling `stop_requested()`. From our `main()` function, it's now possible to request a stop by calling

`request_stop()` on our `std::jthread` instance:

```
int main() {
  auto joinable_thread = std::jthread(print);
  std::cout << "main: goes to sleep\n";
  std::this_thread::sleep_for(std::chrono::seconds{3});
  std::cout << "main: request jthread to stop\n";
  joinable_thread.request_stop();
}
```

When I run this program, it generates the following output:

```
main: goes to sleep
Thread ID: 0x70000f7e1000
Thread ID: 0x70000f7e1000
Thread ID: 0x70000f7e1000
main: request jthread to stop
Stop requested
```

In this example, we could have omitted the explicit call to `request_stop()` because `jthread` will call `request_stop()` automatically on destruction.

The new `jthread` class is a welcome addition to the C++ thread library and it should be the first choice when reaching for a thread class in C++.

Protecting critical sections

As I already mentioned, our code must not contain any data races. Unfortunately, writing code with data races is very easy. Finding the critical sections and protecting them with locks is something we constantly need to think about when writing concurrent programs in this style using threads.

C++ provides us with a `std::mutex` class that can be used for protecting critical sections and avoiding data races. I will demonstrate how to use a mutex with a classic example using a shared mutable counter variable updated by multiple threads.

First, we define a global mutable variable and the function incrementing the counter:

```
auto counter = 0; // Warning! Global mutable variable
void increment_counter(int n) {
 for (int i = 0; i < n; ++i)
  ++counter;
}
```

The `main()` function that follows creates two threads that will both execute the `increment_counter()` function. Note also in this example how we can pass arguments to the function invoked by the thread. We can pass an arbitrary number of arguments to the thread constructor in order to match the parameters in the signature of the function to be called. Finally, we assert that the counter has the value we would expect it to have if the program was free from race conditions:

```
int main() {
 constexpr auto n = int{100'000'000};
 {
  auto t1 = std::jthread{increment_counter, n};
  auto t2 = std::jthread{increment_counter, n};
```

```
  }
  std::cout << counter << '\n';
  // If we don't have a data race, this assert should hold:
  assert(counter == (n * 2));
}
```

This program will most likely fail. The `assert()` function doesn't hold since the program currently contains a race condition. When I repeatedly run the program, I end up with different values of the counter. Instead of reaching the value `200000000`, I once ended up with no more than `137182234`. This example is very similar to the data race example that was illustrated earlier in this chapter.

The line with the expression `++counter` is a critical section — it uses a shared mutable variable and is executed by multiple threads. In order to protect the critical section, we will now use the `std::mutex` included in the `<mutex>` header. Later on, you will see how we can avoid data races in this example by using atomics, but, for now, we will use a lock.

First, we add the global `std::mutex` object next to the `counter`:

```
auto counter = 0; // Counter will be protected by counter_mutex
auto counter_mutex = std::mutex{};
```

But isn't the `std::mutex` object itself a mutable shared variable that can generate data races if used by multiple threads? Yes, it is a mutable shared variable, but no, it will not generate data races. The synchronization primitives from the C++ thread library, such as `std::mutex`, are designed for this particular purpose. In that respect, they are very special and use hardware instructions, or whatever is necessary on our platform, to guarantee that they don't generate data races themselves.

Now we need to use the mutex in our critical section that reads and updates the counter variable. We could use the `lock()` and `unlock()` member functions on the `counter_mutex`, but the preferred and safer method is to always use RAII for handling the mutex. Think of the mutex as a resource that always needs to be unlocked when we have finished using it. The thread library provides us with some useful RAII class templates for handling locking. Here, we will use the `std::scoped_lock<Mutex>` template to ensure that we release the mutex safely. Below is the updated `increment_counter()` function, which is now protected with a mutex lock:

```
void increment_counter(int n) {
 for (int i = 0; i < n; ++i) {
  auto lock = std::scoped_lock{counter_mutex};
  ++counter;
 }
}
```

The program is now free from data races and works as expected. If we run it again, the condition in the `assert()` function will now hold true.

Avoiding deadlocks

As long as a thread never acquires more than one lock at a time, there is no risk of deadlocks. Sometimes, though, it is necessary to acquire another lock while already holding onto a previously acquired lock. The risk of deadlocks in these situations can be avoided by grabbing both locks at the exact same time. C++ has a way to do this by using the `std::lock()` function, which takes an arbitrary number of locks and blocks until all locks have been acquired.

The following is an example of transferring money between accounts. Both accounts need to be protected during the transaction, and therefore we need to acquire two locks at the same time. Here is how it works:

```
struct Account {
  Account() {}
  int balance_{0};
  std::mutex m_{};
};

void transfer_money(Account& from, Account& to, int amount) {
   auto lock1 = std::unique_lock<std::mutex>{from.m_, std::defer_lock};
   auto lock2 = std::unique_lock<std::mutex>{to.m_, std::defer_lock};

   // Lock both unique_locks at the same time
   std::lock(lock1, lock2);

   from.balance_ -= amount;
   to.balance_ += amount;
}
```

We again use a RAII class template to ensure that we release the lock whenever this function returns. In this case, we use `std::unique_lock`, which provides us with the possibility to defer the locking of the mutex. Then, we explicitly lock both mutexes at the same time by using the `std::lock()` function.

Condition variables

A **condition variable** makes it possible for threads to wait until some specific condition has been met. Threads can also use a condition variable to signal to other threads that the condition has changed.

A common pattern in a concurrent program is to have one or many threads that are waiting for data to be consumed somehow. These threads are usually called **consumers**. Another group of threads is then responsible for producing data that is ready to be consumed. These threads producing data are called **producers**, or a **producer** if it is only one thread.

The producer and consumer pattern can be implemented using a condition variable. We can use a combination of `std::condition_variable` and `std::unique_lock` for this purpose. Let's have a look at an example of a producer and consumer to make them less abstract:

```
auto cv = std::condition_variable{};
auto q = std::queue<int>{};
auto mtx = std::mutex{};    // Protects the shared queue
constexpr int sentinel = -1; // Value to signal that we are done

void print_ints() {
 auto i = 0;
 while (i != sentinel) {
  {
   auto lock = std::unique_lock<std::mutex>{mtx};
   while (q.empty()) {
    cv.wait(lock); // The lock is released while waiting
   }
   i = q.front();
   q.pop();
  }
```

```
    if (i != sentinel) {
      std::cout << "Got: " << i << '\n';
    }
  }
}

auto generate_ints() {
  for (auto i : {1, 2, 3, sentinel}) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
      auto lock = std::scoped_lock{mtx};
      q.push(i);
    }
    cv.notify_one();
  }
}

int main() {
  auto producer = std::jthread{generate_ints};
  auto consumer = std::jthread{print_ints};
}
```

We are creating two threads: one `consumer` thread and one `producer` thread. The `producer` thread generates a sequence of integers and pushes them to a global `std::queue<int>` once every second. Whenever an element is added to the queue, the producer signals that the condition has changed using `notify_one()`.

The program checks whether there is data in the queue that is available for consumption by the consumer thread. Note also that it is not required to hold the lock while notifying the condition variable.

The consumer thread is responsible for printing the data (that is, the integers) to the console. It uses the condition variable to wait for the empty queue to change. When the consumer calls `cv.wait(lock)`, the thread goes to sleep and leaves the CPU for other threads to execute. It is important to understand why we need to pass the variable `lock` when calling `wait()`. Apart from putting the thread to sleep, `wait()` also unlocks the mutex while sleeping and then acquires the mutex before it returns. If `wait()` didn't release the mutex, the producer would not be able to add elements to the queue.

Why is the consumer waiting on the condition variable with a `while`-loop around it and not an `if` statement? This is a common pattern, and sometimes we need to do that since there might be other consumers that were also woken up and emptied the queue before us. In our program, we only have one consumer thread, though, so that cannot happen. However, it is possible for the consumer to be awoken from its wait even though the producer thread did not signal. This phenomenon is called **spurious wakeup**, and the reasons that this can happen are beyond the scope of this book.

As an alternative to using a `while`-loop, we can use an overloaded version of `wait()` that accepts a predicate. This version of `wait()` check if the predicate is satisfied and will do the looping for us. In our example it would look like this:

```
// ...
auto lock = std::unique_lock<std::mutex>{mtx};
cv.wait(lock, [] { return !q.empty(); });
// ...
```

You can find more information about spurious wakeups in *C++ Concurrency in Action*, *Second Edition*, by Anthony Williams. You now at least know how to handle situations where spurious wakeups can happen:

always check the condition in a while loop or use the overloaded version of `wait()` that accepts a predicate.

Condition variables and mutexes are synchronization primitives that have been available in C++ since the introduction of threads in C++. C++20 comes with additional useful class templates for synchronizing threads, namely `std::counting_semaphore`, `std::barrier`, and `std::latch`. We will cover these new primitives later on. First we are going to spend some time on return values and error handling.

Returning data and handling errors

The examples presented so far in this chapter have used shared variables to communicate state between threads. We have used mutex locks to ensure that we avoid data races. Using shared data with mutexes, as we have been doing, can be very hard to do correctly when the size of a program increases. There is also a lot of work in maintaining code that uses explicit locking spread out over a code base. Keeping track of shared memory and explicit locking moves us further away from what we really want to accomplish and spend time on when writing a program.

In addition, we haven't dealt with error handling at all yet. What if a thread needs to report an error to some other thread? How do we do that using exceptions, as we are used to doing when a function needs to report a runtime error?

