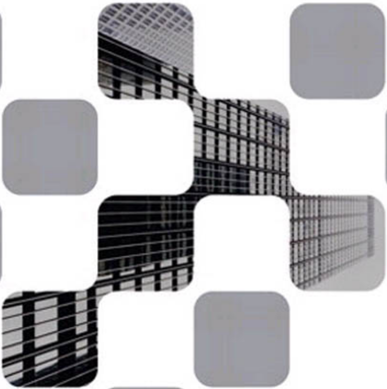
Foreword by Craig Mundie, Chief Research and Strategy Officer, Microsoft

Concurrent Programming on Windows



Microsoft .net Development Series

Joe Duffy

Library of Congress Cataloging-in-Publication Data

Duffy, Joe, 1980-
Concurrent programming on Windows / Joe Duffy.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-43482-1 (pbk. : alk. paper) 1. Parallel programming (Computer science)
2. Electronic data processing—Distributed processing. 3. Multitasking (Computer science)
4. Microsoft Windows (Computer file) I. Title.

QA76.642D84 2008
005.2'75—dc22

2008033911

ISBN-13: 978-0-321-43482-1
ISBN-10: 0-321-43482-X
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, October 2008

Foreword

THE COMPUTER INDUSTRY is once again at a crossroads. Hardware concurrency, in the form of new *manycore processors,* together with growing software complexity, will require that the technology industry fundamentally rethink both the architecture of modern computers and the resulting software development paradigms.

For the past few decades, the computer has progressed comfortably along the path of exponential performance and capacity growth without any fundamental changes in the underlying computation model. Hardware followed Moore's Law, clock rates increased, and software was written to exploit this relentless growth in performance, often ahead of the hardware curve. That symbiotic hardware–software relationship continued unabated until very recently. Moore's Law is still in effect, but gone is the unnamed law that said clock rates would continue to increase commensurately.

The reasons for this change in hardware direction can be summarized by a simple equation, formulated by David Patterson of the University of California at Berkeley:

Power Wall + Memory Wall + ILP Wall = A Brick Wall for Serial Performance

Power dissipation in the CPU increases proportionally with clock frequency, imposing a practical limit on clock rates. Today, the ability to dissipate heat has reached a practical physical limit. As a result, a significant increase in clock speed without heroic (and expensive) cooling (or materials technology breakthroughs) is not possible. This is the "Power Wall" part of the equation. Improvements in memory performance increasingly lag behind gains in processor performance, causing the number of CPU cycles required to access main memory to grow continuously. This is the "Memory Wall." Finally, hardware engineers have improved the performance of sequential software by speculatively executing instructions before the results of current instructions are known, a technique called instruction level parallelism (ILP). ILP improvements are difficult to forecast, and their complexity raises power consumption. As a result, ILP improvements have also stalled, resulting in the "ILP Wall."

We have, therefore, arrived at an inflection point. The software ecosystem must evolve to better support manycore systems, and this evolution will take time. To benefit from rapidly improving computer performance and to retain the "write once, run faster on new hardware" paradigm, the programming community must learn to construct concurrent applications. Broader adoption of concurrency will also enable *Software + Services* through asynchrony and loose-coupling, client-side parallelism, and server-side cloud computing.

The Windows and .NET Framework platforms offer rich support for concurrency. This support has evolved over more than a decade, since the introduction of multiprocessor support in Windows NT. Continued improvements in thread scheduling performance, synchronization APIs, and memory hierarchy awareness—particularly those added in Windows Vista—make Windows the operating system of choice for maximizing the use of hardware concurrency. This book covers all of these areas. When you begin using multithreading throughout an application, the importance of clean architecture and design is critical to reducing software complexity and improving maintainability. This places an emphasis on understanding not only the platform's capabilities but also emerging best practices. Joe does a great job interspersing best practice alongside mechanism throughout this book.

Manycore provides improved performance for the kinds of applications we already create. But it also offers an opportunity to think completely differently about what computers should be able to do for people. The continued increase in compute power will qualitatively change the applications that we can create in ways that make them a lot more interesting and helpful to people, and able to do new things that have never been possible in the past. Through this evolution, software will enable more personalized and humanistic ways for us to interact with computers. So enjoy this book. It offers a lot of great information that will guide you as you take your first steps toward writing concurrent, manycore aware software on the Windows platform.

Craig Mundie
Chief Research and Strategy Officer
Microsoft Corporation
June 2008

Preface

I BEGAN WRITING this book toward the end of 2005. At the time, dual-core processors were becoming standard on the mainstream PCs that ordinary (nonprogrammer) consumers were buying, and a small number of people in industry had begun to make noise about the impending concurrency problem. (Herb Sutter's, The Free Lunch is Over, paper immediately comes to mind.) The problem people were worried about, of course, was that the software of the past was not written in a way that would allow it to naturally exploit that additional compute power. Contrast that with the never-ending increase in clock speeds. No more free lunch, indeed.

It seemed to me that concurrency was going to eventually be an important part of every software developer's job and that a book such as this one would be important and useful. Just two years later, the impact is beginning to ripple up from the OS, through the libraries, and on up to applications themselves.

This was about the same time I had just wrapped up prototyping a small side project I had been working on for six months, called Parallel Language Integrated Query (PLINQ). The PLINQ project was a conduit for me to explore the intricacies of concurrency, multicore, and specifically how parallelism might be used in real-world, everyday programs. I used it as a tool to figure out where the platform was lacking. This was in addition to spending my day job at Microsoft focused on software transactional memory (STM), a technology that in the intervening two years has become somewhat of an industry buzzword. Needless to say, I had become pretty entrenched in all topics concurrency. What better way to get entrenched even further than to write a book on the subject?

As I worked on all of these projects, and eventually PLINQ grew into the Parallel Extensions to the .NET Framework technology, I was amazed at how few good books on Windows concurrency were available. I remember time and time again being astonished or amazed at some intricate and esoteric bit of concurrency-related information, jotting it down, and earmarking it for inclusion in this book. I only wished somebody had written it down before me, so that I didn't need to scour it from numerous sources: hallway conversations, long nights of pouring over Windows and CLR source code, and reading and rereading countless Microsoft employee blogs. But the best books on the topic dated back to the early '90s and, while still really good, focused too much on the mechanics and not enough on how to structure parallel programs, implement parallel algorithms, deal with concurrency hazards, and all those important concepts. Everything else targeted academics and researchers, rather than application, system, and library developers.

I set out to write a book that I'd have found fascinating and a useful way to shortcut all of the random bits of information I had to learn throughout. Although it took me a surprisingly long two-and-a-half years to finish this book, the state of the art has evolved slowly, and the state of good books on the topic hasn't changed much either. The result of my efforts, I hope, is a new book that is down to earth and useful, but still full of very deep technical information. It is for any Windows or .NET developer who believes that concurrency is going to be a fundamental requirement of all software somewhere down the road, as all industry trends seem to imply.

I look forward to kicking back and enjoying this book. And I sincerely hope you do too.

Book Structure

I've structured the book into four major sections. The first, Concepts, introduces concurrency at a high level without going too deep into any one topic. The next section, Mechanisms, focuses squarely on the fundamental platform features, inner workings, and API details. After that, the Techniques section describes common patterns, best practices, algorithms, and data structures that emerge while writing concurrent software. The fourth section, Systems, covers many of the system-wide architectural and process concerns that frequently arise. There is a progression here. Concepts is first because it develops a basic understanding of concurrency in general. Understanding the content in Techniques would be difficult without a solid understanding of the Mechanisms, and similarly, building real Systems would be impossible without understanding the rest. There are also two appendices at the end.

Code Requirements

To run code found in this book, you'll need to download some free pieces of software.

- **Microsoft Windows SDK.** This includes the Microsoft C++ compiler and relevant platform headers and libraries. The latest versions as of this writing are the Windows Vista and Server 2008 SDKs.
- **Microsoft .NET Framework SDK.** This includes the Microsoft C# and Visual Basic compilers, and relevant framework libraries. The latest version as of this writing is the .NET Framework 3.5 SDK.

Both can be found on MSDN: http://msdn.microsoft.com.

In addition, it's highly recommended that you consider using Visual Studio. This is not required—and in fact, much of the code in this book was written in emacs—but provides for a more seamless development and debugging experience. Visual Studio 2008 Express Edition can be downloaded for free, although it lacks many useful capabilities such as performance profiling.

Finally, the debugging tools for Windows package, which includes the popular WINDBG debugging utility—can also come in handy, particularly if you don't have Visual Studio. It is freely downloadable from http://www.microsoft.com. Similarly, the Sysinternals utilities available from http://technet.microsoft.com/sysinternals are quite useful for inspecting aspects of the Windows OS.

A companion website is available at:
http://www.bluebytesoftware.com/books

Joe Duffy
June 2008

joe@bluebytesoftware.com
http://www.bluebytesoftware.com

2 Synchronization and Time

STATE IS AN important part of any computer system. This point seems so obvious that it sounds silly to say it explicitly. But state within even a single computer program is seldom a simple thing, and, in fact, is often scattered throughout the program, involving complex interrelationships and different components responsible for managing state transitions, persistence, and so on. Some of this state may reside inside a process's memory—whether that means memory allocated dynamically in the heap (e.g., objects) or on thread stacks—as well as files on-disk, data stored remotely in database systems, spread across one or more remote systems accessed over a network, and so on. The relationships between related parts may be protected by transactions, handcrafted semitransactional systems, or nothing at all.

The broad problems associated with state management, such as keeping all sources of state in-synch, and architecting consistency and recoverability plans all grow in complexity as the system itself grows and are all traditionally very tricky problems. If one part of the system fails, either state must have been protected so as to avoid corruption entirely (which is generally not possible) or some means of recovering from a known safe point must be put into place.

While state management is primarily outside of the scope of this book, state "in-the-small" is fundamental to building concurrent programs. Most Windows systems are built with a strong dependency on *shared memory* due to the way in which many threads inside a process share access to the same virtual memory address space. The introduction of concurrent access to such state introduces some tough challenges. With concurrency, many parts of the program may simultaneously try to read or write to the same shared memory locations, which, if left uncontrolled, will quickly wreak havoc. This is due to a fundamental concurrency problem called a *data race* or often just *race condition.* Because such things manifest only during certain interactions between concurrent parts of the system, it's all too easy to be given a false sense of security—that the possibility of havoc does not exist.

In this chapter, we'll take a look at state and synchronization at a fairly high level. We'll review the three general approaches to managing state in a concurrent system:

1. **Isolation,** ensuring each concurrent part of the system has its own copy of state.
2. **Immutability,** meaning that shared state is read-only and never modified, and
3. **Synchronization,** which ensures multiple concurrent parts that wish to access the same shared state simultaneously cooperate to do so in a safe way.

We won't explore the real mechanisms offered by Windows and the .NET Framework yet. The aim is to understand the fundamental principles first, leaving many important details for subsequent chapters, though pseudo-code will be used often for illustration.

We also will look at the relationship between state, control flow, and the impact on coordination among concurrent threads in this chapter. This brings about a different kind of synchronization that helps to coordinate state dependencies between threads. This usually requires some form of waiting and notification. We use the term **control synchronization** to differentiate this from the kind of synchronization described above, which we will term **data synchronization.**

Managing Program State

Before discussing the three techniques mentioned above, let's first be very precise about what the terminology **shared state** means. In short, it's any state that is accessible by more than one thread at a time. It's surprisingly difficult to pin down more precisely, and the programming languages commonly in use on the platform are not of help.

Identifying Shared vs. Private State

In object oriented systems, state in the system is primarily instance and static (a.k.a. class) fields. In procedural systems, or in languages like C++ that support a mixture of object oriented and procedural constructs, state is also held in global variables. In thread based programming systems, state may also take the form of local variables and arguments on thread stacks used during the execution and invocation of functions. There are also several other subtle sources of state distributed throughout many layers in the overall infrastructure: code, DLLs, thread local storage (TLS), runtime and OS resources, and even state that spans multiple processes (such as memory mapped files and even many OS resources).

Now the question is "What constitutes 'shared state' versus 'private state?'" The answer depends on the precise mechanisms you are using to introduce concurrency into the system. Stated generally, shared state is any state that may, at any point in time, be accessed by multiple threads concurrently. In the systems we care about, that means:

- All state pointed to by a global or static field is shared.
- Any state passed during thread creation (from creator to createe) is shared.
- Any state reachable through references in said state is also shared, transitively.

As a programmer, it's important to be very conscious of these points, particularly the last. The transitive nature of sharing and the fact that, given any arbitrary pointer, you cannot tell whether the state it refers to has been shared or not, cause tremendous difficulty in building concurrent systems on Windows. Once something becomes shared, it can be difficult to track its ownership in the system, particularly to determine precisely at what point it becomes shared and at what point it becomes unshared in the future (if at all). These can be referred to as **data publication** and **privatization,** respectively. Certain programming patterns such as producer/consumer use consistent sharing and transfer of ownership patterns, making the points of publication and privatization more apparent. Even then it's easy to trip up and make a mistake, such as treating something private although it is still shared, causing race conditions.

