C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

Table of Contents

Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10 Testing and debugging multithreaded applications

This chapter covers

- Concurrency-related bugs
- Locating bugs through testing and code review
- Designing multithreaded tests
- Testing the performance of multithreaded code

Up to now, I've focused on what's involved in writing concurrent code—the tools that are available, how to use them, and the overall design and structure of the code. But there's a crucial part of software development that I haven't addressed yet: testing and debugging. If you're reading this chapter hoping for an easy way to test concurrent code, you're going to be sorely disappointed. Testing and debugging concurrent code is *hard*. What I *am* going to give you are some techniques that will make things easier, alongside issues that are important to think about.

Testing and debugging are like two sides of a coin—you subject your code to tests in order to find any bugs that might be there, and you debug it to remove those bugs. With luck, you only have to remove the bugs found by your own tests rather than bugs found by the end users of your application. Before we look at either testing or debugging, it's important to understand the problems that might arise, so let's look at those.

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10.1 Types of concurrency-related bugs

You can get just about any sort of bug in concurrent code; it's not special in that regard. But some types of bugs are directly related to the use of concurrency and therefore of particular relevance to this book. Typically, these concurrency-related bugs fall into two primary categories:

- Unwanted blocking
- Race conditions

These are huge categories, so let's divide them up a bit. First, let's look at unwanted blocking.

10.1.1 Unwanted blocking

Licensed by Pralay Patoria 2085776

What do I mean by unwanted blocking? First, a thread is *blocked* when it's unable to proceed because it's waiting for something. This is typically something like a mutex, a condition variable, or a future, but it could be waiting for I/O. This is a natural part of multithreaded code, but it's not always desirable—hence the problem of unwanted blocking. This leads us to the next question: why is this blocking unwanted? Typically, this is because some other thread is also waiting for the blocked thread to perform some action, and so that thread in turn is blocked. There are several variations on this theme:

- *Deadlock*—As you saw in chapter 3, in the case of deadlock one thread is waiting for another, which is in turn waiting for the first. If your threads deadlock, the tasks they're supposed to be doing won't get done. In the most visible cases, one of the threads involved is the thread responsible for the user interface, in which case the interface will cease to respond. In other cases, the interface will remain responsive, but some required task won't complete, such as a search not returning or a document not printing.
- *Livelock*—Livelock is similar to deadlock in that one thread is waiting for another, which is in turn waiting for the first. The key difference here is that the wait is not a blocking wait but an active checking loop, such as a spin lock. In serious cases, the symptoms are the same as deadlock (the app doesn't make any progress), except that the CPU usage is high because threads are still running but blocking each other. In not-so-serious cases, the livelock will eventually resolve because of the random scheduling, but there will be a long delay in the task that got livelocked, with a high CPU usage during that delay.
- *Blocking on I/O or other external input*—If your thread is blocked waiting for external input, it can't proceed, even if the waited-for input is never going to come. It's therefore undesirable to block on external input from a thread that also performs tasks that other threads may be waiting for.

That briefly covers unwanted blocking. What about race conditions?

Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10.1.2 Race conditions

Race conditions are the most common cause of problems in multithreaded code—many deadlocks and livelocks only actually manifest because of a race condition. Not all race conditions are problematic—a race condition occurs anytime the behavior depends on the relative scheduling of operations in separate threads. A large number of race conditions are entirely benign; for example, which worker thread processes the next task in the task queue is largely irrelevant. However, many concurrency bugs are due to race conditions. In particular, race conditions often cause the following types of problems:

- *Data races*—A data race is the specific type of race condition that results in undefined behavior because of unsynchronized concurrent access to a shared memory location. I introduced data races in chapter 5 when we looked at the C++ memory model. Data races usually occur through incorrect usage of atomic operations to synchronize threads or through access to shared data without locking the appropriate mutex.
- *Broken invariants*—These can manifest as dangling pointers (because another thread deleted the data being accessed), random memory corruption (due to a thread reading inconsistent values resulting from partial updates), and double-free (such as when two threads pop the same value from a queue, and so both delete some associated data), among others. The invariants being broken can be temporal- as well as value-based. If operations on separate threads are required to execute in a particular order, incorrect synchronization can lead to a race condition in which the required order is sometimes violated.
- *Lifetime issues*—Although you could bundle these problems in with broken invariants, this really is a separate category. The basic problem with bugs in this category is that the thread outlives the data that it accesses, so it is accessing data that has been deleted or otherwise destroyed, and potentially the storage is even reused for another object. You typically get lifetime issues where a thread references local variables that go out of scope before the thread function has completed, but they aren't limited to that scenario. Whenever the lifetime of the thread and the data it operates on aren't tied together in some way, there's the potential for the data to be destroyed before the thread has finished and for the thread function to have the rug pulled out from under its feet. If you manually call `join()` in order to wait for the thread to complete, you need to ensure that the call to `join()` can't be skipped if an exception is thrown. This is basic exception safety applied to threads.

It's the problematic race conditions that are the killers. With deadlock and livelock, the application appears to hang and become completely unresponsive or takes too long to complete a task. Often, you can attach a debugger to the running process to identify which threads are involved in the deadlock or livelock and which synchronization objects they're fighting over. With data races, broken invariants, and lifetime

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

issues, the visible symptoms of the problem (such as random crashes or incorrect output) can manifest anywhere in the code—the code may overwrite memory used by another part of the system that isn't touched until much later. The fault will then manifest in code completely unrelated to the location of the buggy code, possibly much later in the execution of the program. This is the true curse of shared memory systems—however much you try to limit which data is accessible by which thread and try to ensure that correct synchronization is used, any thread can overwrite the data being used by any other thread in the application.

Now that we've briefly identified the sorts of problems we're looking for, let's look at what you can do to locate any instances in your code so you can fix them.

10.2 Techniques for locating concurrency-related bugs

In the previous section we looked at the types of concurrency-related bugs you might see and how they might manifest in your code. With that information in mind, you can then look at your code to see where bugs might lie and how you can attempt to determine whether there are any bugs in a particular section.

Perhaps the most obvious and straightforward thing to do is *look at the code*. Although this might seem obvious, it's actually difficult to do in a thorough way. When you read code you've just written, it's all too easy to read what you intended to write rather than what's actually there. Likewise, when reviewing code that others have written, it's tempting to just give it a quick read-through, check it off against your local coding standards, and highlight any glaringly obvious problems. What's needed is to spend the time really going through the code with a fine-tooth comb, thinking about the concurrency issues—and the non-concurrency issues as well. (You might as well, while you're doing it. After all, a bug is a bug.) We'll cover specific things to think about when reviewing code shortly.

Even after thoroughly reviewing your code, you still might have missed some bugs, and in any case you need to confirm that it does indeed work, for peace of mind if nothing else. Consequently, we'll continue on from reviewing the code to a few techniques to employ when testing multithreaded code.

10.2.1 Reviewing code to locate potential bugs

As I've already mentioned, when reviewing multithreaded code to check for concurrency-related bugs, it's important to review it thoroughly, with a fine-tooth comb. If possible, get someone else to review it. Because they haven't written the code, they'll have to think through how it works, and this will help uncover any bugs that may be there. It's important that the reviewer have the time to do the review properly—not a casual two-minute quick glance, but a proper, considered review. Most concurrency bugs require more than a quick glance to spot—they usually rely on subtle timing issues to actually manifest.

If you get one of your colleagues to review the code, they'll be coming at it fresh. They'll therefore see things from a different point of view and may well spot things

Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

that you don't. If you don't have colleagues you can ask, ask a friend, or even post the code on the internet (taking care not to upset your company lawyers). If you can't get anybody to review your code for you, or they don't find anything, don't worry—there's still more you can do. For starters, it might be worth leaving the code alone for a while—work on another part of the application, read a book, or go for a walk. If you take a break, your subconscious can work on the problem in the background while you're consciously focused on something else. Also, the code will be less familiar when you come back to it—you might manage to look at it from a different perspective yourself.