In the standard library `<future>` header, we can find some class templates that help us with writing concurrent code without global variables and locks, and, in addition, can communicate exceptions between threads for handling errors. I will now present **futures** and **promises**, which represent two sides of a value. The future is the receiving side of the value and the promise is the returning side of the value.

The following is an example of using `std::promise` to return the result to the caller:

```cpp
auto divide(int a, int b, std::promise<int>& p) {
  if (b == 0) {
    auto e = std::runtime_error{"Divide by zero exception"};
    p.set_exception(std::make_exception_ptr(e));
  }
  else {
    const auto result = a / b;
    p.set_value(result);
  }
}

int main() {
  auto p = std::promise<int>{};
  std::thread(divide, 45, 5, std::ref(p)).detach();

  auto f = p.get_future();
  try {
    const auto& result = f.get(); // Blocks until ready
    std::cout << "Result: " << result << '\n';
  }
  catch (const std::exception& e) {
    std::cout << "Caught exception: " << e.what() << '\n';
  }
}
```

The caller (the `main()` function) creates the `std::promise` object and passes it to the `divide()` function. We need to use `std::ref` from `<functional>` so that a reference can be correctly forwarded through the `std::thread` to `compute()`.

When the `divide()` function has computed the result, it passes the return value through the promise by calling the `set_value()` function. If an error occurs in the `divide()` function, it calls the `set_exception()` function on the promise instead.

The future represents the value of the computation that may or may not be computed yet. Since the future is an ordinary object, we can, for example, pass it around to other objects that need the computed value. Finally, when the value is needed by some client, it calls `get()` to get hold of the actual value. If it is not computed at that point in time, the call to `get()` will block until it is finished.

Note also how we managed to pass data back and forth with proper error handling, without using any shared global data and with no explicit locking. The promise takes care of that for us, and we can focus on implementing the essential logic of the program instead.

Tasks

With futures and promises, we managed to get away from explicit locks and shared global data. Our code will benefit from using higher-level abstractions when possible, especially when the code base grows. Here, we will go further and explore classes that automatically set up the futures and promises for us. You will also see how we can get rid of the manual administration of threads and leave that to the library.

In many cases, we don't have any need for managing threads; instead, what we really need is to be able to execute a **task** asynchronously and have that task execute on its own concurrently with the rest of the program, and then eventually get the result or error communicated to the parts of the program that need it. The task should be carried out in isolation to minimize contention and the risk of data races.

We will begin by rewriting our previous example that divided two numbers. This time, we will use the `std::packaged_task` from `<future>`, which makes all the work of setting up the promise correct for us:

```cpp
int divide(int a, int b) { // No need to pass a promise ref here!
  if (b == 0) {
    throw std::runtime_error{"Divide by zero exception"};
  }
  return a / b;
}

int main() {
  auto task = std::packaged_task<decltype(divide)>{divide};
  auto f = task.get_future();
  std::thread{std::move(task), 45, 5}.detach();

  // The code below is unchanged from the previous example
  try {
    const auto& result = f.get(); // Blocks until ready
    std::cout << "Result: " << result << '\n';
  }
  catch (const std::exception& e) {
    std::cout << "Caught exception: " << e.what() << '\n';
  }
  return 0;
}
```

`std::packaged_task` is itself a callable object that can be moved to the `std::thread` object we are creating. As you can see, `std::packaged_task` now does most of the work for us: we don't have to create the promise ourselves. But, more importantly, we can write our `divide()` function just like a normal function, without the need for explicitly returning values or exceptions through the promise; the `std::packaged_task` will do that for us.

As a last step in this section, we would also like to get rid of the manual thread management. Creating threads is not free, and you will see later on that the number of threads in a program can affect performance. It seems like the question of whether we should create a new thread for our `divide()` function is not necessarily up to the caller of `divide()`. The library again helps us here by providing another useful function template called `std::async()`. The only thing we need to do in our `divide()` example is replace the code creating the `std::packaged_task` and the `std::thread` object with a simple call to `std::async()`:

```
auto f = std::async(divide, 45, 5);
```

We have now switched from a thread-based programming model to a task-based model. The complete task-based example now looks like this:

```
int divide(int a, int b) {
  if (b == 0) {
    throw std::runtime_error{"Divide by zero exception"};
  }
  return a / b;
}

int main() {
  auto future = std::async(divide, 45, 5);
  try {
    const auto& result = future.get();
    std::cout << "Result: " << result << '\n';
  }
  catch (const std::exception& e) {
    std::cout << "Caught exception: " << e.what() << '\n';
```

```
  }
}
```

There is really a minimal amount of code left here for handling concurrency. The recommended way to call functions asynchronously is to use `std::async()`. For a deeper discussion about why and when `std::async()` is preferred, I highly recommend the *Concurrency* chapter in *Effective Modern C++* by Scott Meyers.

Additional synchronization primitives in C++20

C++20 comes with a few additional synchronization primitives, namely `std::latch`, `std::barrier`, and `std::counting_semaphore` (and the template specialization `std::binary_semaphore`). This section will be an overview of these new types and some typical scenarios where they can be useful. We'll begin with `std::latch`.

Using latches

A latch is a synchronization primitive that can be used for synchronizing multiple threads. It creates a synchronization point where all threads must arrive at. You can think of a latch as a decrementing counter. Typically, all threads decrement the counter once and then wait for the latch to reach zero before moving on.

A latch is constructed by passing an initial value of the internal counter:

```
auto lat = std::latch{8}; // Construct a latch initialized with 8
```

Threads can then decrement the counter using `count_down()`:

```
lat.count_down(); // Decrement but don't wait
```

A thread can wait on the latch to reach zero:

```
lat.wait(); // Block until zero
```

It's also possible to check (without blocking) to see whether the counter has reached zero:

```
if (lat.try_wait()) {
  // All threads have arrived ...
}
```

It's common to wait for the latch to reach zero right after decrementing the counter, as follows:

```
lat.count_down();
lat.wait();
```

In fact, this use case is common enough to deserve a tailor-made member function; `arrive_and_wait()` decrements the latch and then waits for the latch to reach zero:

```
lat.arrive_and_wait(); // Decrement and block while not zero
```

Joining a set of forked tasks is a common scenario when working with concurrency. If the tasks only need to be joined at the end, we can use an array of future objects (to wait on) or just wait for all the threads to complete. But in other cases, we want a set of asynchronous tasks to arrive at a common synchronization point, and then have the tasks continue running. These situations typically occur when some sort of initialization is needed before multiple worker threads start their actual work.

Example: Initializing threads using std::latch

The following example demonstrates how `std::latch` can be used when multiple worker threads need to run some initialization code before they start working.

When a thread is created, a contiguous block of memory is allocated for the stack. Typically, this memory does not yet reside in physical memory when it is first allocated in the virtual address space. Instead, when the stack is being used, *page faults* will be generated in order to map the virtual memory to physical memory. The operating system handles the mapping for us, and it is an efficient way to lazily map memory when needed. Usually, this is just what we want: we pay for the cost of mapping memory as late as possible and only if needed. However, in circumstances where low latency is important, for example in real-time code, it might be necessary to completely avoid page faults. The stack memory is unlikely to be paged out by the operating system, so it is usually enough to run some code that will generate page faults and thereby map the virtual stack memory to physical memory. This process is called **prefaulting**.

There is no portable way to set or get the stack size of a C++ thread, so here we will just assume that the stack is at least 500 KB. The following code is an attempt to prefault the first 500 KB of the stack:

```
void prefault_stack() {
 // We don't know the size of the stack
 constexpr auto stack_size = 500u * 1024u;
```

```
  // Make volatile to avoid optimization
  volatile unsigned char mem[stack_size];
  std::fill(std::begin(mem), std::end(mem), 0);
}
```

The idea here is to allocate an array on the stack that will occupy a significant chunk of stack memory. Then, in order to generate page faults, we write to every element in the array using `std::fill()`. The volatile keyword was not mentioned earlier and is a somewhat confusing keyword in C++. It has nothing to do with concurrency; it's only added here to prevent the compiler from optimizing away this code. By declaring the `mem` array `volatile`, the compiler is not allowed to ignore the writes to the array.

Now, let's focus on the actual `std::latch`. Let's say we want to create a number of worker threads that should only start their work once all thread stacks have been prefaulted. We can achieve this synchronization using a `std::latch`, as follows:

```
auto do_work() { /* ... */ }
int main() {
  constexpr auto n_threads = 2;
  auto initialized = std::latch{n_threads};
  auto threads = std::vector<std::thread>{};
  for (auto i = 0; i < n_threads; ++i) {
    threads.emplace_back([&] {
      prefault_stack();
      initialized.arrive_and_wait();
      do_work();
    });
  }
  initialized.wait();
```

```
  std::cout << "Initialized, starting to work\n";
  for (auto&& t : threads) {
    t.join();
  }
}
```

After all threads have arrived, the main thread can start to submit work to the worker threads. In this example, all threads are waiting for the other threads to arrive by calling `arrive_and_wait()` on the latch. Once the latch has reached zero, it can no longer be reused. There is no function for resetting the latch. If we have a scenario that requires multiple synchronization points, we can instead use a `std::barrier`.

Using barriers

Barriers are similar to latches but with two major additions: a barrier can be *reused*, and it can run a *completion function* whenever all threads have reached the barrier.

A barrier is constructed by passing an initial value of the internal counter and a completion function:

```
auto bar = std::barrier{8, [] {
  // Completion function
  std::cout "All threads arrived at barrier\n";
}};
```

Threads can arrive and wait in the same way we use a latch:

```
bar.arrive_and_wait(); // Decrement but don't wait
```

Whenever all threads have arrived (that is, when the internal counter of the barrier reaches zero) two things happens:

- The completion function provided to the constructor is called by the barrier.
- The internal counter is reset to its initial value after the completion function has returned.