It's also important to note that the above definitions depend to some degree on modern type safety. In the .NET Framework this is generally not negotiable, whereas in systems like C++ it is highly encouraged but can be circumvented. When any part of the program can manufacture a pointer to any arbitrary address in the process's address space, all data in the entire address space is shared state. We will ignore this loophole. But when pointer arithmetic is involved in your system, know that many of the same problems we'll look at in this chapter can manifest. They can be even more nondeterministic and hard to debug, however.

To illustrate some of the challenges in identifying shared state, here's a class definition in C++. It has one simple method, `f`, and two fields, one static (`s_f`) and the other instance (`m_f`). Despite the use of C++ here, the same principles clearly apply to managed code too.

```
class C
{
    static int s_f;
    int m_f;
public:
    void f(int * py)
    {
        int x;
        x++;      // local variable
        s_f++;    // static class member
        m_f++;    // class member
        (*py)++; // pointer to something
    }
};
```

The method contains four read/increment/write operations (via C++'s `++` unary operator). In a concurrent system, it is possible that multiple threads could be invoking `f` on the same instance of `C` concurrently with one another. Some of these increments will be safe to perform while others are not. Others still might only be safe if `f` is called in certain ways. We'll see many detailed examples of what can go wrong with this example. Simply put, any increments of shared data are problematic. This is not strictly true because higher level programming conventions and constructs may actually prevent problematic shared interactions, but given the information above, we have no choice but to assume the worst.

By simply looking at the class definition above, how do we determine what state is shared? Unfortunately we can't. We need more information. The answer to this question depends on how instances of `C` are used in addition to where the `py` pointer came from.

We can quickly label the operations that do not act on shared state because there are so few (just one). The only memory location not shared with other threads is the `x` variable, so the `x++` statement doesn't modify shared state. (Similar to the statement above about type safety, we are relying on the fact that we haven't previously shared the address of `x` on the thread's stack with another thread. Of course, another thread might have found an address to the stack through some other means and could perform address arithmetic to access `x` indirectly, but this is a remote possibility. Again, we will assume some reasonable degree of type safety.) Though it doesn't appear in this example, if there was a statement to increment the value of `py`, i.e., `py++`, it would not affect shared state because `py` is passed by value.

The `s_f++` statement affects shared state because, by the definition of static variables, the class's static memory is visible to multiple threads running at once. Had we used a static local variable in `f` in the above example, it would fall into this category too.

Here's where it becomes complicated. The `m_f++` line might, at first glance, appear to act on private memory, but we don't have enough information to know. Whether it modifies shared state or not depends on if the caller has shared the instance of `C` across multiple threads (or itself received the pointer from a caller that has shared the instance). Remember, `m_f++` is a pointer dereference internally, `(this->m_f)++`. The `this` pointer might refer to an object allocated on the current thread's stack or allocated dynamically on the heap and may or may not be shared among threads in either case.

```
class D
{
    static C s_c;  // initialized elsewhere...
    C m_c;         // also initialized elsewhere...
```

```
        void g()
        {
            int x = 0;

            C c1(); // stack-alloc
            c1.f(&x);

            C c2 = new C(); // heap-alloc
            c2.f(&x);
            s_c.f(&x);
            m_c.f(&x);
        }
    }
```

In the case of the `c1.f(&x)` function call, the object is private because it was allocated on the stack. Similarly, with `c2.f(&x)` the object is probably private because, although allocated on the heap, the instance is not shared with other threads. (Neither case is simple: `C`'s constructor could publish a reference to itself to a shared location, making the object shared before the call to `f` happens.) When called through `s_c`, clearly the object is shared because it is stored in a shared static variable. And the answer for the call through `m_c` is "it depends." What does it depend on? It depends on the allocation of the instance of `D` through which `g` has being invoked. Is it referred to by a static variable elsewhere, another shared object, and so forth? This illustrates how quickly the process of identifying shared state is transitive and often depends on complex, dynamically composed object graphs.

Because the member variable and explicit pointer dereference are similar in nature, you can probably guess why "it depends" for `(*py)++` too. The caller of `f` might be passing a pointer to a private or shared piece of memory. We really have no way of telling.

Determining all of this statically is impossible without some form of type system support (which is not offered by VC++ or any mainstream .NET languages). The process of calculating the set of shared objects dynamically also is even difficult but possible. The process can be modeled much in the same way garbage collection works: by defining the set of shared roots as those objects referenced directly by static variables, we could then traverse the entire reachable set of objects beginning with only those roots, marking all objects as we encounter them (avoiding cycles). At the end, we know that all marked objects are shared. But this approach is too naïve. An object can also become shared at thread creation time by passing a pointer to it as an argument to thread creation routines. The same goes for thread pool APIs, among others. Some objects are special, such as the one global shared `OutOfMemoryException` object that the CLR throws when memory is very low. Some degree of compiler analysis could help. A technique called escape analysis determines when private memory "escapes" into the shared memory space, but its application is limited mostly to academic papers (see Further Reading, Choi, Gupta, Serrano, Sreedhar, Midkiff). In practice, complications, such as late bound method calls, pointer aliasing, and hidden sources of cross-thread sharing, make static analysis generally infeasible and subject to false negatives without restrictions in the programming model. There is research exploring such ideas, such as ownership types, but it is probably years from mainstream use (see Further Reading, Boyapati, Liskov, Shrira).

In the end, logically separating memory that is shared from memory that is private is of utmost importance. This is perhaps the most fundamental and crucial skill to develop when building concurrent systems in modern programming environments: accurately identifying and properly managing shared state. And, more often than not, shared state must be managed carefully and with a great eye for detail. This is also why understanding and debugging concurrent code that someone else wrote is often very difficult.

State Machines and Time

All programs are state machines. Not all people think of their programs this way, but it turns out to be a convenient mental model for concurrent programs. Even if you don't think about your program as a state machine proper, you probably at least think about your program in terms of time and the sequence of program events on a sequential timeline: the order in which reads from and writes to variables occur, the time distance between two such events, and so on. A unique problem with concurrency thus arises. We are accustomed to reasoning about the code we write on the screen in sequential order, which is necessarily written in a sequential layout. We form mental models and conclusions about the state transitions possible with these assumptions firmly in mind. However, concurrency invalidates many such assumptions.

When state is shared, multiple concurrent threads, each of which may have been constructed with a set of sequential execution assumptions, may end up overlapping in time. And when they overlap in time, their operations become interleaved. If these operations access common memory locations, they may possibly violate the legal set of state transitions that the program's state machine was planned and written to accommodate. Once this happens, the program may veer wildly off course, doing strange and inexplicable things that the author never intended, including performing bogus operations, corrupting memory, or crashing.

Broken Invariants and Invalid States

As an illustration, let's say on your first day at a new programming job you were assigned the task of implementing a reusable, dynamically resizing queue data structure. You'd probably start out with a sketch of the algorithms and outline some storage alternatives. You'd end up with some fields and methods and some basic decisions having been made, perhaps such as using an array to store elements versus a linked list. If you're really methodical, you might write down the state invariants and transitions and write them down as asserts in the code or even use a formal specification system to capture (and later verify) them. But even if you didn't go to these lengths, those invariants still exist. Break any one of them during development, or worse after code has been embedded into a system, and you've got a bug.

Let's consider a really simple invariant. The count of the queue must be less than or equal to the length of the array used to store the individual elements. (There are of course several others: the head and tail indices must be within the legal range, and so on.) If this queue was meant only to be used by sequential programs, then preserving the invariant at the entrance and exit of all public methods would be sufficient as a correctness condition. It would be trivial: only those methods that modify the fields need to be written to carefully respect the invariant. The most difficult aspect of attaining this would be dealing with failures, such as an inability to allocate memory when needed.

Things become much more difficult as soon as concurrency is added to the system. Unless another approach is used, you would have to ensure invariants held at every single line of code in your implementation. And even that might not be sufficient if some lines of code (in whatever higher level language you are programming in) were compiled into multiple instructions in the machine language. Moreover, this task becomes impossible when there are multiple variables involved in the operation (as is probably the case with our queue), leading to the requirement of some extra form of state management: i.e., isolation, immutability, or synchronization.

The fact is that it's very easy to accidentally expose invalid program states as a result of subtle interactions between threads. These states might not exist on any legal state machine diagram we would have drawn for our data structure, but interleaving can cause them. Such problems frequently differ in symptom from one execution of your code to the next—causing new exceptions, data corruption, and so forth and depend on timing in order to manifest. The constant change in symptom and dependence on timing makes it difficult to anticipate the types of failures you will experience when more concurrency is added to the system and makes such failures incredibly hard to debug and fix.

The various solutions hinted at above can solve this problem. The simplest solutions are to avoid sharing data or to avoid updating data completely. Unfortunately, taking such an approach does not completely eliminate the need to synchronize. For instance, you must keep intermediate state changes confined within one thread until they are all complete and then, once the changes are suitable to become visible, you must use some mechanism to publish state updates to the globally visible set of memory as a single, indivisible operation (i.e., atomic operation). All other threads must cooperate by reading such state from the global memory space as a single, indivisible atomic operation.

This is not simple to achieve. Because reading and writing an arbitrary number of memory locations atomically at once are not supported by current hardware, software must simulate this effect using **critical regions.** A critical region ensures that only one thread executes a certain piece of code at once, eliminating problematic interleaved operations and forcing one after the other timing. This implies some threads in the system will have to wait for others to finish work before doing their own. We will discuss critical regions later. But first, let's look at a motivating example where data synchronization is direly needed.

A Simple Data Race

Consider this deceivingly simple program statement.

```
int * a = ...;
(*a)++;
```

(Forgive the C++-isms for those managed programmers reading this. `(*a)++` is used instead of `a++`, just to make it obvious that `a` points to some shared memory location.)

When translated into machine code by the compiler this seemingly simple, high-level, single-line statement involves multiple machine instructions:

```
MOV EAX, [a]
INC EAX
MOV [a], EAX
```

Notice that, as a first step, the machine code dereferences `a` to get some virtual memory address and copies 4 bytes' worth of memory starting at that address into the processor local `EAX` register. The code then increments the value of its private copy in `EAX`, and, lastly, makes yet another copy of the value, this time to copy the incremented value held in its private register back to the shared memory location referred to by `a`.

The multiple steps and copies involved in the `++` operator weren't apparent in the source file at all. If you were manipulating multiple variables explicitly, the fact that there are multiple steps would be a little more apparent. In fact, it's as though we had written:

```
int * a = ...;
int tmp = *a;
tmp++;
*a = tmp;
```

Any software operation that requires multiple hardware instructions is nonatomic. And thus we've now established that `++` is nonatomic (as is `--`), meaning we will have to take extra steps to ensure concurrency safety. There are some other nonobvious sources of nonatomic operations. Modern processors guarantee that single reads from and writes to memory in increments of the natural word size of the machine will be carried out atomically covering 32-bit values on 32-bit machines and 64-bit values on 64-bit machines.

Conversely, reading or writing data with a size larger than the addressable unit of memory on your CPU is nonatomic. For instance, if you wrote a 64-bit value on a 32-bit machine, it will entail two move instructions from processor private to shared memory, each to copy a 4-byte segment. Similarly, reading from or writing to unaligned addresses (i.e., address ranges that span an addressable unit of memory) also require multiple memory operations in addition to some bit masking and shifting, even if the size of the value is less than or equal to the machine's addressable memory size. Alignment is a tricky subject and is discussed in much more detail in Chapter 10, Memory Models and Lock Freedom.

So why is all of this a problem?

An increment statement is meant to monotonically increase the value held in some memory location by a delta of 1. If three increments were made to a counter with an original value 0, you'd expect the final result to be 3. It should never be possible (overflow aside) for the value of the counter to decrease from one read to the next; therefore, if a thread executes two `(*a)++` operations, one after the other, you would expect that the second update always yields a higher value than the first. These are some very basic correctness conditions for our simple `(*a)++` program. (Note: You shouldn't be expecting that the two values will differ by precisely 1, however, since another thread might have snuck in and run between them.)

There's a problem. While the actual loads and stores execute atomically by themselves, the three operation sequence of load, increment, and store is nonatomic, as we've already established. Imagine three threads, t1, t2, and t3, are running the compiled program instructions simultaneously.

```
t1                     t2                     t3
t1(0): MOV EAX,[a]     t2(0): MOV EAX,[a]     t3(0): MOV EAX,[a]
t1(1): INC,EAX         t2(1): INC,EAX         t3(1): INC,EAX
t1(2): MOV [a],EAX     t2(2): MOV [a],EAX     t3(2): MOV [a],EAX
```

Each thread is running on a separate processor. Of course, this means that each processor has its own private `EAX` register, but all threads see the same value in `a` and therefore access the same shared memory. This is where time becomes a very useful tool for explaining the behavior of our concurrent programs. Each of these steps won't really happen "simultaneously." Although separate processors can certainly execute instructions simultaneously, there is only one central, shared memory system with a cache coherency system that ensures a globally consistent view of memory. We can therefore describe the execution history of our program in terms of a simple, sequential time scale.