An alternative to getting someone else to review your code is to do it yourself. One useful technique is to try to explain how it works *in detail* to someone else. They don't even have to be physically there—many teams have a bear or rubber chicken for this purpose, and I personally find that writing detailed notes can be hugely beneficial. As you explain, think about each line, what could happen, which data it accesses, and so forth. Ask yourself questions about the code, and explain the answers. I find this to be an incredibly powerful technique—by asking myself these questions and thinking carefully about the answers, the problem often reveals itself. These questions can be helpful for *any* code review, not just when reviewing your own code.

QUESTIONS TO THINK ABOUT WHEN REVIEWING MULTITHREADED CODE

As I've already mentioned, it can be useful for a reviewer (whether the code's author or someone else) to think about specific questions relating to the code being reviewed. These questions can focus the reviewer's mind on the relevant details of the code and can help identify potential problems. The questions I like to ask include the following, though this is most definitely not a comprehensive list. You might find other questions that help you to focus better. Anyway, here are the questions:

- Which data needs to be protected from concurrent access?
- How do you ensure that the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this thread hold?
- Which mutexes might other threads hold?
- Are there any ordering requirements between the operations done in this thread and those done in another? How are those requirements enforced?
- Is the data loaded by this thread still valid? Could it have been modified by other threads?
- If you assume that another thread could be modifying the data, what would that mean and how could you ensure that this never happens?

This last question is my favorite, because it really makes me think about the relationships between the threads. By assuming the existence of a bug related to a particular line of code, you can then act as a detective and track down the cause. In order to convince yourself that there's no bug, you have to consider every corner case and possible ordering. This is particularly useful where the data is protected by more than one mutex over its lifetime, such as with the thread-safe queue from chapter 6 where we

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

had separate mutexes for the head and tail of the queue: in order to be sure that an access is safe while holding one mutex, you have to be certain that a thread holding the *other* mutex can't also access the same element. It also makes it obvious that public data, or data for which other code can readily obtain a pointer or reference, has to come under particular scrutiny.

The penultimate question in the list is also important, because it addresses what's an easy mistake to make: if you release and then reacquire a mutex, you must assume that other threads may have modified the shared data. Although this is obvious, if the mutex locks aren't immediately visible—perhaps because they're internal to an object—you may unwittingly be doing exactly that. In chapter 6 you saw how this can lead to race conditions and bugs where the functions provided on a thread-safe data structure are too fine-grained. Whereas for a non-thread-safe stack it makes sense to have separate `top()` and `pop()` operations, for a stack that may be accessed by multiple threads concurrently, this is no longer the case because the lock on the internal mutex is released between the two calls, and so another thread can modify the stack. As you saw in chapter 6, the solution is to combine the two operations so they are both performed under the protection of the same mutex lock, thus eliminating the potential race condition.

OK, so you've reviewed your code (or got someone else to review it). You're sure there are no bugs. The proof of the pudding is, as they say, in the eating—how can you test your code to confirm or deny your belief in its lack of bugs?

10.2.2 Locating concurrency-related bugs by testing

When developing single-threaded applications, testing your applications is relatively straightforward, if time consuming. You could, in principle, identify all the possible sets of input data (or at least all the interesting cases) and run them through the application. If the application produced the correct behavior and output, you'd know it works for that given set of input. Testing for error states such as the handling of disk-full errors is more complicated than that, but the idea is the same—set up the initial conditions and allow the application to run.

Testing multithreaded code is an order of magnitude harder, because the precise scheduling of the threads is indeterminate and may vary from run to run. Consequently, even if you run the application with the same input data, it might work correctly some times and fail at other times if there's a race condition lurking in the code. Just because there's a potential race condition doesn't mean the code will fail *always*, just that it *might* fail *sometimes*.