Barriers are useful in parallel programming algorithms that are based on the **fork-join model**. Typically, an iterative algorithm contains a part that can be run in parallel and another part that needs to run sequentially. Multiple tasks are forked and run in parallel. Then, when all tasks have finished and joined, some single-threaded code is executed to determine whether the algorithm should continue or finish.

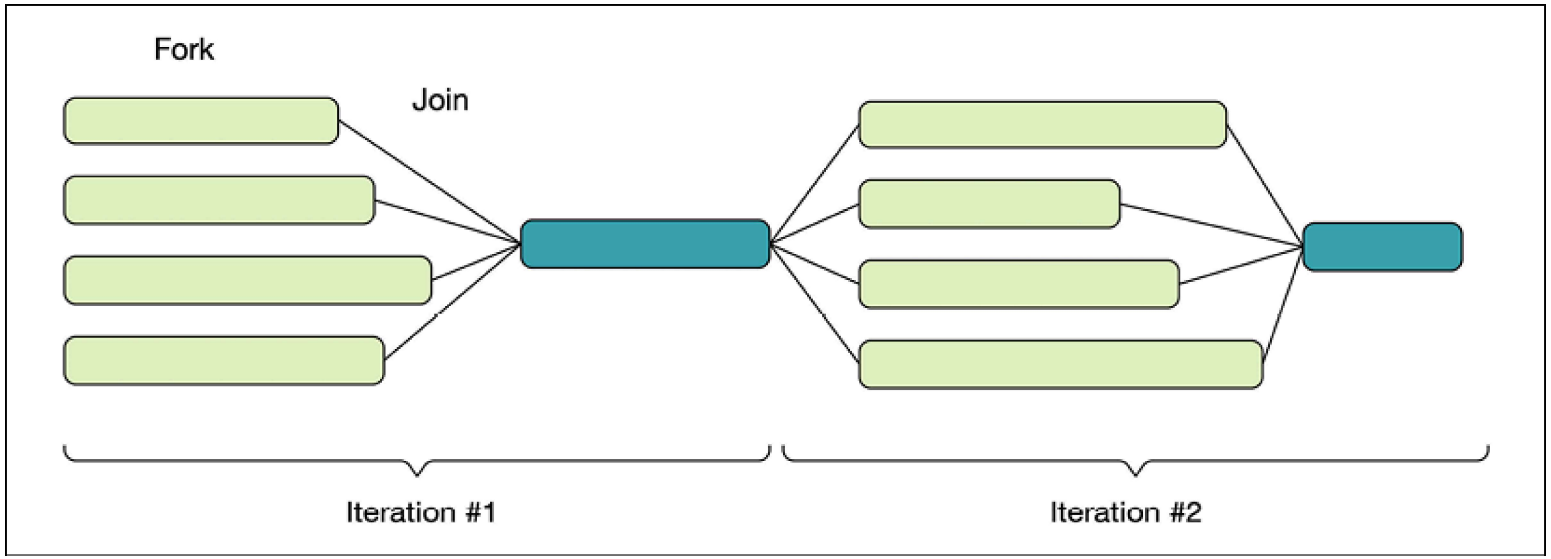

Figure 11.10: An example of the fork-join model

Concurrent algorithms that follow the fork-join model will benefit from using barriers and can avoid other explicit locking mechanisms in an elegant and efficient way. Let's see how we can use a barrier but

with two major for a simple problem.

Example: Fork-join using std::barrier

Our next example is a toy problem that will demonstrate the fork-join model. We will create a small program that will simulate a set of dice being rolled and count the number of rolls it takes before getting all 6s. Rolling a set of dice is something we can do concurrently (forked). The join step, executing in a single thread, checks the result and determines whether to roll the dice again or to finish.

First, we need to implement the code for rolling a dice with six faces. For generating a number between 1 and 6 we can use a combination of classes found in the `<random>` header, as follows:

```
auto engine =
  std::default_random_engine{std::random_device{}()};
auto dist = std::uniform_int_distribution<>{1, 6};
auto result = dist(engine);
```

Here the `std::random_device` is responsible for generating a seed to the engine that will produce pseudo-random numbers. To pick an integer between 1 and 6 with equal probability, we are using `std::uniform_int_distribution`. The variable `result` is the result of rolling a dice.

Now we want to encapsulate this code into a function that will generate a random integer. Generating the seed and creating the engine is typically slow and something we want to avoid doing at every call. A common way to do this is to declare the random engine with `static` duration so that it lives during the entire lifetime of the program. However, the classes in `<random>` are not thread-safe so we need to protect the `static` engine somehow. Instead of synchronizing access with a mutex, which would make

the random number generator run sequentially, I will take the opportunity to demonstrate how to use thread-local storage.

Here is how to declare the engine as a `static thread_local` object:

```
auto random_int(int min, int max) {
  // One engine instance per thread
  static thread_local auto engine =
    std::default_random_engine{std::random_device{}()};
  auto dist = std::uniform_int_distribution<>{min, max};
  return dist(engine);
}
```

A static variable with `thread_local` storage duration will be created once per thread; it's therefore safe to call `random_int()` from multiple threads concurrently without using any synchronization primitives. With this small helper function in place, we can move on to implement the rest of our program using a `std::barrier`:

```
int main() {
  constexpr auto n = 5; // Number of dice
  auto done = false;
  auto dice = std::array<int, n>{};
  auto threads = std::vector<std::thread>{};
  auto n_turns = 0;
  auto check_result = [&] { // Completion function
    ++n_turns;
    auto is_six = [](auto i) { return i == 6; };
```

```
    done = std::all_of(dice.begin(), dice.end(), is_six);
  };
  auto bar = std::barrier{n, check_result};
  for (int i = 0; i < n; ++i) {
    threads.emplace_back([&, i] {
      while (!done) {
        dice [i] = random_int(1, 6); // Roll dice
        bar.arrive_and_wait();       // Join
      }});
  }
  for (auto&& t : threads) {
    t.join();
  }
  std::cout << n_turns << '\n';
}
```

The lambda `check_result()` is the completion function that will be called every time all the threads have arrived at the barrier. The completion function checks the values of each dice and determines whether a new round should be played or if we are done.

The lambda passed to the `std::thread` objects captures the index `i` by value so that all threads have a unique index. The other variables, `done`, `dice`, and `bar`, are captured by reference.

Note also how we can mutate and read the variables captured by reference from different threads without introducing any data races thanks to the coordination performed by the barrier.

Signalling and resource counting using semaphores

The word **semaphore** means something that can be used for signaling, such as a flag or a light. In the example that follows, you will see how we can use semaphores for signaling different states that other threads can be waiting for.

A semaphore can also be used to control access to a resource, similarly to how a `std::mutex` restricts access to a critical section:

```
class Server {
public:
  void handle(const Request& req) {
    sem_.acquire();
    // Restricted section begins here.
    // Handle at most 4 requests concurrently.
    do_handle(req);
    sem_.release();
  }
private:
  void do_handle(const Request& req) { /* ... */ }
  std::counting_semaphore<4> sem_{4};
};
```

In this case, the semaphore is initialized with a value of `4`, which means that at most four concurrent requests can be handled at the same time. Instead of mutually exclusive access to a section in the code, multiple threads can have access to the same section but with restrictions concerning the number of threads currently in that section.

The member function `acquire()` decrements the semaphore if the semaphore is greater than zero. Otherwise `acquire()` blocks until the semaphore allows it to decrement and enter the restricted section. `release()` increments the counter without blocking. If the semaphore was zero before it was incremented by `release()`, waiting threads will be signaled.

In addition to the `acquire()` function, it's also possible to try to decrement the counter *without blocking* using the `try_acquire()` function. It returns `true` if it managed to decrement the counter, or `false` otherwise. The functions `try_acquire_for()` and `try_acquire_until()` can be used in a similar way. But instead of immediately returning `false` when the counter is already zero, they automatically try to decrement the counter within a specified time before returning to the caller.

This trio of functions follows the same pattern as other types in the standard library, for example, `std::timed_mutex` and its `try_lock()`, `try_lock_for()`, and `try_lock_until()` member functions.

The `std::counting_semaphore` is a template with one template parameter accepting the maximum value of the semaphore. It is considered a programming error to increment (release) a semaphore above its maximum value.

A `std::counting_semaphore` with a maximum size of 1 is called a **binary semaphore**. The `<semaphore>` header includes an alias-declaration for binary semaphores:

```
std::binary_semaphore = std::counting_semaphore<1>;
```

A binary semaphore is guaranteed to be implemented more efficiently than a counting semaphore with a higher maximum value.

Another important property of semaphores is that the thread that releases a semaphore may not be the thread that acquired it. This is in contrast with `std::mutex`, which requires that the thread that acquired the mutex is also the thread that must release it. However, with semaphores it's common to have one type of task to do the waiting (acquire) and another type of task to do the signaling (release). This will be demonstrated in our next example.

Example: A bounded buffer using semaphores

The following example demonstrates a bounded buffer. It's a fixed-size buffer that can have multiple threads reading and writing from it. Again, this example demonstrates the kind of producer-consumer pattern that you have already seen using condition variables. The producer threads are the ones writing to the buffer and the reader threads are the ones reading (and popping elements) from the buffer.