In the following time scale, the y-axis (labeled T) represents time, and the abscissa, in addition to a label of the form thread (sequence#) and the instruction itself, depicts a value in the form #n, where n is the value in the memory target of the move after the instruction has been executed.

```
T t1                        t2                          t3
0 t1(0): MOV EAX,[a] #0
1 t1(1): INC,EAX     #1
2 t1(2): MOV [a],EAX #1
3                           t2(0): MOV EAX,[a] #1
4                           t2(1): INC,EAX     #2
5                           t2(2): MOV [a],EAX #2
6                                                       t3(0): MOV EAX,[a] #2
7                                                       t3(1): INC,EAX     #3
8                                                       t3(2): MOV [a],EAX #3
```

If `a` is an integer that begins with a value of 0 at time step 0, then after three `(*a)++` operations have executed, we expect the value to be 0 + 3 = 3. Indeed, we see that this is true for this particular history: t1 runs to completion, leaving value 1 in `*a`, and then t2, leaving value 2, and finally, after executing the instruction at time 8 in our timeline, t3 has finished and `*a` contains the expected value 3.

We can compress program histories into more concise representations so that they fit on one line instead of needing a table like this. Because only one instruction executes at any time step, this is simple to accomplish. We'll write each event in sequence, each with a thread (sequence#) label, using the notation a -> b to denote that event a happens before b. A sequence of operations is written from left to right, with the time advancing as we move from one operation to the next. Using this scheme, the above history could be written instead as follows.

```
t1(0)->t1(1)->t1(2)->t2(0)->t2(1)->t2(2)->t3(0)->t3(1)->t3(2)
```

We'll use one form or the other depending on the level of scrutiny in which we're interested for that particular example. The longhand form is often clearer to illustrate specific values and is better at visualizing subtle timing issues, particularly for larger numbers of threads.

No matter the notation, examining timing like this is a great way of reasoning about the execution of concurrent programs. Programmers are accustomed to thinking about programs as a sequence of individual steps. As you develop your own algorithms, writing out the concurrent threads and exploring various legal interleavings and what they mean to the state of your program, it is imperative to understanding the behavior of your concurrent programs. When you think you might have a problematic timing issue, going to the whiteboard and trying to devise some problematic history, perhaps in front of a colleague, is often an effective way to uncover concurrency hazards (or determine their absence).

Simple, noninterleaved histories pose no problems for our example. The following histories are also safe with our algorithm as written.

```
t1(0)->t1(1)->t1(2)->t3(0)->t3(1)->t3(2)->t2(0)->t2(1)->t2(2)
t2(0)->t2(1)->t2(2)->t1(0)->t1(1)->t1(2)->t3(0)->t3(1)->t3(2)
t2(0)->t2(1)->t2(2)->t3(0)->t3(1)->t3(2)->t1(0)->t1(1)->t1(2)
t3(0)->t3(1)->t3(2)->t1(0)->t1(1)->t1(2)->t2(0)->t2(1)->t2(2)
t3(0)->t3(1)->t3(2)->t2(0)->t2(1)->t2(2)->t1(0)->t1(1)->t1(2)
```

These histories yield correct results because none results in one thread's statements interleaving amongst another's. In each scenario, the first thread runs to completion, then another, and then the last one. In these histories, the threads are **serialized** with respect to one another (or the history is **serializable**).

But this example is working properly by virtue of sheer luck. There is nothing to prevent the other interleaved histories from occurring at runtime, where two (or more) threads overlap in time, leading to an interleaved timing and resulting race conditions. Omitting t3 from the example for a moment, consider this simple timing, written out longhand so we can emphasize the state transitions from one time step to the next.

```
T t1                          t2
0 t1(0): MOV EAX,[a] #0
1                             t2(0): MOV EAX,[a] #0
2                             t2(1): INC,EAX     #1
3                             t2(2): MOV [a],EAX #1
4 t1(1): INC,EAX     #1
5 t1(2): MOV [a],EAX #1
```

The value of `*a` starts at 0. Because two increments happen, we would expect the resulting value to be 0 + 2 = 2. However, `*a` ends up at 1. This clearly violates the first correctness condition of our algorithm as stated initially: for each thread that invokes the increment operator, the global counter increments by exactly 1.

This is a classic race condition, or more precisely, a data race, because, in this case, our problems are caused by a lack of data synchronization. It is called a "race" because the correctness of our code depends squarely on the outcome of multiple threads racing with one another. It's as if each is trying to get to the finish line first, and, depending on which gets there first, the program will yield different results, sometimes correct and sometimes not. Races are just one of many issues that can arise when shared state is involved and can be a serious threat to program correctness. A thorough exploration of concurrency hazards, including races, is presented in Chapter 11, Concurrency Hazards.

Why did this race manifest? It happened because t1 and t2 each made a copy of the shared memory value in their own processor local register, one after the other, both observing the same value of 0, and then incremented their own private copies. Then both copied their new values back into the shared memory without any validation or synchronization that would prevent one from overwriting the other's value. Both threads calculate the value 1 in their private registers, without knowledge of each other, and, in this particular case, t1 just overwrites t2's earlier write of 1 to the shared location with the same value.

One might question how likely this is to occur. (Note that the likelihood matters very little. The mere fact that it can occur means that it is a very serious bug. Depending on the statistical improbability of such things is seriously discouraged. A program is not correct unless all possible sources of data races have been eliminated.) This interleaved history can happen quite easily, for obvious reasons, if t1 and t2 were running on separate processors. The frequency depends on the frequency with which the routine is accessed, among other things. This problem can also arise on a single processor machine, if a context switch occurred—because t1's quantum had expired, because t2 was running at a higher priority, and so forth—right after `t1` had moved the contents of a into its `EAX` register or after it had incremented its private value. The probability of this happening is higher on a machine with multiple processors, but just having multiple threads running on a single processor machine is enough. The only way this may be impossible is if code accessing the same shared state is never called from multiple threads simultaneously.

Other execution histories exhibit the same problem.

```
t1(0)->t2(0)->t1(1)->t1(2)->t2(1)->t2(2)
t1(0)->t1(1)->t2(0)->t1(2)->t2(1)->t2(2)
t2(0)->t1(0)->t1(1)->t1(2)->t2(1)->t2(2)
...and so on
```

If we add the t3 thread back into the picture, we can violate the second correctness condition of our simple increment statement, in addition to the first, all at once.

```
T t1                    t2                    t3
0                                             t3(0): MOV EAX,[a] #0
1 t1(0): MOV EAX,[a] #0
2 t1(1): INC,EAX     #1
3 t1(2): MOV [a],EAX #1
4                       t2(0): MOV EAX,[a] #1
5                       t2(1): INC,EAX     #2
6                       t2(2): MOV [a],EAX #2
7                                             t3(1): INC,EAX     #1
8                                             t3(2): MOV [a],EAX #1
```

In this program history, the global counter is updated to 1 by t1, and then to 2 by t2. Everything looks fine from the perspective of other threads in the system at this point in time. But as soon as t3 resumes, it wipes out t1's and t2's updates, "losing" two values from the counter and going backward to a value of 1. This is because t3 made its private copy of the shared value of *a before t1 and t2 even ran. The second correctness condition was that the value only ever increases; but if t2 runs again, it will see a value smaller than the one it previously published. This is clearly a problem that is apt to break whatever algorithm is involved. As we add more and more threads that are frequently running close together in time, we increase the probability of such problematic timings accordingly.

All of these histories demonstrate different kinds of hazards.

- **Read/write hazard.** A thread, t1, reads from a location, t2, then writes to that location, and t1 subsequently makes a decision based on its (now invalid) read of t1. This also can be referred to as a **stale read.**
- **Write/write hazard.** A thread, t1, writes to the same location as t2 in a concurrency unsafe way, leading to lost updates, as in the example given above.
- **Write/read hazard.** A thread, t1, writes to a location and then t2 reads from it before it is safe to do so. In some cases, t1 may decide to undo its partial update to state due to a subsequent failure, leading t2 to make decisions on an invalid snapshot of state that should have never been witnessed. This also can be referred to as an **unrepeatable read.**
- **Read/read hazard.** There is no problem with multiple concurrent threads reading the same shared data simultaneously. This property can be exploited to build a critical region variant called a **reader/ writer lock** to provide better performance for read/read conflicts; this idea is explored more in Chapter 6, Data and Control Synchronization.

(This last point is a simplification. Normally read/read conflicts are safe in the case of simple shared memory, but there are some cases in which they are not: when a read has a side effect, like reading a stack's guard page, or when reading some data associated with a physical device, it may be necessary to ensure no two threads try to do it concurrently.)

Very little of this discussion is specific to the ++ operator itself. It just turns out to be a convenient example because it intrinsically exhibits all of the problematic conditions that lead to these timing issues.

1. Multiple threads make private copies of data from a shared location.
2. Threads publish results back to shared memory, overwriting existing values.
3. Compound updates may be made with the intent of establishing or preserving invariants between multiple independent shared locations.
4. Threads run concurrently such that their timing overlaps and operations interleave.

There is no greater skill that differentiates great concurrent programmers from the rest than the ability to innately predict and consider various timings to some reasonable depth of complexity. With experience comes the ability to see several steps ahead and proactively identify the timings that can lead to race conditions and other hazards. This is especially important when writing sophisticated **lock free** algorithms, which eschew isolation, immutability, and synchronization in favor of strict discipline and reliance on hardware guarantees, which we'll review in Chapter 10, Memory Models and Lock Freedom.

On Atomicity, Serializability, and Linearizability

A fundamental problem is that many program operations are not truly atomic because an operation consists of multiple logical steps, a certain logical step is comprised of many physical steps, or both. **Atomicity,** quite simply, is the property that a single operation or set of operations appear as if they happened at once. Any state modifications and side effects performed are completely instantaneous, and no other thread in the system can observe intermediary (and invalid) states that occur in the midst of such an atomic operation. Similarly, the atomic operation must not be permitted to fail part way through the update, or if it does so, there must be a corresponding roll back of state updates to the previous state.

By this definition, atomicity would seldom be practical to achieve, at least physically. Although processors guarantee single writes to aligned words of memory are truly atomic, higher level logical operations—like the execution of a single method call on an object, consisting of several statements, function calls, and reads and writes—are not so simple. In fact, sometimes the operations we'd like to make atomic can even span physical machines, perhaps interacting with a Web service or database, at which point the difficulty of ensuring atomicity is greater. System wide control mechanisms must be used to achieve atomicity except for very simple read and write operations. As already noted, critical regions can simulate atomicity for in-memory updates. Transactions, of the ilk found in databases, COM+, and the `System.Transactions` namespace in .NET, are also attractive solutions when multiple or persistent durable resources are involved.

When two operations are atomic, they do not appear to overlap in time. If we were to plot several atomic operations on a timeline, then we could place one before or after the other without worrying about having to interleave them. We did this earlier for individual reads and writes, and it was possible because of the guarantees made by the hardware that they are atomic. Object oriented programs are typically built from higher level atomic methods, however, and reasoning about concurrency at this level (like "puts an element in the queue," "writes data to disk," and so forth), and not about the individual memory reads and writes involved, is often more useful.

Serializability is when two operations happen one after the other; if a happens before b, then *a serializes before b*. Building your program out of atomic operations achieves serializability. It's as though your program was executed sequentially, by a single processor, by executing each atomic operation in the sequence as it appeared in the resulting serializable order. But serializability on its own is insufficient for correctness; and it's often in the eye of the beholder—remember that even individual reads and writes are themselves atomic. For a concurrent program to be correct, all possible serialization orders must be legal. Techniques like critical regions can be used to constrain legal serialization orders.

Linearizability is a property related to serializability and also is used to describe correctness of atomic operations (see Further Reading, Herlihy, Wing): a *linearization point* is a place when a batch of atomic updates becomes visible to other threads. This commonly corresponds to exiting a critical region where the updates made within suddenly become visible. It is typically easier to reason about linearization points instead of the more abstract serialization property.

Atomic operations also must be reorderable, such that having one start completely before the other still leads to a correct program schedule. That's not to say that subsequently initiated operations will not change behavior based on the changed order of commutative operations, due to causality, but this reordering should not fundamentally alter the correctness of a program.