Given the inherent difficulty of reproducing concurrency-related bugs, it pays to design your tests carefully. You want each test to run the smallest amount of code that could potentially demonstrate a problem, so that you can best isolate the code that's faulty if the test fails—it's better to test a concurrent queue directly to verify that concurrent pushes and pops work rather than testing it through a whole chunk of code that uses the queue. It can help if you think about how code should be tested when designing it—see the section on designing for testability later in this chapter.

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

It's also worth eliminating the concurrency from the test in order to verify that the problem is concurrency-related. If you have a problem when everything is running in a single thread, it's just a plain common or garden-variety bug rather than a concurrency-related bug. This is particularly important when trying to track down a bug that occurs "in the wild" as opposed to being detected in your test harness. Just because a bug occurs in the multithreaded portion of your application doesn't mean it's automatically concurrency-related. If you're using thread pools to manage the level of concurrency, there's usually a configuration parameter you can set to specify the number of worker threads. If you're managing threads manually, you'll have to modify the code to use a single thread for the test. Either way, if you can reduce your application to a single thread, you can eliminate concurrency as a cause. On the flip side, if the problem goes away on a *single-core* system (even with multiple threads running) but is present on *multicore* systems or *multiprocessor* systems, you have a race condition and possibly a synchronization or memory-ordering issue.

There's more to testing concurrent code than the structure of the code being tested; the structure of the test is just as important, as is the test environment. If you continue on with the example of testing a concurrent queue, you have to think about various scenarios:

- One thread calling `push()` or `pop()` on its own to verify that the queue does work at a basic level
- One thread calling `push()` on an empty queue while another thread calls `pop()`
- Multiple threads calling `push()` on an empty queue
- Multiple threads calling `push()` on a full queue
- Multiple threads calling `pop()` on an empty queue
- Multiple threads calling `pop()` on a full queue
- Multiple threads calling `pop()` on a partially full queue with insufficient items for all threads
- Multiple threads calling `push()` while one thread calls `pop()` on an empty queue
- Multiple threads calling `push()` while one thread calls `pop()` on a full queue
- Multiple threads calling `push()` while multiple threads call `pop()` on an empty queue
- Multiple threads calling `push()` while multiple threads call `pop()` on a full queue

Having thought about all these scenarios and more, you then need to consider additional factors about the test environment:

- What you mean by "multiple threads" in each case (3, 4, 1024?)
- Whether there are enough processing cores in the system for each thread to run on its own core
- Which processor architectures the tests should be run on
- How you ensure suitable scheduling for the "while" parts of your tests

Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

There are additional factors to think about specific to your particular situation. Of these four environmental considerations, the first and last affect the structure of the test itself (and are covered in section 10.2.5), whereas the other two are related to the physical test system being used. The number of threads to use relates to the particular code being tested, but there are various ways of structuring tests to obtain suitable scheduling. Before we look at these techniques, let's look at how you can design your application code to be easier to test.

10.2.3 Designing for testability

Testing multithreaded code is difficult, so you want to do what you can to make it easier. One of the most important things you can do is *design* the code for testability. A lot has been written about designing single-threaded code for testability, and much of the advice still applies. In general, code is easier to test if the following factors apply:

- The responsibilities of each function and class are clear.
- The functions are short and to the point.
- Your tests can take complete control of the environment surrounding the code being tested.
- The code that performs the particular operation being tested is close together rather than spread throughout the system.
- You thought about how to test the code before you wrote it.

All of these are still true for multithreaded code. In fact, I'd argue that it's even more important to pay attention to the testability of multithreaded code than for single-threaded code, because it's inherently that much harder to test. That last point is important: even if you don't go as far as writing your tests before the code, it's well worth thinking about how you can test the code before you write it—what inputs to use, which conditions are likely to be problematic, how to stimulate the code in potentially problematic ways, and so on.