The following figure shows the buffer (a fixed-size array) and the two variables that keep track of the read and write positions:

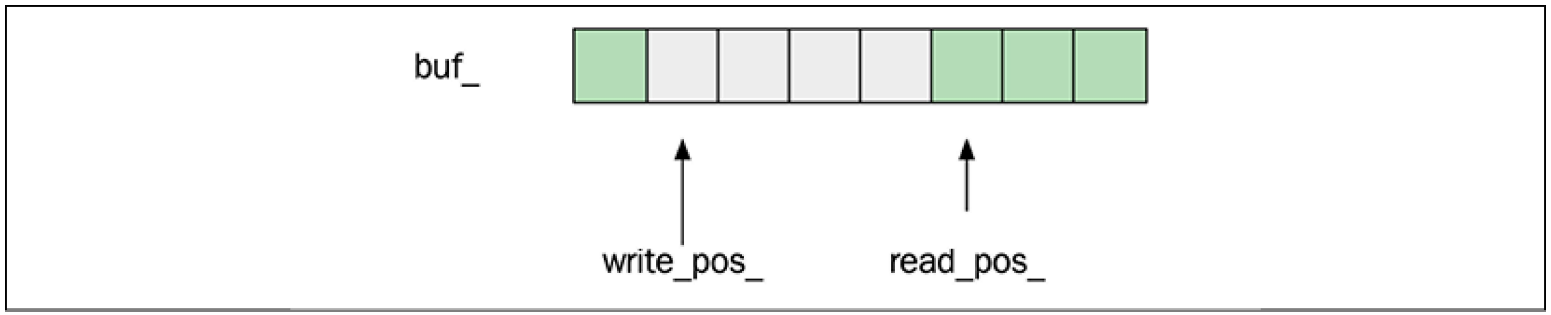

Figure 11.11: A bounded buffer has a fixed size

We will take one step at a time and start with a version that focuses on the internal logic of the bounded buffer. The signaling using semaphores will be added in the next version. Here, the initial attempt demonstrates how the read and write positions are used:

```
template <class T, int N>
class BoundedBuffer {
  std::array<T, N> buf_;
  std::size_t read_pos_{};
  std::size_t write_pos_{};
  std::mutex m_;
  void do_push(auto&& item) {
    /* Missing: Should block if buffer is full */
    auto lock = std::unique_lock{m_};
    buf_[write_pos_] = std::forward<decltype(item)>(item);
    write_pos_ = (write_pos_ + 1) % N;
  }
public:
  void push(const T& item) { do_push(item); }
  void push(T&& item) { do_push(std::move(item)); }
  auto pop() {
    /* Missing: Should block if buffer is empty */
    auto item = std::optional<T>{};
    {
      auto lock = std::unique_lock{m_};
      item = std::move(buf_[read_pos_]);
      read_pos_ = (read_pos_ + 1) % N;
    }
    return std::move(*item);
```

```
  }
};
```

This first attempt contains the fixed-sized buffer, the read and write positions, and a mutex for protecting the data members from data races. This implementation should be able to have an arbitrary number of threads calling `push()` and `pop()` concurrently.

The `push()` function overloads on `const T&` and `T&&`. This is an optimization technique used by the standard library containers. The `T&&` version avoids copying the argument when the caller passes an rvalue.

To avoid duplicating the logic of the push operation, a helper function, `do_push()`, contains the actual logic. By using a forwarding reference (`auto&& item`) together with `std::forward`, the `item` parameter will be move assigned or copy assigned, depending on whether the client called `push()` with an rvalue or lvalue.

This version of the bounded buffer is not complete, though, because it doesn't protect us from having the `write_pos` point at (or beyond) the `read_pos`. Similarly, the `read_pos` must never point at the `write_pos` (or beyond). What we want is a buffer where producer threads block when the buffer is full and consumer threads block when the buffer is empty.

This is a perfect application for using counting semaphores. A semaphore *blocks* a thread that tries to decrease the semaphore when it is already zero. A semaphore *signals* the blocked threads whenever a semaphore that has the value zero increments.

For the bounded buffer we need two semaphores:

- The first semaphore, `n_empty_slots`, keeps track of the number of empty slots in the buffer. It will start with a value of the size of the buffer.
- The second semaphore, `n_full_slots`, keeps track of the number of full slots in the buffer.

Make sure you understand why two counting semaphores are needed (rather than one). The reason is that there are two distinct *states* that need to be signaled: when the buffer is *full* and when the buffer is *empty*.

After adding signal handling using two counting semaphores, the bounded buffer now looks like this (lines added in this version are marked with "new"):

```
template <class T, int N>
class BoundedBuffer {
  std::array<T, N> buf_;
  std::size_t read_pos_{};
  std::size_t write_pos_{};
  std::mutex m_;
  std::counting_semaphore<N> n_empty_slots_{N}; // New
  std::counting_semaphore<N> n_full_slots_{0};  // New
  void do_push(auto&& item) {
    // Take one of the empty slots (might block)
    n_empty_slots_.acquire();                   // New
    try {
      auto lock = std::unique_lock{m_};
      buf_[write_pos_] = std::forward<decltype(item)>(item);
      write_pos_ = (write_pos_ + 1) % N;
    } catch (...) {
      n_empty_slots_.release();                 // New
```

```
      throw;
    }
    // Increment and signal that there is one more full slot
    n_full_slots_.release();              // New
  }
public:
  void push(const T& item) { do_push(item); }
  void push(T&& item) { do_push(std::move(item)); }
  auto pop() {
    // Take one of the full slots (might block)
    n_full_slots_.acquire();              // New
    auto item = std::optional<T>{};
    try {
      auto lock = std::unique_lock{m_};
      item = std::move(buf_[read_pos_]);
      read_pos_ = (read_pos_ + 1) % N;
    } catch (...) {
      n_full_slots_.release();            // New
      throw;
    }
    // Increment and signal that there is one more empty slot
    n_empty_slots_.release();             // New
    return std::move(*item);
  }
};
```

This version supports multiple producers and consumers. The use of both semaphores guarantees that neither of the semaphores will reach a value greater than the maximum number of elements in the buf-

fer. For example, there is no way a producer thread can add a value and increment the `n_full_slots` semaphore without first checking that there is at least one empty slot.

Note also that `acquire()` and `release()` are called from different threads. For example, the consumer threads are waiting (`acquire()`) on the `n_full_slots` semaphore and the producer threads are signaling (`release()`) on the very same semaphore.

The new synchronization primitives added to C++20 are well known constructs that are commonly found in threading libraries. They offer convenient and often more efficient alternatives to synchronize access to shared resources compared to `std::mutex` and `std::condition_variable` .

Atomic support in C++

The standard library contains support for **atomic variables**, sometimes called **atomics**. An atomic variable is a variable that can safely be used and mutated from multiple threads without introducing data races.

Do you remember the data race example we looked at earlier where two threads updated a global counter? We solved it by adding a mutex lock together with the counter. Instead of using an explicit lock, we could have used a `std::atomic<int>` instead:

```
std::atomic<int> counter;

auto increment_counter(int n) {
  for (int i = 0; i < n; ++i)
    ++counter; // Safe, counter is now an atomic<int>
}
```

The `++counter` is a convenient way of saying `counter.fetch_add(1)`. All member functions that can be invoked on an atomic are safe to call from multiple threads concurrently.

The atomic types are from the `<atomic>` header. There are typedefs for all the scalar data types named on the `std::atomic_int` form. This is identical to saying `std::atomic<int>`. It is possible to wrap a custom type in a `std::atomic` template, as long as the custom type is trivially copyable. Basically, this means that an object of a class is fully described by the bits of its data members. In that way, an object can be copied with, for example, `std::memcpy()`, by only copying the raw bytes. So, if a class contains virtual functions, pointers to dynamic memory, and so on, it's no longer possible to just copy the raw bits of the object and expect it to work, and hence it is not trivially copyable. This can be checked at compile time, so you will get a compilation error if you try to create an atomic of a type that is not trivially copyable:

```
struct Point {
  int x_{};
  int y_{};
};

auto p = std::atomic<Point>{};      // OK: Point is trivially copyable
auto s = std::atomic<std::string>{}; // Error: Not trivially copyable
```

It's also possible to create atomic pointers. This makes the pointer itself atomic, but not the object it points at. We will talk more about atomic pointers and references in a while.

The lock-free property

A reason for using atomics rather than protecting access to a variable with a mutex is to avoid the performance overhead introduced by using `std::mutex`. Also, the fact that a mutex can block the thread for a

non-deterministic duration of time and introduce priority inversion (see the section *Thread priorities*) rules out mutexes in low latency contexts. In other words, there might be parts of your code with latency requirements that completely forbid the use of mutexes. In those cases, it's important to know whether an atomic variable is using a mutex.

An atomic variable may or may not use a lock to protect the data; this depends on the type of the variable and the platform. If the atomic does not use a lock, it is said to be **lock-free**. You can query the variable in runtime if it's lock-free:

```
auto variable = std::atomic<int>{1};
assert(variable.is_lock_free());        // Runtime assert
```

This is good, because now we at least assert when running the program that using the `variable` object is lock-free. Typically, all atomic objects of the same type will be either lock-free or not, but on some exotic platforms there is a possibility that two atomic objects might generate different answers.

It's generally more interesting to know whether an atomic type (`std::atomic<T>`) is guaranteed to be lock-free on a certain platform, and preferably we would like to know that at compile time rather than runtime. Since C++17, it's also possible to verify that an atomic specialization is lock-free at compile time by using `is_always_lock_free()` , like this:

```
static_assert(std::atomic<int>::is_always_lock_free);
```

This code will generate a compilation error if `atomic<int>` is not lock-free on the platform we are targeting. Now, if we compile a program that assumes that `std::atomic<int>` doesn't use locks, it will fail to

compile, which is exactly want we want.