As software developers, we like to think of serializable schedules and atomic operations. But we'd also like to use concurrency for the reasons identified earlier in this book, for performance, responsiveness, and so on. For this reason, the Win32 and .NET Framework platforms give you a set of tools to achieve atomicity via data synchronization constructs, as implied earlier. Those familiar with relational databases will recognize a similarity: databases employ transactions to achieve serializable operations, giving the programmer an interface with atomicity, consistency, isolation, and durability (a.k.a. ACID). You will notice many similarities, but you will also notice that these properties must be achieved "by hand" in general purpose concurrent programming environments.

Isolation

An obvious approach to eliminating problematic shared state interactions is to avoid sharing state in the first place. We described how concurrent systems are typically formed out of higher level components that eschew sharing in favor of isolation, and that lower level components typically do share data for purposes of fine-grained, performance sensitive operations. This is a middle ground, but the two extremes are certainly possible: on one hand, all components in the system may share state, while, on the other hand, no components share state and instead communicate only via loosely coupled messages. And there are certainly situations in which the architecture is less clearly defined: i.e., some lower level components will use isolation, while some higher level components will share state for efficiency reasons.

When it comes to employing isolation, there are three basic techniques from which to choose.

- **Process isolation.** Each Windows process has a separate memory address space, ensuring that one process cannot read or write memory used by another. Hardware protection is used to absolutely guarantee that there is no chance of accidental sharing by bleeding memory references. Note that processes do share some things, like machine-wide kernel objects, the file system, memory mapped files, and so on, so even rigid process isolation can be broken. An even more extreme technique is isolating components on separate machines or inside virtualized partitions on a single machine.

- **Intraprocess isolation.** If you are using managed code, CLR Application Domains (AppDomains) can be used to isolate objects so that code running in one AppDomain cannot read or write an object running in another AppDomain. While hardware protection is not used to enforce this isolation, the verifiable type safety employed by the CLR ensures that no sharing will occur. There are some specific ways to circumvent this broadly stated policy, but they are generally opt-in and rare.
- **By convention.** When some code allocates a piece of memory or an object, either dynamically from the heap or on the stack, this data begins life as unshared, and, hence, is in effect isolated. This data remains isolated so long as care is taken to not share the data (as described previously), that is, by not storing a reference to the data in a shared location (like a static variable or object reachable through a static variable). This is the trickiest of the three approaches to implement safely because it is entirely based on programming convention and care, is not checkable in any way and has no infrastructure regulated support such as hardware isolation or type system verification.

It's common to use isolated state as a form of cache. In other words, though some state is physically isolated, it is merely a copy of some master copy that is not isolated. Such designs require that the master copy is periodically refreshed (if updates are made to the cache) and that caches are refreshed as the master copy changes. Depending on the requirements, a more sophisticated cache coherency mechanism may be needed, to guarantee that refreshes happen in a safe and serializable way, requiring a combination of isolation and synchronization techniques.

The last mechanism, enforcement by convention, requires that programs follow some strict disciplines, each of which is cause for concern because they are informal and inherently brittle. It can be useful to think of state in terms of being "owned" by particular "agents" at any point in time. Thinking this way allows you to very clearly articulate where ownership changes for a particular piece of data, including at what point data transitions between isolated and shared.

Data Ownership

At any point in time, a particular piece of isolated data can be said to be owned by one agent in the system. Ownership is used in this context to mean that the agent understands what other components or agents may concurrently access that piece of data, and what this means for the read and write safety of its own operations. If, at any time, multiple agents believe they own the same piece of data, it is likely that the data is no longer truly isolated. Clearly there are many kinds of ownership patterns a system might employ, including shared ownership, but let's stick to the idea of single agent ownership for a moment.

An agent may transfer ownership, but it must do so with care. For example, some agent may allocate and initialize some interesting object, but then insert it into a global shared list. This is called publication. Publication transfers ownership from the initializing agent to the global namespace; at some point in the future, an agent may remove the data from the shared list, at which point the ownership transfers from the global namespace to that agent. This is called privatization. Publication must be done such that the agent doing the transferring no longer tries to access the state as though it is the sole owner: the data is no longer confined (or isolated) within the agent. Similarly, privatization must be done such that other agents do not subsequently try to access the privatized data.

One of the more difficult aspects of ownership is that a piece of data may move between isolation and shared status over the course of its life. These publication and privatization points must be managed with care. A slight misstep, such as erroneously believing an object is private and no longer shared when in reality other threads still have outstanding references to it that they might use, can introduce all of the same kinds of race condition problems noted earlier.

Another challenge with isolation is determining where the points of escape in the program might be. Publication is not always such a clear-cut point in the program's execution. This requires that agents attempting to control ownership of data only ever share references to this data with trusted agents. The agent is trusting that the other agents will not publish the reference so that the data becomes shared, either directly or indirectly (e.g., by passing the reference along to another untrusted agent).

Similarly, an agent that receives a reference to data from an outside source must assume the worst—that the data is shared—unless an alternative arrangement is known, such as only ever being called by an agent that guarantees the data is isolated. Again, the lack of type system and verification support makes this convention notoriously tricky to implement and manage in real programs, particularly when multiple developers are involved.

Immutability

As noted earlier, read/read "hazards" are not really hazardous at all. Many threads can safely read from some shared memory location concurrently without cause for concern. Therefore, if some piece of shared state is guaranteed to be immutable—that is, read-only—then accessing it from many threads inside a concurrent system will be safe.

Proving that a piece of complex data is immutable is not terribly difficult with some discipline. Both C++ and .NET offer constructs to help make immutable types. If each of an object's fields never changes during its lifetime, it is **shallow immutable.** If the object's fields also only refer to objects whose state does not change over time, the object is **deeply immutable.** An entire object graph can be transitively immutable if all objects within it are themselves deeply immutable.

In the case that data transitions between private and shared throughout its lifetime, as discussed above in the context of isolation, it is sometimes useful to have a conditionally-immutable type, in which it remains immutable so long as it is shared but can be mutated while private. So, for example, a thread may remove a piece of shared state from public view, making it temporarily private, mutate it, and then later share the state again to public view.

Single Assignment

A popular technique for enforcing the immutability of data is to use **single assignment variables.** Many programming systems offer static verification that certain data is indeed assigned a value only once, leading to the term **static single assignment,** or **SSA.**

The CLR offers limited support for single assignment variables in its common type system through the `initonly` field modifier, surfaced in C#

through the `readonly` keyword. And C++ offers the `const` modifier to achieve a similar effect, though it is far more powerful: pointers may be marked as being `const`, ensuring (statically) that the instance referred to is not modified by the user of such a pointer (though unlike `readonly` C++ programmers can explicitly "cast away the `const`-ness" of a reference, bypassing the safety guarantees). Using these constructs can be tremendously useful because it avoids having to depend on brittle and subtle programming convention and rules. Let's look at each briefly.

CLR `initonly` Fields (a.k.a. C# `readonly` Fields). When you mark a field as `readonly` in C#, the compiler emits a field with the `initonly` modifier in the resulting IL. The only writes to such variables that will pass the type system's verification process are those that occur inside that type's constructor or field initializers. This ensures that the value of such a field cannot change after the object has been constructed. While it is not a true single assignment variable, as it can be written multiple times during initialization, it is similar in spirit.

Another subtle issue can arise if a reference to an object with `readonly` fields escapes from its constructor. Fields are not guaranteed to have been initialized with the permanent immutable values until after the constructor has finished running and could be assigned multiple values during the construction process. If an object's constructor shares itself before finishing initialization, then other concurrent threads in the system cannot safely depend on the `readonly` nature of the fields. Letting the object's `this` reference escape before the object is fully constructed like this is a bad practice anyway, and is easily avoided. When a field is marked `readonly`, it simply means the field's value cannot change. In other words, a type with only `readonly` fields is shallow immutable but not necessarily deeply immutable. If an object depends on the state of the objects it references, then those objects should be immutable also. Unfortunately, the CLR offers no type system support for building deeply immutable types. Of course they may be immutable by convention, `readonly` fields, or a combination of both.

There are some cases where the mutability of referenced objects does not matter. For example, if we had an immutable pair class that refers to two mutable objects but never accesses the state of those objects, then is the pair itself immutable?

```
class ImmutablePair<T, U>
{
    private readonly T m_first;
    private readonly U m_second;

    public ImmutablePair(T first, U second)
    {
        m_first = first;
        m_second = second;
    }

    public T First { get { return m_first; } }
    public U Second { get { return m_second; } }
}
```

From one perspective, the answer is yes. The `ImmutablePair<T, U>` implementation itself cannot tell whether the `m_first` or `m_second` objects have been mutated, since it never accesses their internal state. If it relied on a stable `ToString` value, then it might matter. Those who instantiate `ImmutablePair<T, U>` may or may not care about deep immutability, depending on whether they access the pair's fields; they control this by the arguments they supply for `T` and `U`. So it seems shallow immutability here is sufficient. That said, if a developer desires deep immutability, they need only supply immutable types for `T` and `U`.

C++ Const. C++ const is a very powerful and feature rich-programming language construct, extending well beyond simple single assignment variable capabilities, and encompassing variables, pointers, and class members. A complete overview of the feature is outside of the scope of this book. Please refer instead to a book such as *The C++ Programming Language*, Third Edition (see Further Reading, Stroustrup), for a detailed overview.

Briefly, the `const` modifier can be a useful and effective way of relying on the C++ compiler to guarantee a certain level of immutability in your data structures, including single assignment variables. In summary:

- Class fields may be marked `const`, which enforces that their value is assigned at initialization time in the constructor's field initialization list and may not subsequently change. This effectively turns a field into a single assignment variable, though it may still be modified by a pointer that has been cast a certain way (as we'll see soon).

The value of static const fields cannot depend on runtime evaluation, unlike class member fields that can involve arbitrary runtime computation to generate a value, much like CLR initonly fields. This means they are limited to compiler constants, statically known addresses, and inline allocated arrays of such things.

- Member functions may be marked const, which means that the function body must not modify any fields and ensures that other non-const member functions cannot be invoked (since they may modify fields).
- Pointers can be marked as "pointing to a constant," via the prefix const modifier. For instance, the following declaration states that d points to a constant object of type C: const C * d. When a pointer refers to a constant, only const member functions may be called on it, and the pointer may not be passed where an ordinary non-const pointer is expected. A const pointer to an integral type cannot be written through. A non-const pointer can be supplied where a const is expected. Constant references are also possible.

As noted earlier, a pointer to a constant can be cast to a non-const pointer, which violates most of what was mentioned above. For example, the C++ compiler enforces that a pointer to a const member field also must be a pointer to const; but you can cast this to a non-const pointer and completely subvert the const guarantees protecting the field. For example, given the following class declaration, pointers may be manufactured and used in certain ways.

```
class C
{
public:
    const int d;
    C(int x) : d(x) {}
};

// ... elsewhere ...

C * pC = ...;
const int * pCd1 = &pC->d; // ok!
*pC->d = 42; // compiler error: cannot write to const
int * pCd2 = &pC->d; // compiler error: non-const pointer to const field
int * pCd3 = const_cast<int *>(&pC->d); // succeeds!
```

Casting away `const` is a generally frowned upon practice, but is sometimes necessary. And, a `const` member function can actually modify state, but only if those fields have been marked with the `mutable` modifier. Using this modifier is favored over casting. Despite these limitations, liberal and structured use of `const` can help build up a stronger and more formally checked notion of immutability in your programs. Some of the best code bases I have ever worked on have used `const` pervasively, and in each case, I have found it to help tremendously with the maintainability of the system, even with concurrency set aside.

Dynamic Single Assignment Verification. In most concurrent systems, single assignment has been statically enforced, and C# and C++ have both taken similar approaches. It's possible to dynamically enforce single assignment too. You would just have to reject all subsequent attempts to set the variable after the first (perhaps via an exception), and handle the case where threads attempt to use an uninitialized variable. Implementing this does require some understanding of the synchronization topics about to be discussed, particularly if you wish the end result to be efficient; some sample implementation approaches can be found in research papers (see Further Reading, Drejhammar, Schulte).

Synchronization: Kinds and Techniques

When shared mutable state is present, synchronization is the only remaining technique for ensuring correctness. As you might guess, given that there's an entire chapter in this book dedicated to this topic—Chapter 11, Concurrency Hazards—implementing a properly synchronized system is complicated. In addition to ensuring correctness, synchronization often is necessary for behavioral reasons: threads in a concurrent system often depend on or communicate with other threads in order to accomplish useful functionality.

The term synchronization is admittedly overloaded and too vague on its own to be very useful. Let's be careful to distinguish between two different, but closely related, categories of synchronization, which we'll explore in this book:

1. **Data synchronization.** Shared resources, including memory, must be protected so that threads using the same resource in parallel do

not interfere with one another. Such interference could cause problems ranging from crashes to data corruption, and worse, could occur seemingly at random: the program might produce correct results one time but not the next. A piece of code meant to move money from one bank account to another, written with the assumption of sequential execution, for instance, would likely fail if concurrency were naively added. This includes the possibility of reaching a state in which the transferred money is in neither account! Fixing this problem often requires using mutual exclusion to ensure no two threads access data at the same time.