One of the best ways to design concurrent code for testing is to eliminate the concurrency. If you can break down the code into those parts that are responsible for the communication paths between threads and those parts that operate on the communicated data within a single thread, then you've greatly reduced the problem. Those parts of the application that operate on data that's being accessed by only that one thread can then be tested using the normal single-threaded techniques. The hard-to-test concurrent code that deals with communicating between threads and ensuring that only one thread at a time *is* accessing a particular block of data is now much smaller and the testing more tractable.

For example, if your application is designed as a multithreaded state machine, you could split it into several parts. The state logic for each thread, which ensures that the transitions and operations are correct for each possible set of input events, can be tested independently with single-threaded techniques, with the test harness providing the input events that would be coming from other threads. Then, the core state

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

machine and message routing code that ensures that events are correctly delivered to the right thread in the right order can be tested independently, but with multiple concurrent threads and simple state logic designed specifically for the tests.

Alternatively, if you can divide your code into multiple blocks of *read shared data/transform data/update shared data*, you can test the *transform data* portions using all the usual single-threaded techniques, because this is now just single-threaded code. The hard problem of testing a multithreaded transformation will be reduced to testing the reading and updating of the shared data, which is much simpler.

One thing to watch out for is that library calls can use internal variables to store state, which then becomes shared if multiple threads use the same set of library calls. This can be a problem because it's not immediately apparent that the code accesses shared data. However, with time you learn which library calls these are, and they stick out like sore thumbs. You can then either add appropriate protection and synchronization or use an alternate function that's safe for concurrent access from multiple threads.

There's more to designing multithreaded code for testability than structuring your code to minimize the amount of code that needs to deal with concurrency-related issues and paying attention to the use of non-thread-safe library calls. It's also helpful to bear in mind the same set of questions you ask yourself when reviewing the code, from section 10.2.1. Although these questions aren't directly about testing and testability, if you think about the issues with your "testing hat" on and consider how to test the code, it will affect which design choices you make and will make testing easier.

Now that we've looked at designing code to make testing easier, and potentially modified the code to separate the "concurrent" parts (such as the thread-safe containers or state machine event logic) from the "single-threaded" parts (which may still interact with other threads through the concurrent chunks), let's look at the techniques for testing concurrency-aware code.

10.2.4 Multithreaded testing techniques

So, you've thought through the scenario you wish to test and written a small amount of code that exercises the functions being tested. How do you ensure that any potentially problematic scheduling sequences are exercised in order to flush out the bugs?

Well, there are a few ways of approaching this, starting with brute-force testing, or stress testing.

BRUTE-FORCE TESTING

The idea behind brute-force testing is to stress the code to see if it breaks. This typically means running the code many times, possibly with many threads running at once. If there's a bug that manifests only when the threads are scheduled in a particular fashion, then the more times the code is run, the more likely the bug is to appear. If you run the test once and it passes, you might feel a bit of confidence that the code works. If you run it ten times in a row and it passes every time, you'll likely feel more confident. If you run the test a billion times and it passes every time, you'll feel more confident still.

Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The confidence you have in the results does depend on the amount of code being tested by each test. If your tests are quite fine-grained, like the tests outlined previously for a thread-safe queue, such brute-force testing can give you a high degree of confidence in your code. On the other hand, if the code being tested is considerably larger, the number of possible scheduling permutations is so vast that even a billion test runs might yield a low level of confidence.

The downside to brute-force testing is that it might give you false confidence. If the way you've written the test means that the problematic circumstances can't occur, you can run the test as many times as you like and it won't fail, even if it would fail *every* time in slightly different circumstances. The worst example is where the problematic circumstances can't occur on your test system because of the way the particular system you're testing on happens to run. Unless your code is to run only on systems identical to the one being tested, the particular hardware and operating system combination may not allow the circumstances that would cause a problem to arise.