On modern platforms, any `std::atomic<T>` where `T` fits into the native word size will typically be *always lock-free*. And on modern x64 chips, you even get double that amount. For example, on libc++ compiled on a modern Intel CPU, `std::atomic<std::complex<double>>` is always lock-free.

Atomic flags

An atomic type that is guaranteed to always be lock-free is `std::atomic_flag` (regardless of the target platform). As a consequence, `std::atomic_flag` does not provide us with the `is_always_lock_free()` / `is_lock_free()` functions since they would always return `true`.

Atomic flags can be used to protect critical sections as an alternative to using `std::mutex`. Since a lock is conceptually easy to understand, I will use that as an example here. It should be noted, though, that the implementations of locks that I demonstrate in this book are not production-ready code, but rather conceptual implementation. The following example demonstrates how to conceptually implement a simple spinlock:

```
class SimpleMutex {
  std::atomic_flag is_locked_{};        // Cleared by default
public:
  auto lock() noexcept {
    while (is_locked_.test_and_set()) {
      while (is_locked_.test());        // Spin here
    }
  }
  auto unlock() noexcept {
    is_locked_.clear();
```

```
  }
};
```

The `lock()` function calls `test_and_set()` to set the flag and at the same time obtain the previous value of the flag. If `test_and_set()` returns `false`, it means that the caller managed to acquire the lock (setting the flag when it was previously cleared). Otherwise, the inner `while`-loop will constantly poll the state of the flag using `test()` in a spinning loop. The reason we use `test()` in an extra inner loop is performance: `test()` doesn't invalidate the cache line, whereas `test_and_set()` does. This locking protocol is called **test and test-and-set**.

This spinlock works but is not very resource-friendly; when the thread is executing, it constantly uses the CPU to check the same condition over and over again. We could add a short sleep with an exponential backoff in each iteration, but finetuning this for various platforms and scenarios is hard.

Fortunately, C++20 added a wait and notify API to `std::atomic`, which makes it possible for threads to wait (in a resource-friendly manner) on an atomic variable to change its value.

Atomic wait and notify

Since C++20, `std::atomic` and `std::atomic_flag` provide the functionality for waiting and notifying. The function `wait()` blocks the current thread until the value of the atomic variable changes and some other thread notifies the waiting thread about it. A thread can notify that a change has occurred by calling either `notify_one()` or `notify_all()`.

With this new functionality, we can avoid continuously polling the state of the atomic and instead wait in a more resource-friendly way until the value changes; this is similar to how a `std::condition_variable` al-

lows us to wait and notify state changes.

By using wait and notify, the `SimpleMutex` implemented in the previous section can be rewritten like this:

```
class SimpleMutex {
  std::atomic_flag is_locked_{};
public:
  auto lock() noexcept {
    while (is_locked_.test_and_set())
      is_locked_.wait(true);    // Don't spin, wait
  }
  auto unlock() noexcept {
    is_locked_.clear();
    is_locked_.notify_one();   // Notify blocked thread
  }
};
```

We pass the old value (`true`) to `wait()` . By the time `wait()` returns, the atomic variable is guaranteed to have changed so that it is no longer `true` . However, there is no guarantee that we will catch *all* the changes to the variable. The variable might have changed from state A to state B and then back to state A without notifying the waiting thread. This is a phenomenon in lock-free programming called the **ABA problem**.

This example demonstrated the wait and notify functions using `std::atomic_flag` . The same wait and notify API is also available on the `std::atomic` class template.

Please note that the spinlocks presented in this chapter are not production-ready code. Implementing a highly efficient lock typically involves the correct use of memory orderings (discussed later) and non-portable code for yielding, which is beyond the scope of this book. A detailed discussion can be found at **https://timur.audio/using-locks-in-real-time-audio-processing-safely**.

Now, we will continue talking about atomic pointers and atomic references.

Using shared_ptr in a multithreaded environment

What about the `std::shared_ptr`? Can it be used in a multithreaded environment, and how is reference counting handled when multiple threads are accessing an object referenced by multiple shared pointers?

To understand shared pointers and thread safety, we need to recall how `std::shared_ptr` is typically implemented (see also *Chapter 7*, *Memory Management*). Consider the following code:

```
// Thread 1
auto p1 = std::make_shared<int>(42);
```

The code creates an `int` on the heap and a reference-counted smart pointer pointing at the `int` object. When creating the shared pointer with `std::make_shared()`, a `control block` is created next to the `int`. The control block contains, among other things, a variable for the reference count, which is incremented whenever a new pointer to the `int` is created and decremented whenever a pointer to the `int` is destroyed. To summarize, when the preceding code line is executed, three separate entities are created:

- The actual `std::shared_ptr` object `p1` (local variable on the stack)

- A control block (heap object)
- An `int` (heap object)

The following figure shows the three objects:

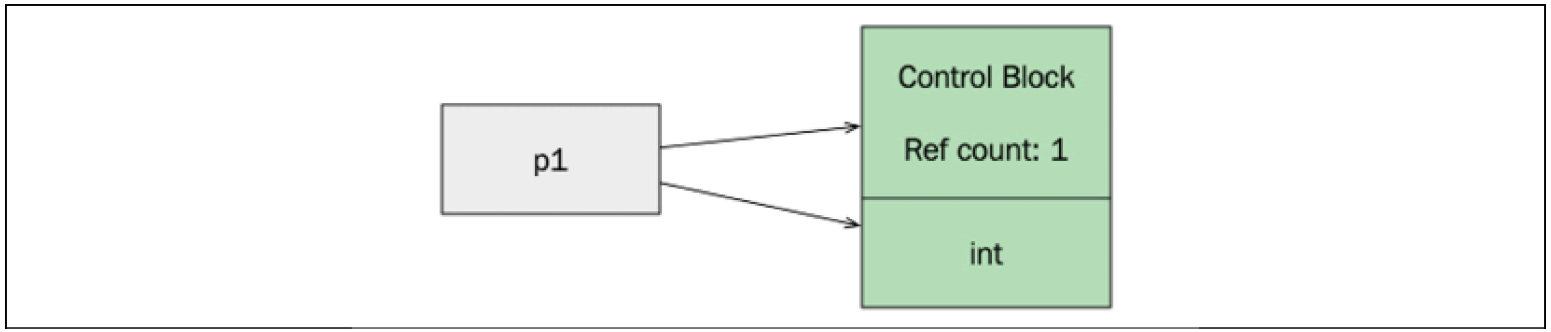

Figure 11.12: A shared_ptr instance, p1, that points to the integer object and a control block that contains the reference counting. In this case, there is only one shared pointer using the int, and hence the ref count is 1.

Now, consider what would happen if the following code was executed by a second thread:

```
// Thread 2
auto p2 = p1;
```

We are creating a new pointer pointing at the `int` (and the control block). When creating the `p2` pointer, we read `p1`, but we also need to mutate the control block when updating the reference counter. The control block lives on the heap and is shared among the two threads, so it needs synchronization to avoid data races. Since the control block is an implementation detail hidden behind the `std::shared_ptr` inter-

face, there is no way for us to know how to protect it, and it turns out that it has already been taken care of by the implementation.

Typically, it would use a mutable atomic counter. In other words, the ref counter update is thread-safe so that we can use multiple shared pointers from different threads without worrying about synchronizing the ref counter. This is a good practice and something to think about when designing classes. If you are mutating variables in methods that appear to be semantically read-only (`const`) from the client's perspective, you should make the mutating variables thread-safe. On the other hand, everything that can be detected by the client as mutating functions should be left to the client of the class to synchronize.

The following figure shows two `std::shared_ptrs` , `p1` and `p2` , that have access to the same object. The `int` is the shared object and the control block is an internally shared object between the `std::shared_ptr` instances. The control block is thread-safe by default:

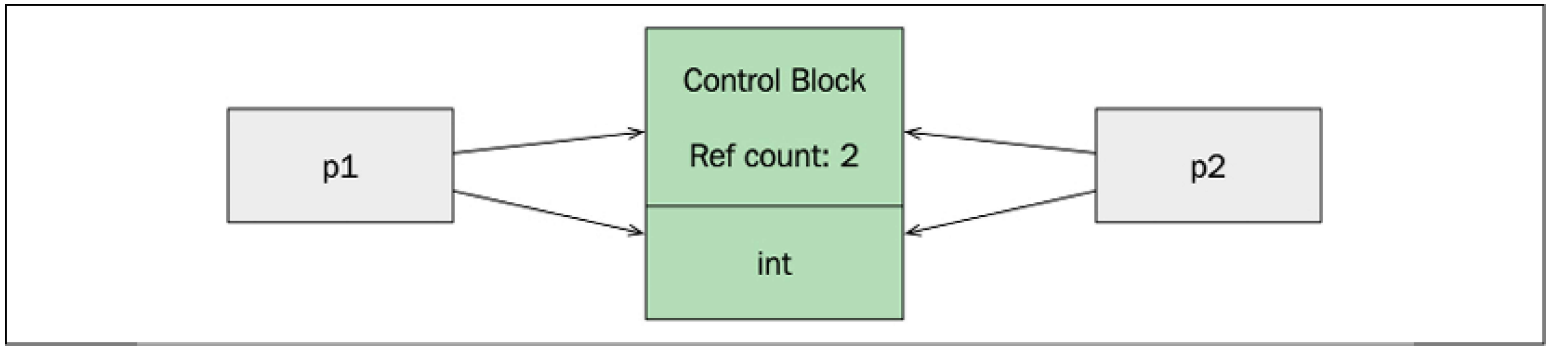

Figure 11.13: Two shared_ptrs accessing the same object

To summarize:

- The shared object, the `int` in this example, is not thread-safe and needs explicit locking if it is accessed from multiple threads.
- The control block is already thread-safe, so the reference counting mechanism works in multi-threaded environments.

