1.3. Concurrency and multithreading in C++

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

1.3.1. History of multithreading in C++

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

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

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

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

1.3.2. Concurrency support in the new standard

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

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

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

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

1.3.3. Efficiency in the C++ Thread Library

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

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

Another goal of the C++ Standards Committee has been to ensure that C++ provides sufficient low-level facilities for those wishing to work close to the metal for the ultimate performance. To this end, along with the new memory model comes a comprehensive atomic operations library for direct control over individual bits and bytes and the inter-thread synchronization and visibility of any changes. These atomic types and the corresponding operations can now be used in many places where developers would previously have chosen to drop down to platform-specific assembly language. Code using the new standard types and operations is thus more portable and easier to maintain.

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

additional code anyway.

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

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

1.3.4. Platform-specific facilities

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

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