The classic example here is testing a multithreaded application on a single-processor system. Because every thread has to run on the same processor, everything is automatically serialized, and many race conditions and cache ping-pong problems that you may get with a true multiprocessor system evaporate. This isn't the only variable though; different processor architectures provide different synchronization and ordering facilities. For example, on x86 and x86-64 architectures, atomic load operations are always the same, whether tagged `memory_order_relaxed` or `memory_order_seq_cst` (see section 5.3.3). This means that code written using relaxed memory ordering may work on systems with an x86 architecture, where it would fail on a system with a finer-grained set of memory-ordering instructions such as SPARC.

If you need your application to be portable across a range of target systems, it's important to test it on representative instances of those systems. This is why I listed the processor architectures being used for testing as a consideration in section 10.2.2.

Avoiding the potential for false confidence is crucial to successful brute-force testing. This requires careful thought over test design, not just with respect to the choice of unit for the code being tested but also with respect to the design of the test harness and the choice of testing environment. You need to ensure that as many of the code paths as possible are tested and as many of the possible thread interactions as feasible. Not only that, but you need to know *which* options are covered and *which are left untested*.

Although brute-force testing does give you some degree of confidence in your code, it's not guaranteed to find all the problems. There's one technique that *is* guaranteed to find the problems, if you have the time to apply it to your code and the appropriate software. I call it *combination simulation testing*.

COMBINATION SIMULATION TESTING

That's a bit of a mouthful, so I'd best explain what I mean. The idea is that you run your code with a special piece of software that *simulates* the real runtime environment of the code. You may be aware of software that allows you to run multiple virtual machines on a single physical computer, where the characteristics of the virtual machine

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

and its hardware are emulated by the supervisor software. The idea here is similar, except rather than just emulating the system, the simulation software records the sequences of data accesses, locks, and atomic operations from each thread. It then uses the rules of the C++ memory model to repeat the run with every permitted *combination* of operations and thus identify race conditions and deadlocks.

Although such exhaustive combination testing is guaranteed to find all the problems the system is designed to detect, for anything but the most trivial of programs it will take a huge amount of time, because the number of combinations increases exponentially with the number of threads and the number of operations performed by each thread. This technique is thus best reserved for fine-grained tests of individual pieces of code rather than an entire application. The other obvious downside is that it relies on the availability of simulation software that can handle the operations used in your code.

So, you have a technique that involves running your test many times under normal conditions but that might miss problems, and you have a technique that involves running your test many times under special conditions but that's more likely to find any problems that exist. Are there any other options?

A third option is to use a library that detects problems as they occur in the running of the tests.

DETECTING PROBLEMS EXPOSED BY TESTS WITH A SPECIAL LIBRARY

Although this option doesn't provide the exhaustive checking of a combination simulation test, you can identify many problems by using a special implementation of the library synchronization primitives such as mutexes, locks, and condition variables. For example, it's common to require that all accesses to a piece of shared data be done with a particular mutex locked. If you could check which mutexes were locked when the data was accessed, you could verify that the appropriate mutex was indeed locked by the calling thread when the data was accessed and report a failure if this was not the case. By marking your shared data in some way, you can allow the library to check this for you.

Such a library implementation can also record the sequence of locks if more than one mutex is held by a particular thread at once. If another thread locks the same mutexes in a different order, this could be recorded as a *potential* deadlock even if the test didn't actually deadlock while running.

Another type of special library that could be used when testing multithreaded code is one where the implementations of the threading primitives such as mutexes and condition variables give the test writer control over which thread gets the lock when multiple threads are waiting or which thread is notified by a `notify_one()` call on a condition variable. This would allow you to set up particular scenarios and verify that your code works as expected in those scenarios.

Some of these testing facilities would have to be supplied as part of the C++ Standard Library implementation, whereas others can be built on top of the Standard Library as part of your test harness.

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Having looked at various ways of executing test code, let's now look at ways of structuring the code to achieve the scheduling you want.

10.2.5 Structuring multithreaded test code

Back in section 10.2.2 I said that you need to find ways of providing suitable scheduling for the "while" part of your tests. Now it's time to look at the issues involved in that.