Let's move on to protecting the `shared_ptr` instance.

Protecting the shared_ptr instance

Now there is only one part remaining: what about the actual `std::shared_ptr` objects, `p1` and `p2`, in the previous example? To understand this, let's turn to an example using only one global `std::shared_ptr` object called `p`:

```
// Global, how to protect?
auto p = std::shared_ptr<int>{};
```

How can we mutate `p` from multiple threads without introducing a data race? One option is to protect `p` with an explicit mutex whenever we use `p`. Or, we could use a template specialization of `std::atomic` for `std::shared_ptr` (introduced in C++20). In other words, it's possible to declare `p` as an atomic shared pointer like this:

```
// Global, protect using atomic
auto p = std::atomic<std::shared_ptr<int>>{};
```

This template specialization may or may not be lock-free. You can verify this with the `is_lock_free()` member function. Another thing to note is that the specialization `std::atomic<std::shared_ptr<T>>` is an exception to the rule that `std::atomic` can only be specialized with types that are trivially copyable. Regardless, we are glad to finally have this useful type in the standard library.

The following example demonstrates how to load and store a shared pointer object atomically from multiple threads:

```
// Thread T1 calls this function
auto f1() {
  auto new_p = std::make_shared<int>(std::rand());  // ...
  p.store(new_p);
}

// Thread T2 calls this function
auto f2() {
  auto local_p = p.load();
  // Use local_p...
}
```

In the preceding example, we assume that there are two threads, `T1` and `T2`, that call functions `f1()` and `f2()`, respectively. New heap-allocated `int` objects are created from the thread `T1` with the call to `std::make_shared<int>()`.

There is one subtle detail to consider in this example: in which thread is the heap-allocated `int` deleted? When `local_p` goes out of scope in the `f2()` function, it might be the last reference to the `int` (the reference count reaches zero). In that case, the deletion of the heap-allocated `int` will happen from thread

`T2`. Otherwise, the deletion will happen from thread `T1` when `std::atomic_store()` is called. So, the answer is that the deletion of the `int` can happen from both threads.

Atomic references

So far you have seen `std::atomc_flag` and `std::atomic<>` with numerous useful specializations. `std::atomic` can be specialized with pointers such as `std::atomic<T*>`, but you haven't seen how to use atomics with reference types. It's not possible to write `std::atomic<T&>`; instead, the standard library provides us with a template called `std::atomic_ref`.

The template `std::atomic_ref` was introduced in C++20. Its interface is identical to `std::atomic` and the reason for having a separate name is to avoid the risk of impacting existing generic code that uses `std::atomic<T>`.

An atomic reference allows us to perform atomic operations on a non-atomic object that we have a reference to. This can be convenient when we reference objects provided by a client or some third-party code that doesn't provide internally synchronized objects. We will look at an example to demonstrate the usefulness of atomic references.

Example: Using atomic references

Assume that we are writing a function that flips a coin a specified number of times:

```
void flip_coin(std::size_t n, Stats& outcomes);
```

The outcomes are accumulated in the `outcomes` object of type `Stats`, which looks like this:

```
struct Stats {
  int heads_{};
  int tails_{};
};
std::ostream& operator<<(std::ostream& os, const Stats &s) {
  os << "heads: " << s.heads_ << ", tails: " << s.tails_;
  return os;
}
```

A client can call `flip_coins()` multiple times using the same `Stats` instance, and the outcomes of the flipping are added to the `Stats`:

```
auto outcomes = Stats{};
flip_coin(30, outcomes);
flip_coin(10, outcomes);
```

Let's say we want to parallelize the implementation of `flip_coin()` and have multiple threads mutate the `Stats` object. In addition, we can assume the following:

- The `Stats` struct cannot be changed (maybe it's from a third-party library).
- We want the client to be unaware of the fact that our utility function `flip_coin()` is concurrent; that is, the concurrency of the `flip_coin()` function should be completely *transparent to the caller*.

For this example, we will reuse our previously defined function for generating random numbers:

```
int random_int(int min, int max); // See implementation above
```

Now we are ready to define our `flip_coin()` function, which will use two threads to flip a coin `n` number of times:

```
void flip_coin(std::size_t n, Stats &outcomes) {
  auto flip = [&outcomes](auto n) {
    auto heads = std::atomic_ref<int>{outcomes.heads_};
    auto tails = std::atomic_ref<int>{outcomes.tails_};
    for (auto i = 0u; i < n; ++i) {
      random_int(0, 1) == 0 ? ++heads : ++tails;
    }
  };
  auto t1 = std::jthread{flip, n / 2};      // First half
  auto t2 = std::jthread{flip, n - (n / 2)}; // The rest
}
```

Both threads will update the non-atomic outcome object whenever they have tossed a coin. Instead of using a `std::mutex`, we will create two `std::atomic_ref<int>` variables that atomically update the members of the outcome object. It is important to remember that in order to protect the heads and tails counters from data races, all concurrent accesses to the counters need to be protected using `std::atomic_ref`.

The following small program demonstrates that the `flip_coin()` function can be called without any knowledge about the concurrent implementation of `flip_coin()`:

```
int main() {
  auto stats = Stats{};
  flip_coin(5000, stats);      // Flip 5000 times
  std::cout << stats << '\n';
  assert((stats.tails_ + stats.heads_) == 5000);
}
```

Running this program on my machine produced the following output:

```
heads: 2592, tails: 2408
```

This example concludes our section about the various atomic class templates in C++. Atomics have been part of the standard library since C++11 and have continued to evolve. C++20 introduced:

- The specialization `std::atomic<std::shared_ptr<T>>`
- Atomic references; that is, the `std::atomic_ref<T>` template
- The wait and notify API, which is a lightweight alternative to using condition variables

We will now move on to discuss the C++ memory model and how it relates to atomics and concurrent programming.

The C++ memory model

Why are we talking about the memory model of C++ in a chapter about concurrency? The memory model is closely related to concurrency since it defines how the reads and writes to the memory should be visi-

ble among threads. This is a rather complicated subject that touches on both compiler optimizations and multicore computer architecture. The good news, though, is that if your program is free from data races and you use the memory order that the atomics library provides by default, your concurrent program will behave according to an intuitive memory model that is easy to understand. Still, it is important to at least have an understanding of what the memory model is and what the default memory order guarantees.

The concepts covered in this section are thoroughly explained by Herb Sutter in his talks *Atomic Weapons: The C++ Memory Model and Modern Hardware 1 & 2*. The talks are freely available at **https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/** and are highly recommended if you need more depth on this subject.

Instruction reordering

To understand the importance of the memory model, you first need some background about how the programs we write are actually executed.

When we write and run a program, it would be reasonable to assume that the instructions in the source code will be executed in the same order as they appear in the source code. This is not true. The code we write will be optimized in multiple stages before it is finally executed. Both the compiler and the hardware will reorder instructions with the goal of executing the program more efficiently. This is not new technology: compilers have done this for a long time, and this is one reason why an optimized build runs faster than a non-optimized build. The compiler (and hardware) are free to reorder instructions as long as the reordering is not observable when running the program. The program runs *as if* everything happens in program order.

Let's look at an example code snippet:

```
int a = 10;     // 1
std::cout << a; // 2
int b = a;      // 3
std::cout << b; // 4
// Observed output: 1010
```

Here, it is obvious that line number two and line number three could be swapped without introducing any observable effect:

```
int a = 10;     // 1
int b = a;      // 3 This line moved up
std::cout << a; // 2 This line moved down
std::cout << b; // 4
// Observed output: 1010
```

Here is another example, which is similar, but not identical, to the example from *Chapter 4*, *Data Structures*, where the compiler can optimize a cache-unfriendly version when iterating over a two-dimensional matrix:

```
constexpr auto ksize = size_t{100};
using MatrixType = std::array<std::array<int, ksize>, ksize>;

auto cache_thrashing(MatrixType& matrix, int v) { // 1
  for (size_t i = 0; i < ksize; ++i)          // 2
    for (size_t j = 0; j < ksize; ++j)        // 3
```

```
    matrix[j][i] = v;                   // 4
}
```

You saw in *Chapter 4*, *Data Structures*, that code similar to this produces a lot of cache misses, which hurts performance. A compiler is free to optimize this by reordering the `for` statements, like this:

```
auto cache_thrashing(MatrixType& matrix, int v) { // 1
  for (size_t j = 0; j < ksize; ++j)            // 3 Line moved up
    for (size_t i = 0; i < ksize; ++i)          // 2 Line moved down
      matrix[j][i] = v;                         // 4
}
```

There is no way to observe the difference between the two versions when executing the program, but the latter will run faster.

Optimizations performed by the compiler and the hardware (including instruction pipelining, branch prediction, and cache hierarchies) are very complicated and constantly evolving technologies. Fortunately, all these transformations of the original program can be seen as re-orderings of reads and writes in the source code. This also means that it doesn't matter whether it is the compiler or some part of the hardware that performs the transformations. The important thing for C++ programmers to know is that the instructions can be re-ordered but without any observable effect.

If you have been trying to debug an optimized build of your program, you have probably noticed that it can be hard to step through it because of the re-orderings. So, by using a debugger, the re-orderings are in some sense observable, but they are not observable when running the program in a normal way.