2. **Control synchronization.** Threads can depend on each others' traversal through the program's flow of control and state space. One thread often needs to wait until another thread or set of threads have reached a specific point in the program's execution, perhaps to rendezvous and exchange data after finishing one step in a cooperative algorithm, or maybe because one thread has assumed the role of orchestrating a set of other threads and they need to be told what to do next. In either case, this is called control synchronization.

The two techniques are not mutually exclusive, and it is quite common to use a combination of the two. For instance, we might want a producer thread to notify a consumer that some data has been made available in a shared buffer, with control synchronization, but we also have to make sure both the producer and consumer access the data safely, using data synchronization.

Although all synchronization can be logically placed into the two general categories mentioned previously, the reality is that there are many ways to implement data and control synchronization in your programs on Windows and the .NET Framework. The choice is often fundamental to your success with concurrency, mostly because of performance. Many design forces come into play during this choice: from correctness—that is, whether the choice leads to correct code—to performance—that is, the impact to the sequential performance of your algorithm—to liveness and scalability—that is, the ability of your program to ensure that, given the addition of more and more processors, the throughput of the system improves commensurately (or at least doesn't do the inverse of this).

Because these are such large topics, we will tease them apart and review them in several subsequent chapters. In this chapter, we stick to the general ideas, providing motivating examples as we go. In Chapter 5, Windows Kernel Synchronization, we look at the foundational Windows kernel support used for synchronization, and then in Chapter 6, Data and Control Synchronization, we will explore higher level primitives available in Win32 and the .NET Framework. We won't discuss performance and scalability in great depth until Chapter 14, Performance and Scalability, although it's a recurring theme throughout the entire book.

Data Synchronization

The solution to the general problem of data races is to serialize concurrent access to shared state. Mutual exclusion is the most popular technique used to guarantee no two threads can be executing the sensitive region of instructions concurrently. The sequence of operations that must be serialized with respect to all other concurrent executions of that same sequence of operations is called a **critical region.**

Critical regions can be denoted using many mechanisms in today's systems, ranging from language keywords to API calls, and involving such terminology as *locks, mutexes, critical sections, monitors, binary semaphores,* and, recently, *transactions* (see Further Reading, Shavit, Touitou). Each has its own subtle semantic differences. The desired effect, however, is usually roughly the same. So long as all threads use critical regions consistently to access certain data, they can be used to avoid data races.

Some regions support shared modes, for example reader/writer locks, when it is safe for many threads to be reading shared data concurrently. We'll look at examples of this in Chapter 6, Data and Control Synchronization. We will assume strict mutual exclusion for the discussion below.

What happens if multiple threads attempt to enter the same critical region at once? If one thread wants to enter the critical region while another is already executing code inside, it must either wait until the thread leaves or it must occupy itself elsewhere in the meantime, perhaps checking back again sometime later to see if the critical region has become available. The kind of waiting used differs from one implementation to the next, ranging from busy waiting to relying on Windows' support for waiting and signaling. We will return to this topic later.

Let's take a brief example. Given some statement or compound statement of code, `S`, that depends on shared state and may run concurrently on separate threads, we can make use of a critical region to eliminate the possibility of data races.

```
EnterCriticalRegion();
S;
LeaveCriticalRegion();
```

(Note that these APIs are completely fake and simply used for illustration.)

The semantics of the faux `EnterCriticalRegion` API are rather simple: only one thread may enter the region at a time and must otherwise wait for the thread currently inside the region to issue a call to `LeaveCriticalRegion`. This ensures that only one thread may be executing the statement `S` at once in the entire process and, hence, serializes all executions. It appears as if all executions of S happen atomically—provided there is no possibility of concurrent access to the state accessed in `S` outside of critical regions, and that `S` may not fail part-way through—although clearly `S` is not really atomic in the most literal sense of the word.

Using critical regions can solve both data invariant violations illustrated earlier, that is when `S` is `(*a)++`, as shown earlier. Here is the first problematic interleaving we saw, with critical regions added into the picture.

```
T     t1                               t2
0     t1(E): EnterCriticalRegion();
1     t1(0): MOV EAX,[a] #0
2                                      t2(0): EnterCriticalRegion();
3     t1(1): INC,EAX   #1
4     t1(2): MOV [a],EAX #1
5     t1(L): LeaveCriticalRegion();
6                                      t2(0): MOV EAX,[a]  #1
7                                      t2(1): INC,EAX      #2
8                                      t2(2): MOV [a],EAX  #3
9                                      t2(L): LeaveCriticalRegion();
```

In this example, t2 attempts to enter the critical region at time 2. But the thread is not permitted to proceed because t1 is already inside the region and it must wait until time 5 when t1 leaves. The result is that no two threads may be operating on a simultaneously.

As alluded to earlier, any other accesses to a in the program must also be done under the protection of a critical region to preserve atomicity and correctness across the whole program. Should one thread forget to enter the critical region before writing to a, shared state can become corrupted, causing cascading failures throughout the program. For better or for worse, critical regions in today's programming systems are very code-centric rather than being associated with the data accessed inside those regions.

A Generalization of the Idea: Semaphores

The semaphore was invented by E. W. Dijkstra in 1965 as a generalization of the general critical region idea. It permits more sophisticated patterns of data synchronization in which a fixed number of threads are permitted to be inside the critical region simultaneously.

The concept is simple. A semaphore is assigned an **initial count** when created, and, so long as the count remains above 0, threads may continue to decrement the count without waiting. Once the count reaches 0, however, any threads that attempt to decrement the semaphore further must wait until another thread releases the semaphore, increasing the count back above 0. The names Dijkstra invented for these operations are **P,** for the fictitious word **prolaag,** meaning to try to take, and **V,** for the Dutch word **verhoog,** meaning to increase. Since these words are meaningless to those of us who don't speak Dutch, we'll refer to these activities as **taking** and **releasing,** respectively.

A critical region (a.k.a. **mutex**) is therefore just a specialization of the semaphore in which its current count is always either 0 or 1, which is also why critical regions are often called **binary semaphores.** Semaphores with maximum counts of more than 1 are typically called **counting semaphores.** Windows and .NET both offer intrinsic support for semaphore objects. We will explore this support further in Chapter 6, Data and Control Synchronization.

Patterns of Critical Region Usage

The faux syntax shown earlier for entering and leaving critical regions maps closely to real primitives and syntax. We'll generally interchange the terminology enter/leave, enter/exit, acquire/release, and begin/end to mean the same thing. In any case, there is a pair of operations for the critical region: one to enter and one to exit. This syntax might appear to suggest there is only one critical region for the entire program, which is almost never true. In real programs, we will deal with multiple critical regions, protecting different disjoint sets of data, and therefore, we often will have to instantiate, manage, and enter and leave specific critical regions, either by name, object reference, or some combination of both, during execution.

A thread wishing to enter some region 1 does not interfere with a separate region 2 and vice versa. Therefore, we must ensure that all threads consistently enter the correct region when accessing certain data. As an illustration, imagine we have two separate `CriticalRegion` objects, each with `Enter` and `Leave` methods. If two threads tried to increment a shared variable `s_a`, they *must* acquire the same `CriticalRegion` first. If they acquire separate regions, mutual exclusion is not guaranteed and the program has a race.

Here is an example of such a broken program.

```
static int a;
static CriticalRegion cr1, cr2; // initialized elsewhere
void f() { cr1.Enter(); s_a++; cr1.Leave(); }
void g() { cr2.Enter(); s_a++; cr2.Leave(); }
```

This example is flawed because `f` acquires critical region `cr1` and `g` acquires critical region `cr2`. But there are no mutual exclusion guarantees between these separate regions. If one thread runs `f` concurrently with another thread that is running `g`, we will see data races.

Critical regions are most often—but not always—associated with some static lexical scope, in the programming language sense, as shown above. The program enters the region, performs the critical operation, and exits, all occurring on the same stack frame, much like a block scope in C based languages. Keep in mind that this is just a common way to group synchronization sensitive operations under the protection of a critical region and not necessarily a restriction imposed by the mechanisms you will be using. (Many encourage it, however, like C# and VB, which offer keyword support.) It's possible, although often more difficult and much more error prone, to write a critical region that is more dynamic about entering and leaving regions.

```
BOOL f()
{
    if (...)
    {
        EnterCriticalRegion();
        S0; // some critical work
        return TRUE;
    }
    return FALSE;
}

void g()
{
    if (f())
    {
        S1; // more critical work
        LeaveCriticalRegion();
    }
}
```

This style of critical region use is more difficult for a number of reasons, some of which are subtle. First, it is important to write programs that spend as little time as possible in critical regions, for performance reasons. This example inserts some unknown length of instructions into the region (i.e., the function return epilogue of `f` and whatever the caller decides to do before leaving). Synchronization is also difficult enough, and spreading a single region out over multiple functional units adds difficulty where it is not needed.

But perhaps the most notable problem with the more dynamic approach is reacting to an exception from within the region. Normally, programs will want to guarantee the critical region is exited, even if the region is terminated under exceptional circumstances (although not always, as this failure can indicate data corruption). Using a statically scoped block allows you to use things like try/catch blocks to ensure this.

```
EnterCriticalRegion();
__try
{
    S0; S1; // critical work
}
__finally
{
    LeaveCriticalRegion();
}
```

Achieving this control flow for failure and success becomes more difficult with more dynamism. Why might we care so much about guaranteeing release? Well, if we don't always guarantee the lock is released, another thread may subsequently attempt to enter the region and wait indefinitely. This is called an **orphaned lock** and leads to deadlock.

Simply releasing the lock in the face of failure is seldom sufficient, however. Recall that our definition of atomicity specifies two things: that the effects appear instantaneously and that they happen either completely or not at all. If we release the lock immediately when a failure occurs, we may be opening up data corruption to the rest of the program. For example, say we had two shared variables `x` and `y` with some known relationship based invariant; if a region modified `x` but failed before it had a chance to modify `y`, releasing the region would expose the corrupt data and likely lead to additional failure in other parts of the program. Deadlock is generally more debuggable than data corruption, so if the code cannot be written to revert the update to `x` in the face of such a failure, it's often a better idea to leave the region in an acquired state. That said we will use a try/finally type of scheme in examples to ensure the region is exited properly.

Coarse- vs. Fine-Grained Regions

When using a critical region, you must decide what data is to be protected by which critical regions. Coarse- and fine-grained regions are two extreme ends of the spectrum. At one extreme, a single critical region could be used to protect all data in the program; this would force the program to run single-threaded because only one thread could make forward progress at once. At the other extreme, every byte in the heap could be protected by its own critical region; this might alleviate scalability bottlenecks, but would be ridiculously expensive to implement, not to mention impossible to understand, ensure deadlock freedom, and so on. Most systems must strike a careful balance between these two extremes.

The critical region mechanisms available today are defined by regions of program statements in which mutual exclusion is in effect, as shown above, rather than being defined by the data accessed within such regions. The data accessed is closely related to the program logic, but not directly: any given data can be manipulated by many regions of the program and similarly any given region of the program is apt to manipulate different data. This requires many design decisions and tradeoffs to be made around the organization of critical regions.

Programs are often organized as a collection subsystems and composite data structures whose state may be accessed concurrently by many threads at once. Two reasonable and useful approaches to organizing critical regions are as follows:

- **Coarse-grained.** A single lock is used to protect all constituent parts of some subsystem or composite data structure. This is the simplest scheme to get right. There is only one lock to manage and one lock to acquire and release: this reduces the space and time spent on synchronization, and the decision of what comprises a critical region is driven entirely by the need of threads to access some large, easy to identify thing. Much less work is required to ensure safety. This over conservative approach may have a negative impact to scalability due to false sharing, however. False sharing prevents concurrent access to some data unnecessarily, that is it is not necessary to guard access to ensure correctness.
- **Fine-grained.** As a way of improving scalability, we can use a unique lock per constituent piece of data (or some groupings of data), enabling many threads to access disjoint data objects simultaneously. This reduces or eliminates false sharing, allowing threads to achieve greater degrees of concurrency and, hence, better liveness and scalability. The down side to this approach is the increase of number of locks to manage and potentially multiple lock acquisitions needed if more than one data structure must be accessed at once, both of which are bad for space and time complexity. This

strategy also can lead to deadlocks if not used carefully. If there are complex invariant relationships between multiple data structures, it can also become more difficult to eliminate data races.

No single approach will be best for all scenarios. Programs will use a combination of techniques on this spectrum. But as a general rule of thumb, starting with coarse-grained locking to ensure correctness first and fine-tuning the approach to successively use finer-grained regions as scalability requirements demand is an approach that typically leads to a more maintainable, understandable, and bug-free program.