The basic issue is that you need to arrange for a set of threads to each be executing a chosen piece of code at a time that you specify. In the most basic case you have two threads, but this could easily be extended to more. In the first step, you need to identify the distinct parts of each test:

- The general setup code that must be executed before anything else
- The thread-specific setup code that must run on each thread
- The actual code for each thread that you desire to run concurrently
- The code to be run after the concurrent execution has finished, possibly including assertions on the state of the code

To explain further, let's consider a specific example from the test list in section 10.2.2: one thread calling `push()` on an empty queue while another thread calls `pop()`.

The *general* setup code is simple: you must create the queue. The thread executing `pop()` has no *thread-specific* setup code. The thread-specific setup code for the thread executing `push()` depends on the interface to the queue and the type of object being stored. If the object being stored is expensive to construct or must be heap allocated, you want to do this as part of the thread-specific setup, so that it doesn't affect the test. On the other hand, if the queue is just storing plain `int`s, there's nothing to be gained by constructing an `int` in the setup code. The actual code being tested is relatively straightforward—a call to `push()` from one thread and a call to `pop()` from another—but what about the "after completion" code?

In this case, it depends on what you want `pop()` to do. If it's supposed to block until there is data, then clearly you want to see that the returned data is what was supplied to the `push()` call and that the queue is empty afterward. If `pop()` is *not* blocking and may complete even when the queue is empty, you need to test for two possibilities: either the `pop()` returned the data item supplied to the `push()` and the queue is empty or the `pop()` signaled that there was no data and the queue has one element. One or the other must be true; what you want to avoid is the scenario that `pop()` signaled "no data" but the queue is empty, or that `pop()` returned the value and the queue is *still* not empty. In order to simplify the test, assume you have a blocking `pop()`. The final code is therefore an assertion that the popped value is the pushed value and that the queue is empty.

Now, having identified the various chunks of code, you need to do the best you can to ensure that everything runs as planned. One way to do this is to use a set of `std::promise`s to indicate when everything is ready. Each thread sets a promise to indicate that it's ready and then waits on a (copy of a) `std::shared_future` obtained

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

from a third `std::promise`; the main thread waits for all the promises from all the threads to be set and then triggers the threads to `go`. This ensures that each thread has started and is just before the chunk of code that should be run concurrently; any thread-specific setup should be done before setting that thread's promise. Finally, the main thread waits for the threads to complete and checks the final state. You also need to be aware of exceptions and make sure you don't have any threads left waiting for the `go` signal when that's not going to happen. The following listing shows one way of structuring this test.

Listing 10.1 An example test for concurrent `push()` and `pop()` calls on a queue

```
void test_concurrent_push_and_pop_on_empty_queue()
{
    threadsafe_queue<int> q;     ◁─❶

    std::promise<void> go,push_ready,pop_ready;      ◁─❷
    std::shared_future<void> ready(go.get_future());       ◁─❸

    std::future<void> push_done;     ◁─❹
    std::future<int> pop_done;

    try
    {
        push_done=std::async(std::launch::async,      ◁─❺
                             [&q,ready,&push_ready]()
                             {
                                 push_ready.set_value();
                                 ready.wait();
                                 q.push(42);
                             }
            );
        pop_done=std::async(std::launch::async,      ◁─❻
                            [&q,ready,&pop_ready]()
                            {
                                pop_ready.set_value();
                                ready.wait();
                                return q.pop();      ◁─❼
                            }
            );
        push_ready.get_future().wait();      ◁─❽
        pop_ready.get_future().wait();
        go.set_value();                     ◁─❾

        push_done.get();                ◁─❿
        assert(pop_done.get()==42);         ◁─⓫
        assert(q.empty());
    }
    catch(...)
    {
        go.set_value();      ◁─⓬
        throw;
    }
}
```

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The structure is pretty much as described previously. First, you create your empty queue as part of the general setup ❶. Then, you create all your promises for the “ready” signals ❷ and get a `std::shared_future` for the `go` signal ❸. Then, you create the futures you’ll use to indicate that the threads have finished ❹. These have to go outside the `try` block so that you can set the `go` signal on an exception without waiting for the test threads to complete (which would deadlock—a deadlock in the test code would be rather less than ideal).