Atomics and memory orders

When writing single-threaded programs in C++, there is no risk of data races occurring. We can write our programs happily without being aware of instruction re-orderings. However, when it comes to shared variables in multi-threaded programs, it is a completely different story. The compiler (and hardware) does all its optimizations based on what is true and observable for *one* thread only. The compiler cannot know what other threads are able to observe through shared variables, so it is our job as programmers to inform the compiler of what re-orderings are allowed. In fact, that is exactly what we are doing when we are using an atomic variable or a mutex to protect us from data races.

When protecting a critical section with a mutex, it is guaranteed that only the thread that currently owns the lock can execute the critical section. But, the mutex is also creating memory fences around the critical section to inform the system that certain re-orderings are not allowed at the critical section boundaries. When acquiring the lock, an `acquire` fence is added, and when releasing the lock, a `release` fence is added.

I will demonstrate this with an example. Imagine that we have four instructions: **i1**, **i2**, **i3**, and **i4**. There is no dependency between each one, so the system could reorder the instructions arbitrarily without any observable effect. The instructions i2 and i3 are using shared data and are, therefore, critical sections that needs to be protected by a mutex. After adding the `acquire` and `release` of the mutex lock, there are now some re-orderings that are no longer valid. Obviously, we cannot move the instructions that are part of the critical section outside of the critical section, or they will no longer be protected by the mutex. The one-way fences ensure that no instructions can be moved out from the critical section. The i1 instruction could be moved inside the critical section by passing the acquire fence, but not beyond the release fence. The i4 instruction could also be moved inside the critical section by passing the release fence, but not beyond the acquire fence.

The following figure shows how one-way fences limit the reordering of instructions. No read or write instructions can pass above the acquire fence, and nothing can pass below the release fence:

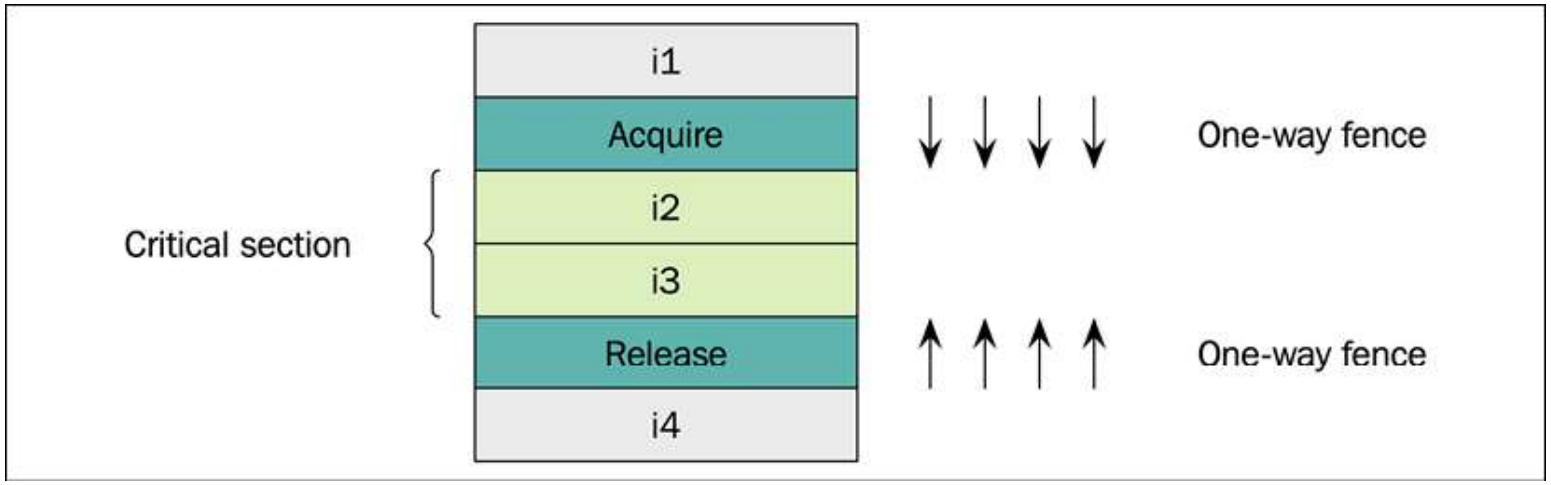

Figure 11.14: One-way fences limit the reordering of the instructions

When acquiring a mutex, we are creating an acquire memory fence. It tells the system that no memory accesses (reads or writes) can be moved above the line where the acquire fence is located. It is possible for the system to move the i4 instruction above the release fence beyond the i3 and i2 instructions, but no further than that because of the acquire fence.

Now, let's have a look at atomic variables instead of mutexes. When we use a shared atomic variable in our program, it gives us two things:

- **Protection against torn writes**: The atomic variable is always updated atomically so there is no way a reader can read a partially written value.

- **Synchronization of memory by adding sufficient memory fences**: This prevents certain instruction re-orderings to guarantee a certain memory order specified by the atomic operations.

The C++ memory model guarantees **sequential consistency** if our program is free from data races and we use the default memory order when using atomics. So, what is sequential consistency? Sequential consistency guarantees that the result of the execution is the same as if the operations were executed in the order specified by the original program. The interleaving of instructions among threads is arbitrary; that is, we have no control over the scheduling of the threads. This may sound complicated at first, but it is probably the way you already think about how a concurrent program is executed.

The downside with sequential consistency is that it can hurt performance. It is, therefore, possible to use atomics with a relaxed memory model instead. This means that you only get the protection against torn writes, but not the memory order guarantees provided by sequential consistency.

I strongly advise you against using anything else except the default sequential consistency memory order, unless you have a very thorough understanding of the effects a weaker memory model can introduce.

We will not discuss relaxed memory order any further here because it is beyond the scope of this book. But as a side note, you may be interested to know that the reference counter in a `std::shared_ptr` uses a relaxed model when incrementing the counter (but not when decrementing the counter). This is the reason why the `std::shared_ptr` member function `use_count()` only reports the approximate number of actual references when it is used in a multi-threaded environment.

One area where the memory model and atomics are highly relevant is lock-free programming. The following section will give you a taste of what lock-free programming is and some of its applications.

Lock-free programming

Lock-free programming is hard. We will not spend a lot of time discussing lock-free programming in this book, but instead I will provide you with an example of how a very simple lock-free data structure could be implemented. There is a great wealth of resources — on the web and in books (such as the Anthony Williams book mentioned earlier) — dedicated to lock-free programming that will explain the concepts you need to understand before writing your own lock-free data structures. Some concepts you might have heard of, such as **compare-and-swap** (**CAS**) and the ABA problem, will not be further discussed in this book.

Example: A lock-free queue

Here, you are going to see an example of a lock-free queue, which is a relatively simple but useful lock-free data structure. Lock-free queues can be used for one-way communication with threads that cannot use locks to synchronize access to shared data.

Its implementation is straightforward because of the limited requirements: it only supports *one reader* thread and *one writer* thread. The capacity of the queue is also fixed and cannot change during runtime.

A lock-free queue is an example of a component that might be used in environments where exceptions are typically abandoned. The queue that follows is therefore designed without exceptions, which makes the API differ from other examples in this book.

The class template `LockFreeQueue<T>` has the following public interface:

- `push()` : Adds an element to the queue and returns `true` on success. This function must only be called by the (one and only) *writer thread*. To avoid unnecessary copying when the client provides an rvalue, `push()` overloads on `const T&` and `T&&`. This technique was also used in the `BoundedBuffer` class presented earlier in this chapter.
- `pop()` : Returns an `std::optional<T>` with the front element of the queue unless the queue is empty. This function must only be called by the (one and only) *reader thread*.
- `size()` : Returns the current size of the queue. This function can be called by *both threads* concurrently.

The following is the complete implementation of the queue:

```
template <class T, size_t N>
class LockFreeQueue {
  std::array<T, N> buffer_{};   // Used by both threads
  std::atomic<size_t> size_{0}; // Used by both threads
  size_t read_pos_{0};          // Used by reader thread
  size_t write_pos_{0};         // Used by writer thread
  static_assert(std::atomic<size_t>::is_always_lock_free);
  bool do_push(auto&& t) {      // Helper function
    if (size_.load() == N) {
      return false;
    }
    buffer_[write_pos_] = std::forward<decltype(t)>(t);
    write_pos_ = (write_pos_ + 1) % N;
    size_.fetch_add(1);
    return true;
  }
public:
```

```
  // Writer thread
  bool push(T&& t) { return do_push(std::move(t)); }
  bool push(const T& t) { return do_push(t); }
  // Reader thread
  auto pop() -> std::optional<T> {
    auto val = std::optional<T>{};
    if (size_.load() > 0) {
      val = std::move(buffer_[read_pos_]);
      read_pos_ = (read_pos_ + 1) % N;
      size_.fetch_sub(1);
    }
    return val;
  }
  // Both threads can call size()
  auto size() const noexcept { return size_.load(); }
};
```

The only data member that needs atomic access is the `size_` variable. The `read_pos_` member is only used by the reader thread, and the `write_pos_` is only used by the writer thread. So what about the buffer of type `std::array`? It is mutable and accessed by both threads? Doesn't that require synchronization? Since the algorithm ensures that the two threads are never accessing the same element in the array concurrently, C++ guarantees that individual elements in an array can be accessed without data races. It doesn't matter how small the elements are; even a `char` array holds this guarantee.

When can a non-blocking queue like this be useful? One example is in audio programming, when there is a UI running on the main thread that needs to send or receive data from a real-time audio thread, which cannot block under any circumstances. The real-time thread cannot use mutex locks, allocate/free mem-

ory, or do anything else that may cause the thread to wait on threads with lower priority. Lock-free data structures are required for scenarios like these.