How Critical Regions Are Implemented

Before moving on, let's briefly explore how critical regions might be implemented. There are a series of requirements for any good critical region implementation.

1. The mutual exclusion property holds. That is, there can never be a circumstance in which more than one thread enters the critical region at once.
2. Liveness of entrance and exit of the region is guaranteed. The system as a whole will continue to make forward progress, meaning that the algorithm can cause neither deadlock nor livelock. More formally, given an infinite amount of time, each thread that arrives at the region is guaranteed to eventually enter the region, provided that no thread stays in the region indefinitely.
3. Some reasonable degree of fairness, such that a thread's arrival time at the region somehow gives it (statistical) preference over other threads, is desirable though not strictly required. This does not necessarily dictate that there is a deterministic fairness guarantee—such as first-in, first-out—but often regions strive to be reasonably fair, probabilistically speaking.
4. Low cost is yet another subjective criterion. It is important that entering and leaving the critical region be very inexpensive. Critical regions are often used pervasively in low-level systems software,

such as operating systems, and thus, there is a lot of pressure on the efficiency of the implementation.

As we'll see, there is a progression of approaches that can be taken. In the end, however, we'll see that all modern mutual exclusion mechanisms rely on a combination of atomic compare and swap (CAS) hardware instructions and operating system support. But before exploring that, let's see why hardware support is even necessary. In other words, shouldn't it be easy to implement `EnterCriticalRegion` and `LeaveCriticalRegion` using familiar sequential programming constructs?

The simplest, overly naïve approach won't work at all. We could have a single flag variable, initially 0, which is set to 1 when a thread enters the region and 0 when it leaves. Each thread attempting to enter the region first checks the flag and then, once it sees the flag at 0, sets it to 1.

```
int taken = 0;

void EnterCriticalRegion()
{
    while (taken != 0) /* busy wait */ ;
    taken = 1; // Mark the region as taken.
}

void LeaveCriticalRegion()
{
    taken = 0; // Mark the region as available.
}
```

This is fundamentally very broken. The reason is that the algorithm uses a sequence of reads and writes that aren't atomic. Imagine if two threads read `taken` as 0 and, based on this information, both decide to write 1 into it. Multiple threads would each think it owned the critical region, but both would be running code inside the critical region at once. This is precisely the thing we're trying to avoid with the use of critical regions in the first place!

Before reviewing the state of the art—that is, the techniques all modern critical regions use—we'll take a bit of a historical detour in order to better understand the evolution of solutions to mutual exclusion during the past 40+ years.

Strict Alternation. We might first try to solve this problem with a technique called **strict alternation,** granting ownership to thread 0, which then grants ownership to thread 1 when it is done, which then grants ownership to 2 when it is done, and so on, for N threads, finally returning ownership back to 0 after thread N – 1 has been given ownership and finished running inside the region. This might be implemented in the form of the following code snippet:

```
const int N = ...; // # of threads in the system.
int turn = 0; // Thread 0 gets its turn first.

void EnterCriticalRegion(int i)
{
    while (turn != i) /* busy wait */ ;
    // Someone gave us the turn... we own the region.
}

void LeaveCriticalRegion(int i)
{
    // Give the turn to the next thread (possibly wrapping to 0).
    turn = (i + 1) % N;
}
```

This algorithm ensures mutual exclusion inside the critical region for precisely N concurrent threads. In this scheme, each thread is given a unique identifier in the range [0 . . . N), which is passed as the argument `i` to `EnterCriticalRegion`. The `turn` variable indicates which thread is currently permitted to run inside the critical region, and when a thread tries to enter the critical region, it must wait for its turn to be granted by another thread, in this particular example by busy spinning. With this algorithm, we have to choose someone to be first, so we somewhat arbitrarily decide to give thread 0 its turn first by initializing `turn` to 0 at the outset. Upon leaving the region, each thread simply notifies the next thread that its turn has come up: it does this notification by setting `turn`, either wrapping it back around to 0, if we've reached the maximum number of threads, or by incrementing it by one otherwise.

There is one huge deal breaker with strict alternation: the decision to grant a thread entry to the critical region is not based in any part on the arrival of threads to the region. Instead, there is a predefined ordering: 0, then 1, then . . ., then N – 1, then 0, and so on, which is nonnegotiable and always fixed. This is hardly fair and effectively means a thread that isn't currently in the critical region holds another thread from entering it. This can threaten the liveness of the system because threads must wait to enter the critical region even when there is no thread currently inside of it. This kind of "false contention" isn't a correctness problem per se, but reduces the performance and scalability of any use of it. This algorithm also only works if threads regularly enter and exit the region, since that's the only way to pass on the turn. Another problem, which we won't get to solving for another few pages, is that the critical region cannot accommodate a varying number of threads. It's quite rare to know a priori the number of threads a given region must serve, and even rarer for this number to stay fixed for the duration of a process's lifetime.

Dekker's and Dijkstra's Algorithms (1965). The first widely publicized general solution to the mutual exclusion problem, which did not require strict alternation, was a response submitted by a reader of a 1965 paper by E. W. Dijkstra in which he identified the mutual exclusion problem and called for solutions (see Further Reading, Dijkstra, 1965, Co-operating sequential processes). One particular reader, T. Dekker, submitted a response that met Dijkstra's criteria but that works only for two concurrent threads. It's referred to as "Dekker's algorithm" and was subsequently generalized in a paper by Dijkstra, also in 1965 (see Further Reading, Dijkstra, 1965, Solution of a problem in concurrent programming control), to accommodate N threads.

Dekker's solution works similar to strict alternation, in which turns are assigned, but extends this with the capability for each thread to note an interest in taking the critical region. If a thread desires the region but yet it isn't its turn to enter, it may "steal" the turn if the other thread has not also noted interest (i.e., isn't in the region).

In our sample implementation, we have a shared 2-element array of Booleans, `flags`, initialized to contain `false` values. A thread stores `true` into its respective element (index 0 for thread 0, 1 for thread 1) when it wishes to enter the region, and `false` as it exits. So long as only one thread wants to enter the region, it is permitted to do so. This works because a thread first writes into the shared `flags` array and then checks whether the other thread has also stored into the flags array. We can be assured that if we write true into flags and then read false from the other thread's element that the other thread will see our true value. (Note that modern processors perform out of order reads and writes that actually break this assumption. We'll return to this topic later.)

We must deal with the case of both threads entering simultaneously. The tie is broken by using a shared turn variable, much like we saw earlier. Just as with strict alternation, when both threads wish to enter, a thread may only enter the critical region when it sees turn equal to its own index and that the other thread is no longer interested (i.e., its flags element is false). If a thread finds that both threads wish to enter but it's not its turn, the thread will "back off" and wait by setting its flags element to false and waiting for the turn to change. This lets the other thread enter the region. When a thread leaves the critical region, it just resets its flags element to false and changes the turn.

This entire algorithm is depicted in the following snippet.

```
static bool[] flags = new bool[2];
static int turn = 0;

void EnterCriticalRegion(int i) // i will only ever be 0 or 1
{
    int j = 1 - i;       // the other thread's index
    flags[i] = true;     // note our interest
    while (flags[j])     // wait until the other is not interested
    {
        if (turn == j)   // not our turn, we must back off and wait
        {
            flags[i] = false;
            while (turn == j) /* busy wait */;
            flags[i] = true;
        }
    }
}

void LeaveCriticalRegion(int i)
{
    turn = 1 - i;        // give away the turn
    flags[i] = false;    // and exit the region
}
```

Dijkstra's modification to this algorithm supports N threads. While it still requires N to be determined a priori, it does accommodate systems in which fewer than N threads are active at any moment, which admittedly makes it much more practical.

The implementation is slightly different than Dekker's algorithm. We have a `flags` array of size N, but instead of Booleans it contains a tri-value. Each element can take on one of three values, and in our example, we will use an enumeration: passive, meaning the thread is uninterested in the region at this time; requesting, meaning the thread is attempting to enter the region; and active, which means the thread is currently executing inside of the region.

A thread, upon arriving at the region, notes interest by setting its flag to requesting. It then attempts to "steal" the current turn: if the current turn is assigned to a thread that isn't interested in the region, the arriving thread will set turn to its own index. Once the thread has stolen the turn, it notes that it is actively in the region. Before actually moving on, however, the thread must verify that no other thread has stolen the turn in the meantime and possibly already entered the region, or we could break mutual exclusion. This is verified by ensuring that no other thread's flag is active. If another active thread is found, the arriving thread will back off and go back to a requesting state, continuing the process until it is able to enter the region. When a thread leaves the region, it simply sets its flag to passive.

Here is a sample implementation in C#.

```
const int N = ...; // # of threads that can enter the region.

enum F : int
{
    Passive,
    Requesting,
    Active
}

F[] flags = new F[N]; // all initialized to passive
int turn = 0;

void EnterCriticalRegion(int i)
{
    int j;
    do
    {
            flags[i] = F.Requesting;  // note our interest

            while (turn != i)         // spin until it's our turn
                if (flags[turn] == F.Passive)
                    turn = i;         // steal the turn

            flags[i] = F.Active;      // announce we're entering

            // Verify that no other thread has entered the region.
            for (j = 0;
                 j < N && (j == i || flags[j] != F.Active);
                 j++);
        }
        while (j < N);
    }

    void LeaveCriticalRegion(int i)
    {
        flags[i] = F.Passive;         // just note we've left
    }
```

Note that just as with Dekker's algorithm as written above this code will not work as written on modern compilers and processors due to the high likelihood of out of order execution. This code is meant to illustrate the logical sequence of steps only.

Peterson's Algorithm (1981). Some 16 years after the original Dekker algorithm was published, a simplified algorithm was developed by G. L. Peterson and detailed in his provocatively titled paper, "Myths about the Mutual Exclusion" (see Further Reading, Peterson). It is simply referred to as Peterson's algorithm. In fewer than two pages, he showed a two thread algorithm alongside a slightly more complicated N thread version of his algorithm, both of which were simpler than the 15 years of previous efforts to simplify Dekker and Dijkstra's original proposals.

For brevity's sake, we review just the two thread version here. The shared variables are the same, that is, a `flags` array and a `turn` variable, as in Dekker's algorithm. Unlike Dekker's algorithm, however, a requesting thread immediately gives away the turn to the other thread after setting its `flags` element to `true`. The requesting thread then waits until either the other thread is not in its critical region or until the turn has been given back to the requesting thread.

```
bool[] flags = new bool[2];
int turn = 0;

void EnterCriticalRegion(int i)
{
    flags[i] = true; // note our interest in the region
    turn = 1 - i;    // give the turn away

    // Wait until the region is available or it's our turn.
    while (flags[1 - i] && turn != i) /* busy wait */ ;
}

void LeaveCriticalRegion(int i)
{
    flags[i] = false; // just exit the region
}
```

Peterson's algorithm, just like Dekker's, also satisfies all of the basic mutual exclusion, fairness, and liveness properties outlined above. It is also much simpler, and so it tends to be used more frequently over Dekker's algorithm to teach mutual exclusion.

Lamport's Bakery Algorithm (1974). L. Lamport also proposed an alternative algorithm, and called it the Baker's algorithm (see Further Reading, Lamport, 1974). This algorithm nicely accommodates varying numbers of threads, but has the added benefit that the failure of one thread midway through executing the critical region entrance or exit code does not destroy liveness of the system, as is the case with the other algorithms seen so far. All that is required is the thread must reset its ticket number to 0 and move to its noncritical region. Lamport was interested in applying his algorithm to distributed systems in which such fault tolerance was obviously a critical component of any viable algorithm.

The algorithm is called the "bakery" algorithm because it works a bit like your neighborhood bakery. When a thread arrives, it takes a ticket number, and only when its ticket number is called (or more precisely, those threads with lower ticket numbers have been serviced) will it be permitted to enter the critical region. The implementation properly deals with the edge case in which multiple threads happen to be assigned the same ticket number by using an ordering among the threads themselves—for example, a unique thread identifier, name, or some other comparable property—to break the tie. Here is a sample implementation.

```
const int N = ...; // # of threads that can enter the region.
int[] choosing = new int[N];
int[] number   = new int[N];

void EnterCriticalRegion(int i)
{
    // Let others know we are choosing a ticket number.
    // Then find the max current ticket number and add one.
    choosing[i] = 1;
    int m = 0;
    for (int j = 0; j < N; j++)
    {
        int jn = number[j];
        m = jn > m ? jn : m;
    }
    number[i] = 1 + m;
    choosing[i] = 0;

    for (int j = 0; j < N; j++)
    {
        // Wait for threads to finish choosing.
        while (choosing[j] != 0) /* busy wait */ ;

        // Wait for those with lower tickets to finish. If we took
        // the same ticket number as another thread, the one with the
        // lowest ID gets to go first instead.
        int jn;
        while ((jn = number[j]) != 0 &&
               (jn < number[i] || (jn == number[i] && j < i)))
            /* busy wait */ ;
    }

    // Our ticket was called. Proceed to our region...
}

void LeaveCriticalRegion(int i)
{
    number[i] = 0;
}
```

This algorithm is also unique when compared to previous efforts because threads are truly granted fair entrance into the region. Tickets are assigned on a first-come, first-served basis (FIFO), and this corresponds directly to the order in which threads enter the region.