Inside the `try` block you can then start the threads ❺, ❻—you use `std::launch::async` to guarantee that the tasks are each running on their own thread. Note that the use of `std::async` makes your exception-safety task easier than it would be with plain `std::thread` because the destructor for the future will join with the thread. The lambda captures specify that each task will reference the queue and the relevant promise for signaling readiness, while taking a copy of the `ready` future you got from the `go` promise.

As described previously, each task sets its own `ready` signal and then waits for the general `ready` signal before running the actual test code. The main thread does the reverse—waiting for the signals from both threads ❽ before signaling them to start the real test ❾.

Finally, the main thread calls `get()` on the futures from the async calls to wait for the tasks to finish ❿, ⓫ and checks the results. Note that the *pop* task returns the retrieved value through the future ❼, so you can use that to get the result for the assert ⓫.

If an exception is thrown, you set the `go` signal to avoid any chance of a dangling thread and rethrow the exception ⓬. The futures corresponding to the tasks ❹ were declared last, so they’ll be destroyed first, and their destructors will wait for the tasks to complete if they haven’t already.

Although this seems like quite a lot of boilerplate just to test two simple calls, it’s necessary to use something similar in order to have the best chance of testing what you actually want to test. For example, actually starting a thread can be quite a time-consuming process, so if you didn’t have the threads wait for the `go` signal, then the push thread may have completed before the pop thread even started, which would completely defeat the point of the test. Using the futures in this way ensures that both threads are running and blocked on the same future. Unblocking the future then allows both threads to run. Once you’re familiar with the structure, it should be relatively straightforward to create new tests in the same pattern. For tests that require more than two threads, this pattern is readily extended to additional threads.

So far, we’ve just been looking at the *correctness* of multithreaded code. Although this is the most important issue, it’s not the only reason you test: it’s also important to test the *performance* of multithreaded code, so let’s look at that next.

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10.2.6 Testing the performance of multithreaded code

One of the main reasons you might choose to use concurrency in an application is to make use of the increasing prevalence of multicore processors to improve the performance of your applications. It's therefore important to actually test your code to confirm that the performance does indeed improve, just as you'd do with any other attempt at optimization.

The particular issue with using concurrency for performance is the *scalability*—you want code that runs approximately 24 times faster or processes 24 times as much data on a 24-core machine than on a single-core machine, all else being equal. You don't want code that runs twice as fast on a dual-core machine but is actually slower on a 24-core machine. As you saw in section 8.4.2, if a significant section of your code runs on only one thread, this can limit the potential performance gain. It's therefore worth looking at the overall design of the code before you start testing, so you know whether you're hoping for a factor-of-24 improvement, or whether the serial portion of your code means you're limited to a maximum of a factor of 3.

As you've already seen in previous chapters, contention between processors for access to a data structure can have a big performance impact. Something that scales nicely with the number of processors when that number is small may actually perform badly when the number of processors is much larger because of the huge increase in contention.

Consequently, when testing for the performance of multithreaded code, it's best to check the performance on systems with as many different configurations as possible, so you get a picture of the scalability graph. At the very least, you ought to test on a single-processor system *and* a system with as many processing cores as are available to you.

10.3 Summary

In this chapter we looked at various types of concurrency-related bugs that you might encounter, from deadlocks and livelocks to data races and other problematic race conditions. We followed that with techniques for locating bugs. These included issues to think about during code reviews, guidelines for writing testable code, and how to structure tests for concurrent code. Finally, we looked at some utility components that can help with testing.

Chapter 10. Testing and debugging multithreaded applications. C++ Concurrency in Action: Practical Multithreading, ISBN: 9781933988771
Prepared for pralay.patoria@credit-suisse.com, Pralay Patoria
©2012 by Manning Publications Co. All rights reserved.. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.