Both the reader and the writer are lock-free in `LockFreeQueue`, so we could have two instances of the queue to communicate in both directions between the main thread and the audio thread, as the following figure demonstrates:

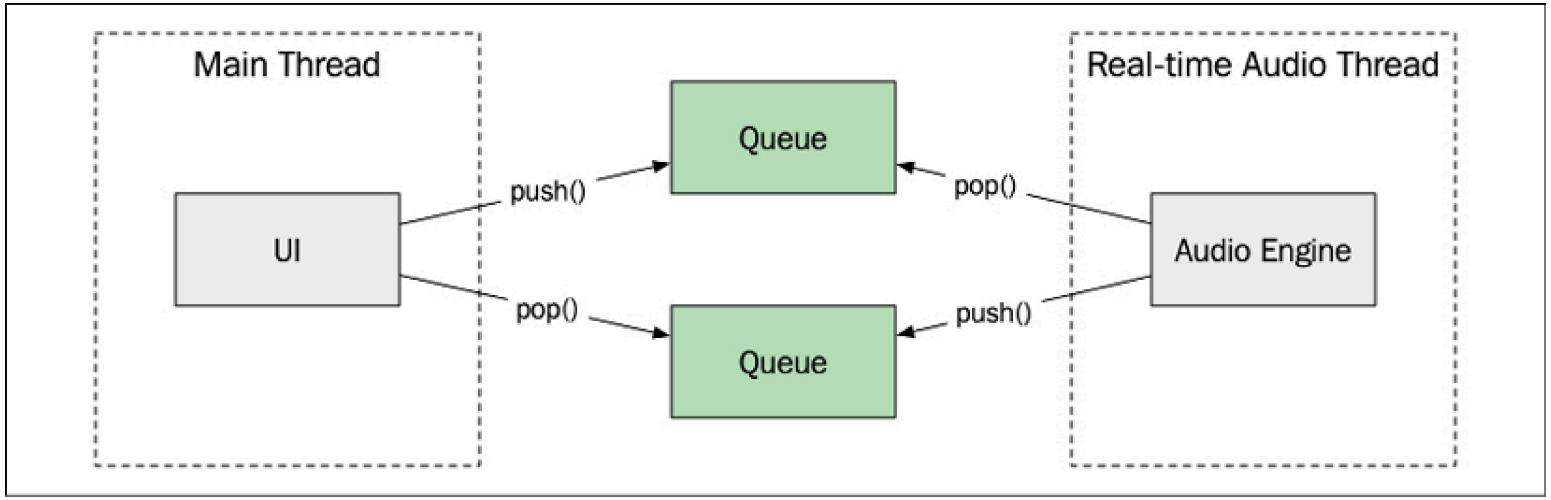

Figure 11.15: Using two lock-free queues to pass state between the main thread and a real-time audio thread

As already mentioned, this book only scratches the surface of lock-free programming. It's time to end this chapter now with a few guidelines on performance when writing concurrent programs.

Performance guidelines

I cannot stress enough the importance of having a concurrent program running *correctly* before trying to improve the performance. Also, before applying any of these guidelines related to performance, you first need to set up a reliable way of measuring what you are trying to improve.

Avoid contention

Whenever multiple threads are using shared data, there will be contention. Contention hurts performance and sometimes the overhead caused by contention can make a parallel algorithm work slower than a single-threaded alternative.

Using a lock that causes a wait and a context switch is an obvious performance penalty, but what is not equally obvious is that both locks and atomics disable optimizations in the code generated by the compiler, and they do so at runtime when the CPU executes the code. This is necessary in order to guarantee sequential consistency. But remember, the solution to such problems is never to ignore synchronization and therefore introduce data races. Data races mean undefined behavior, and having a fast but incorrect program makes nobody happy.

Instead, we need to minimize the time spent in critical sections. We can do that by entering a critical section less often, and by minimizing the critical section itself so that once we are in it, we leave it as soon as possible.

Avoid blocking operations

To write a modern responsive UI application that always runs smoothly, it is absolutely necessary to never block the main thread for more than a few milliseconds. A smoothly running app updates its inter-

face 60 times per second. This means that if you are doing something that blocks the UI thread for more than 16 ms, the FPS will drop.

You can design your internal APIs in an application with this in mind. Whenever you write a function that performs I/O or something else that might take more than a few milliseconds, it needs to be implemented as an asynchronous function. This pattern has become very common in iOS and Windows, where, for example, all network APIs have become asynchronous.

Number of threads/CPU cores

The more CPU cores a machine has, the more active running threads you can have. If you manage to split a sequential CPU-bound task into a parallel version, you can gain performance by having multiple cores working on the task in parallel.

Going from a single-threaded algorithm to an algorithm that can be run by two threads can, in the best-case scenario, double the performance. But, after adding more and more threads, you will eventually reach a limit when there is no more performance gain. Adding more threads beyond that limit will actually degrade performance since the overhead caused by context switching becomes more significant the more threads you add.

I/O-intensive tasks, for example, a web crawler that will spend a lot of time waiting for network data, require a lot of threads before reaching the limit where the CPU is oversubscribed. A thread that is waiting for I/O will most likely be switched out from the CPU to make room for other threads that are ready to execute. For CPU-bound tasks, there is usually no point in using more threads than there are cores on the machine.

Controlling the total number of threads in a big program can be hard. A good way of controlling the number of threads is to use a thread pool that can be sized to match the current hardware.

In *Chapter 14*, *Parallel Algorithms*, you will see examples of how to parallelize algorithms and how to tweak the amount of concurrency based on the number of CPU cores.

Thread priorities

The priority of a thread affects how the thread is scheduled. A thread with high priority is likely to be scheduled more often than threads with lower priorities. Thread priorities are important for lowering the latency of tasks.

Threads provided by the operating system usually have priorities. There is currently no way of setting the priority on a thread with the current C++ thread APIs. However, by using `std::thread::native_handle`, you can get a handle to the underlying operating system thread and use native APIs to set priorities.

One phenomenon related to thread priorities that can hurt the performance, and should be avoided, is called **priority inversion**. It happens when a thread with high priority is waiting to acquire a lock that is currently held by a low-priority thread. Such dependencies hurt the high-priority thread, which is blocked until the next time the low-priority thread gets scheduled so that it can release the lock.

For real-time applications, this is a big problem. In practice, it means that you cannot use locks to protect any shared resources that need to be accessed by real-time threads. A thread that produces real-time audio, for example, runs with the highest possible priority, and in order to avoid priority inversion, it is not possible for the audio thread to call any functions (including `std::malloc()`) that might block and cause a context switch.

Thread affinity

Thread affinity makes it possible to give the scheduler hints about which threads could benefit from sharing the same CPU caches. In other words, this is a request to the scheduler that some threads should be executed on a particular core if possible, to minimize cache misses.

Why would you want one thread to be executed on a particular core? The answer is (again) caching. Threads that operate on the same memory could benefit from running on the same core, and hence take advantage of warm caches. For the scheduler, this is just one of many parameters to take into account when assigning a thread to a core, so this is hardly any guarantee, but again, the behavior is very different among operating systems. Thread priorities, and even utilization of all cores (to avoid overheating), are one of the requirements that need to be taken into account by a modern scheduler.

It is not possible to set thread affinity in a portable way with the current C++ APIs, but most platforms support some way of setting an affinity mask on a thread. In order to access platform-specific functionality, you need to get a handle on the native thread. The example that follows demonstrates how to set the thread affinity mask on Linux:

```
#include <pthreads> // Non-portable header
auto set_affinity(const std::thread& t, int cpu) {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(cpu, &cpuset);
  pthread_t native_thread = t.native_handle();
  pthread_set_affinity(native_thread, sizeof(cpu_set_t), &cpuset);
}
```

Note, this is not portable C++, but it is likely that you need to do some non-portable configuration of threads if you are doing performance-critical concurrency programming.

False sharing

False sharing, or destructive interference, can degrade performance very significantly. It occurs when two threads use some data (that is not logically shared between the threads) but happen to be located in the same cache line. Imagine what will happen if the two threads are executing on different cores and constantly updating the variable residing on the shared cache line. The threads will invalidate the cache line for each other, although there is no true sharing of data between the threads.

False sharing will most likely occur when using global data or dynamically allocated data that is shared between threads. An example where false sharing is likely to occur is when allocating an array that is shared between threads, but each thread is only using a single element of the array.

The solution to this problem is to pad each element in the array so that two adjacent elements cannot reside on the same cache line. Since C++17, there is a portable way of doing this using the `std::hardware_destructive_interference_size` constant defined in `<new>` in combination with the `alignas` specifier. The following example demonstrates how to create an element that prevents false sharing:

```
struct alignas(std::hardware_destructive_interference_size) Element {
  int counter_{};
};

auto elements = std::vector<Element>(num_threads);
```

The elements in the vector are now guaranteed to reside on separate cache lines.

Summary

In this chapter, you have seen how to create programs that can execute multiple threads concurrently. We also covered how to avoid data races by protecting critical sections with locks or by using atomics. You learned that C++20 comes with some useful synchronization primitives: latches, barriers, and semaphores. We then looked into execution order and the C++ memory model, which becomes important to understand when writing lock-free programs. You also discovered that immutable data structures are thread-safe. The chapter ended with some guidelines for improving performance in concurrent applications.

The next two chapters are dedicated to a completely new C++20 feature called coroutines, which allows us to write asynchronous code in a sequential style.