Hardware Compare and Swap Instructions (Fast Forward to Present Day). Mutual exclusion has been the subject of quite a bit of research. It's easy to take it all for granted given how ubiquitous and fundamental synchronization has become, but nevertheless you may be interested in some of the references to learn more than what's possible to describe in just a few pages (see Further Reading, Raynal).

Most of the techniques shown also share one thing in common. Aside from the bakery algorithm, each relies on the fact that reads and writes from and to natural word-sized locations in memory are atomic on all modern processors. But they specifically do not require atomic sequences of instructions in the hardware. These are truly "lock free" in the most literal sense of the phrase. However, most modern critical regions are not implemented using any of these techniques. Instead, they use intrinsic support supplied by the hardware.

One additional drawback of many of these software only algorithms is that one must know N in advance and that the space and time complexity of each algorithm depends on N. This can pose serious challenges in a system where any number of threads—a number that may only be known at runtime and may change over time—may try to enter the critical region. Windows and the CLR assign unique identifiers to all threads, but unfortunately these identifiers span the entire range of a 4-byte integer. Making N equal to 2^32 would be rather absurd.

Modern hardware supports atomic **compare and swap (CAS)** instructions. These are supported in Win32 and the .NET Framework where they are called **interlocked** operations. (There are many related atomic instructions supported by the hardware. This includes an atomic bit-test-and-set instruction, for example, which can also be used to build critical regions. We'll explore these in more detail in Chapter 10, Memory Models and Lock Freedom.) Using a CAS instruction, software can load, compare, and conditionally store a value, all in one atomic, uninterruptible operation. This is supported in the hardware via a combination of CPU and memory subsystem support, differing in performance and complexity across different architectures.

Imagine we have a CAS API that takes three arguments: (1) a pointer to the address we are going to read and write, (2) the value we wish to place into this location, and (3) the value that must be in the location in order for the operation to succeed. It returns `true` if the comparison succeeded—that is, if the value specified in (3) was found in location (1), and therefore the write of (2) succeeded—or `false` if the operation failed, meaning that the comparison revealed that the value in location (1) was not equal to (3). With such a CAS instruction in hand, we can use an algorithm similar to the first intuitive guess we gave at the beginning of this section:

```
int taken = 0;

void EnterCriticalRegion()
{
    // Mark the region as taken.
    while (!CAS(&taken, 1, 0)) /* busy wait */ ;
}

void LeaveCriticalRegion()
{
    taken = 0; // Mark the region as available.
}
```

A thread trying to enter the critical region continuously tries to write 1 into the taken variable, but only if it reads it as 0 first, atomically. Eventually the region will become free and the thread will succeed in writing the value. Only one thread can enter the region because the CAS operation guarantees that the load, compare, and store sequence is done completely atomically.

This implementation gives us a much simpler algorithm that happens to accommodate an unbounded number of threads, and does not require any form of alternation. It does not give any fairness guarantee or preference as to which thread is given the region next, although it could clearly be extended to do so. In fact, busy waiting indefinitely as shown here is usually a bad idea, and instead, true critical region primitives are often built on top of OS support for waiting, which does have some notion of fairness built in.

Most modern primitive synchronization primitives are built on top of CAS operations. Many other useful algorithms also can be built on top of CAS. For instance, returning to our earlier motivating data race, `(*a)++`, we can use CAS to achieve a race-free and serializable program rather than using a first class critical region. For example:

```
void AtomicIncrement(int * p)
{
    int seen;
    do
    {
        seen = *p;
    }
    while (!CAS(p, seen + 1, seen));
}

// ... elsewhere ...

int a = 0;
AtomicIncrement(&a);
```

If another thread changes the value in location `p` in between the reading of it into the `seen` variable, the CAS operation will fail. The function responds to this failed CAS by just looping around and trying the increment again until the CAS succeeds. Just as with the lock above, there are no fairness guarantees. The thread trying to perform an increment can fail any number of times, but probabilistically it will eventually make forward progress.

The Harsh Reality of Reordering, Memory Models. The discussion leading up to this point has been fairly naïve. With all of the software-only examples of mutual exclusion algorithms above, there is a fundamental problem lurking within. Modern processors execute instructions out of order and modern compilers perform sophisticated optimizations that can introduce, delete, or reorder reads and writes. Reference has already been made to this point. But if you try to write and use a critical region as I've shown, it will likely not work as expected. The hardware-based version (with CAS instructions) will typically work on modern processors because CAS guarantees a certain level of read and write reordering safety.

Here are a few concrete examples where the other algorithms can go wrong.

- In the original strict alternation algorithm, we use a loop that continually rereads `turn`, waiting for it to become equal to the thread's index `i`. Because `turn` is not written in the body of the loop, a compiler may conclude that `turn` is loop invariant and thus hoist the read into a temporary variable before the loop even begins. This will lead to an infinite loop for threads trying to enter a busy critical region. Moreover, a compiler may only do this under some conditions, like when non debug optimizations are enabled. This same problem is present in each of the algorithms shown.

- Dekker's algorithm fundamentally demands that a thread's write to its flags entry happens before the read of its partner's flags variable. If this were not the case, both could read each other's flags variable as false and proceed into the critical region, breaking the mutual exclusion guarantee. This reordering is legal and quite common on all modern processors, rendering this algorithm invalid. Similar requirements are present for many of the reads and writes within the body of the critical region acquisition sequence.
- Critical regions typically have the effect of communicating data written inside the critical region to other threads that will subsequently read the data from inside the critical region. For instance, our earlier example showed each thread executing `a++`. We assumed that surrounding this with a critical region meant that a thread, t2, running later in time than another thread, t1, would always read the value written by t1, resulting in the correct final value. But it's legal for code motion optimizations in the compiler to move reads and writes outside of the critical regions shown above. This breaks concurrency safety and exposes the data race once again. Similarly, modern processors can execute individual reads and writes out of order, and modern cache systems can give the appearance that reads and writes occurred out of order (based on what memory operations are satisfied by what level of the cache).

Each of these issues invalidates one or more of the requirements we sought to achieve at the outset. All modern processors, compilers, and runtimes specify which of these optimizations and reorderings are legal and, most importantly, which are not, through a **memory model.** These guarantees can, in principal, then be relied on to write a correct implementation of a critical region, though it's highly unlikely anybody reading this book will have to take on such a thread. The guarantees vary from compiler to compiler and from one processor to the next (when the compiler's guarantees are weaker than the processor's guarantees), making it extraordinarily difficult to write correct code that runs everywhere.

Using one of the synchronization primitives from Win32 or the .NET Framework alleviates all need to understand memory models. Those primitives should be sufficient for 99.9 percent (or more) of the scenarios most programmers face. For the cases in which these primitives are not up to the thread—which is rare, but can be the case for efficiency reasons—or if you're simply fascinated by the topic, we will explore memory models and some lock free techniques in Chapter 10, Memory Models and Lock Freedom. If you thought that reasoning about program correctness and timings was tricky, just imagine if any of the reads and writes could happen in a randomized order and didn't correspond at all to the order in the program's source.

Coordination and Control Synchronization

If it's not obvious yet, interactions between components change substantially in a concurrent system. Once you have multiple things happening simultaneously, you will eventually need a way for those things to collaborate, either via centrally managed orchestration or autonomous and distributed interactions. In the simplest form, one thread might have to notify another when an important operation has just finished, such as a producer thread placing a new item into a shared buffer for which a consumer thread is waiting. More complicated examples are certainly commonplace, such as when a single thread must orchestrate the work of many subservient threads, feeding them data and instructions to make forward progress on a larger shared problem.

Unlike sequential programs, state transitions happen in parallel in concurrent programs and are thus more difficult to reason. It's not necessarily the fact that things are happening at once that makes concurrency difficult so much as getting the interactions between threads correct. Leslie Lamport said it very well:

> We thought that concurrent systems needed new approaches because many things were happening at once. We have learned instead that . . . the

real leap is from functional to reactive systems. A functional system is one that can be thought of as mapping an input to an output. . . . A (reactive) system is one that interacts in more complex ways with its environment (see Further Reading, Lamport, 1993).

Earlier in this chapter, we saw how state can be shared in order to speed up communication between threads and the burden that implies. The patterns of communication present in real systems often build directly on top of such sharing. In the scenario with a producer thread and a consumer thread mentioned earlier, the consumer may have to wait for the producer to generate an item of interest. Once an item is available, it could be written to a shared memory location that the consumer directly accesses, using appropriate data synchronization to eliminate a class of concurrency hazards. But how does one go about orchestrating the more complex part: waiting, in the case that a consumer arrives before the producer has something of interest, and notification, in the case that a consumer has begun waiting by the time the producer creates that thing of interest? And how does one architect the system of interactions in the most efficient way? These are some topics we will touch on in this section.

Because thread coordination can take on many diverse forms and spans many specific implementation techniques, there are many details to address. As noted in the first chapter, there isn't any "one" correct way to write a concurrent program; instead, there are certain ways of structuring and writing programs that make one approach more appropriate than another. There are quite a few primitives in Win32 and the .NET Framework and design techniques from which to choose. For now we will focus on building a conceptual understanding of the approaches.

State Dependence Among Threads

As we described earlier, programs are comprised of big state machines that are traversed during execution. Threads themselves also are composed of smaller state machines that contribute to the overall state of the program itself. Each carries around some interesting data and performs some number of activities. An activity is just some abstract operation that possibly reads and writes the data and, in doing so, also possibly transitions between states, both local to the thread and global to the program. As we already saw, some level of data synchronization often is needed to ensure invalid states are not reached during the execution of such activities.

It is also worth differentiating between internal and external states, for example, those that are just implementation details of the thread itself versus those that are meant to be observed by other threads running in a system, respectively.

Threads frequently have to interact with other threads running concurrently in the system to accomplish some work, forming a dependency. Once such a dependency exists, a dependent thread will typically have some knowledge of the (externally visible) states the depended-upon thread may transition between. It's even common for a thread to require that another thread is in a specific state before proceeding with an operation. A thread might only transition into such a state with the passing of time, as a result of external stimuli (like a GUI event or incoming network message), via some third thread running concurrently in the system producing some interesting state itself, or some combination of these. When one thread depends on another and is affected by its state changes (such as by reading memory that it has written), the thread is said to be **causally dependent** on the other.

Thinking about control synchronization in abstract terms is often helpful, even if the actual mechanism used is less formally defined. As an example, imagine that there is some set of states SP in which the predicate P will evaluate to true. A thread that requires P to be true before it proceeds is actually just waiting for any of the states in SP to arise. Evaluating the predicate P is really asking the question, "Is the program currently in any such state?" And if the answer is no, then the thread must do one of three things: (1) perform some set of reads and writes to transition the program from its current state to one of those in SP, (2) wait for another concurrent thread in the system to perform this activity' or (3) forget about the requirement and do something else instead.

The one example of waiting we've seen so far is that of a critical region. In the CAS based examples, a thread must wait for any state in which the `taken` variable is false to arise before proceeding to the critical region. Either it is already the case, or the thread trying to enter the region must wait for (2), another thread in the system to enable the state, via leaving the region.

Waiting for Something to Happen

We've encountered the topic of waiting a few times now. As just mentioned, a thread trying to enter a critical region that another thread is already actively running within must wait for it to leave. Many threads may simultaneously try to enter a busy critical region, but only one of them will be permitted to enter at a time. Similarly, control synchronization mechanisms require waiting, for example for an occurrence of an arbitrary event, some data of interest to become available, and so forth. Before moving on to the actual coordination techniques popular in the implementation of control synchronization, let's discuss how it works for a moment.

Busy Spin Waiting. Until now we've shown nothing but busy waiting (a.k.a. spin waiting). This is the simplest (and most inefficient) way to "wait" for some condition to become `true`, particularly in shared memory systems. With busy waiting, the thread simply sits in a loop reevaluating the predicate until it yields the desired answer, continuously rereading shared memory locations.

For instance, if P is some arbitrary Boolean predicate statement and S is some statement that must not execute until P is `true`, we might do this:

```
while (!P) /* busy wait */ ;
S;
```

We say that statement S is guarded by the predicate P. This is an extremely common pattern in control synchronization. Elsewhere there will be a concurrent thread that makes P evaluate to `true` through a series of writes to shared memory.

Although this simple spin wait is sufficient to illustrate the behavior of our guarded region—allowing many code illustrations in this chapter that would have otherwise required an up-front overview of various other platform features—it has some serious problems.

Spinning consumes CPU cycles, meaning that the thread spinning will remain scheduled on the processor until its quantum expires or until some other thread preempts it. On a single processor machine, this is a complete waste because the thread that will make P true can't be run until the spinning thread is switched out. Even on a multiprocessor machine, spinning can lead to noticeable CPU spikes, in which it appears as if some thread is doing real work and making forward progress, but the utilization is just caused by one thread waiting for another thread to run. And the thread remains runnable during the entire wait, meaning that other threads waiting to be scheduled (to perform real work) will have to wait in line behind the waiting thread, which is really not doing anything useful. Last, if evaluating P touches shared memory that is frequently accessed concurrently, continuously re-evaluating the predicate so often will have a negative effect on the performance of the memory system, both for the processor that is actually spinning and also for those doing useful work.

Not only is spin waiting inefficient, but the aggressive use of CPU cycles, memory accesses, and frequent bus communications all consume considerable amounts of power. On battery-powered devices, embedded electronics, and in other power constrained circumstances, a large amount of spinning can be downright annoying, reducing battery time to a fraction of its normal expected range, and it can waste money. Spinning can also increase heat in data centers, increasing air conditioning costs, making it attractive to keep CPU utilization far below 100 percent.

As a simple example of a problem with spinning, I'm sitting on an airplane as I write this paragraph. Moments ago, I was experimenting with various mutual exclusion algorithms that use busy waiting, of the kind we looked at above, when I noticed my battery had drained much more quickly than usual. Why was this so? I was continuously running test case after test case that made use of many threads using busy waits concurrently. At least I was able to preempt this problem. I just stopped running my test cases. But if the developers who created my word processor of choice had chosen to use a plethora of busy waits in the background spellchecking algorithm, it's probable that this particular word processor wouldn't be popular among those who write when traveling. Thankfully that doesn't appear to be the case.

Needless to say, we can do much better.

Real Waiting in the Operating System's Kernel. The Windows OS offers support for true waiting in the form of various kernel objects. There are two kinds of event objects, for example, that allow one thread to wait and have some other thread signal the event (waking the waiter[s]) at some point in the future. There are other kinds of kernel objects, and they are used in the implementation of various other higher-level waiting primitives in Win32 and the .NET Framework. They are all described in Chapter 5, Windows Kernel Synchronization.

When a thread waits, it is put into a wait state (versus a runnable state), which triggers a context switch to remove it from the processor immediately, and ensures that the Windows thread scheduler will subsequently ignore it when considering which thread to run next. This avoids wasting CPU availability and power and permits other threads in the system to make forward progress. Imagine a fictional API `WaitSysCall` that allows threads to wait. Our busy wait loop from earlier might become something like this:

```
if (!P)
    WaitSysCall();
S;
```

Now instead of other threads simply making P true, the thread that makes P true must now take into consideration that other threads might be waiting. It then wakes them with a corresponding call to `WakeSysCall`.

```
Enable(P); // ... make P true ...
WakeSysCall();
```

You probably have picked up a negative outlook on busy waiting altogether. Busy waiting can be used (with care) to improve performance and scalability on multiprocessor machines, particularly for fine-grained concurrency. The reason is subtle, having to do with the cost of context switching, waiting, and waking. Getting it correct requires an intelligent combination of both spinning and true waiting. There are also some architecture specific considerations that you will need to make. (If it's not obvious by now, the spin wait as written above is apt to cause you many problems, so please don't try to use it.) We will explore this topic in Chapter 14, Performance and Scalability.

Continuation Passing as an Alternative to Waiting. Sometimes it's advantageous to avoid waiting altogether. This is for a number of reasons, including avoiding the costs associated with blocking a Windows thread.

But perhaps more fundamentally, waiting can present scheduling challenges. If many threads wait and are awoken nearly simultaneously, they will contend for resources. The details depend heavily on the way in which threads are mapped to threads in your system of choice.

As an alternative to waiting, it is often possible to use continuation passing style (CPS), a popular technique in functional programming environments (see Further Reading, Hoare, 1974). A continuation is an executable closure that represents "the rest" of the computation. Instead of waiting for an event to happen, it is sometimes possible to package up the response to that computation in the form of a closure and to pass it to some API that then assumes responsibility for scheduling the continuation again when the wait condition has been satisfied.

Because neither Windows nor the CLR offers first-class support for continuations, CPS can be difficult to achieve in practice. As we'll see in Chapter 8, Asynchronous Programming Models, the .NET Framework's asynchronous programming model offers a way to pass a delegate to be scheduled in response to an activity completing, as do the Windows and CLR thread pools and various other components. In each case, it's the responsibility of the user of the API to deal with the fact that the remainder of the computation involves a possibly deep callstack at the time of the call. Transforming "the rest" of the computation is, therefore, difficult to do and is ordinarily only a reasonable strategy for applications level programming where components are not reused in various settings.

A Simple Wait Abstraction: Events

The most basic control synchronization primitive is the event, also sometimes referred to as a latch, which is a concrete reification of our fictional `WaitSysCall` and `WakeSysCall` functions shown above. Events are a flexible waiting and notification mechanism that threads can use to coordinate among one another in a less-structured and free-form manner when compared to critical regions and semaphores. Additionally, there can be many such events in a program to wait and signal different interesting circumstances, much like there can be multiple critical regions to protect different portions of shared state.

An event can be in one of two states at a given time: **signaled** or **nonsignaled.** If a thread waits on a nonsignaled event, it does not proceed until the event becomes signaled; otherwise, the thread proceeds right away. Various kinds of events are commonplace, including those that stay signaled permanently (until manually reset to nonsignaled), those that automatically reset back to the nonsignaled state after a single thread waits on it, and so on. In subsequent chapters, we will look at the actual event primitives available to you.

To continue with the previous example of guarding a region of code by some arbitrary predicate P, imagine we have a thread that checks P and, if it is not true, wishes to wait. We can use an event E that is signaled when P is enabled and nonsignaled when it is not. That event internally uses whatever waiting mechanism is most appropriate, most likely involving some amount of spinning plus true OS waiting. Threads enabling and disabling P must take care to ensure that E's state mirrors P correctly.

```
// Consuming thread:
if (!P)
    E.Wait();
S;

// Enabling thread:
Enable(P); // ... make P true ...
E.Set();
```

If it is possible for P to subsequently become false in this example and the event is not automatically reset, we must also allow a thread to reset the event.

```
E.Reset();
Disable(P); // ... make P false ...
```

Each kind of event may reasonably implement different policies for waiting and signaling. One event may decide to wake all waiting threads, while another might decide to wake one and automatically put the event back into a nonsignaled state afterward. Yet another technique may wait for a certain number of calls to `Set` before waking up any waiters.

As we'll see, there are some tricky race conditions in all of these examples that we will have to address. For events that stay signaled or have some degree of synchronization built in, you can get away without extra data synchronization, but most control synchronization situations are not quite so simple.

One Step Further: Monitors and Condition Variables

Although events are a general purpose and flexible construct, the pattern of usage shown here is very common, for example to implement guarded regions. In other words, some event E being signaled represents some interesting program condition, namely some related predicate P being true, and thus the event state mirrors P's state accordingly. To accomplish this reliably, data and control synchronization often are needed together. For instance, the evaluation of the predicate P may depend on shared state, in which case data synchronization is required during its evaluation to ensure safety. Moreover, there are data races, mentioned earlier, that we need to handle. Imagine we support setting and resetting; we must avoid the problematic timing of:

```
t1: Enable(P) -> t2: E.Reset() -> t2: Disable(P) -> t1: E.Set()
```

In this example, t1 enables the predicate P, but before it has a chance to set the event, t2 comes along and disables P. The result is that we wake up waiting threads although P is no longer true. These threads must take care to re-evaluate P after being awakened to avoid proceeding blindly. But unless they use additional data synchronization, this is impossible.

A nice codification of this relationship between state transitions and data and control synchronization was invented in the 1970s (see Further Reading, Hansen; Hoare, 1974) and is called **monitors.** Each monitor implicitly has a critical region and may have one or more **condition variables** associated with it, each representing some condition (like P evaluating to true) for which threads may wish to wait. In this sense, a condition variable is just a fancy kind of event.

All waiting and signaling of a monitor's condition variables must occur within the critical region of the monitor itself, ensuring data race protection. When a thread decides to wait on a condition variable, it implicitly releases ownership of the monitor (i.e., leaves the critical region), waits, and then reacquires it immediately after being woken up by another thread. This release-wait sequence is done such that other threads entering the monitor are not permitted to enter until the releaser has made it known that it is waiting (avoiding the aforementioned data races). There are also usually mechanisms offered to either wake just one waiting thread or all waiting threads when signaling a condition variable.

Keeping with our earlier example, we may wish to enable threads to wait for some arbitrary predicate P to become true. We could represent this with some monitor M (with methods `Enter` and `Leave`) and a condition variable CV (with methods `Wait` and `Set`) to represent the condition in which a state transition is made that enables P. (We could have any number of predicates and associated condition variables for M, but our example happens to use only one.) Our example above, which used events, now may look something like this:

```
// Consuming thread:
M.Enter();
while (!P)
    CV.Wait();
M.Leave();
S; // (or inside the monitor, depending on its contents)

// Enabling thread:
M.Enter();
Enable(P);
CV.Set();
M.Leave();

// Disabling thread:
M.Enter();
Disable(P);
M.Leave();
```

Notice in this example that the thread that disables P has no additional requirements because it does so within the critical region. The next thread that is granted access to the monitor will re-evaluate P and notice that it has become false, causing it to wait on CV. There is something subtle in this program. The consuming thread continually re-evaluates P in a while loop, waiting whenever it sees that it is false. This re-evaluation is necessary to avoid the case where a thread enables P, setting CV, but where another thread "sneaks in" and disables P before the consuming thread has a chance to enter the monitor. There is generally no guarantee, just because the condition variable on which a thread was waiting has become signaled, that such a thread is the next one to enter the monitor's critical region.

Structured Parallelism

Some parallel constructs hide concurrency coordination altogether, so that programs that use them do not need to concern themselves with the low-level events, condition variables, and associated coordination challenges. The most compelling example is data parallelism, where partitioning of the work is driven completely by data layout. The term structured parallelism is used to refer to such parallelism, which typically has well-defined begin and end points.

Some examples of structured parallel constructs follow.

- **Cobegin,** normally takes the form of a block in which each of the contained program statements may execute concurrently. An alternative is an API that accepts an array of function pointers or delegates. The cobegin statement spawns threads to run statements in parallel and returns only once all of these threads have finished, hiding all coordination behind a clean abstraction.
- **Forall,** a.k.a. parallel do loops, in which all iterations of a loop body can run concurrently with one another on separate threads. The statement following the loop itself runs only once all concurrent iterations have finished executing.
- **Futures,** in which some value is bound to a computation that may happen at an unspecified point in the future. The computation may run concurrently, and consumers of the future's value can choose to wait for the value to be computed, without having to know that waiting and control synchronization is involved.

The languages on Windows and the .NET Framework currently do not offer direct support for these constructs, but we will build up a library of them in Chapters 12, Parallel Containers and 13, Data and Task Parallelism.

This library enables higher level concurrent programs to be built with more ease. Appendix B, Parallel Extensions to .NET, also takes a look at the future of concurrency APIs on .NET which contains similar constructs.

Message Passing

In shared memory systems—the dominant concurrent programming model on Microsoft's development platform (including native Win32 and the CLR)—there is no apparent distinction in the programming interface between state that is used to communicate between threads and state that is thread local. The language and library constructs to work with these two very different categories of memory are identical. At the same time, reads from and writes to shared state usually mean very different things than those that work with thread-private state: they are usually meant to instruct concurrent threads about the state of the system so they can react to the state change. The fact that it is difficult to identify operations that work with this special case also makes it difficult to identify where synchronization is required and, hence, to reason about the subtle interactions among concurrent threads.

In message passing systems, all interthread state sharing is encapsulated within the messages sent between threads. This typically requires that state is copied when messages are sent and normally implies handing off ownership of state at the messaging boundary. Logically, at least, this is the same as performing atomic updates in a shared memory system, but is physically quite different. (In fact, using shared memory could be viewed as an optimization for message passing, when it can be proven safe to turn message sends into writes to shared memory. Recent research in operating system design in fact has explored using such techniques [see Further Reading, Aiken, Fahndrich, Hawblitzel, Hunt, Larus].) Due to the copying, message passing in most implementations is less efficient from a performance standpoint. But the overall thread of state management is usually simplified.

The first popular message passing system was proposed by C. A. R. Hoare as his Communicating Sequential Processes (CSP) research (see Further Reading, Hoare, 1978, 1985). In a CSP system, all concurrency is achieved by having independent processes running asynchronously. As they must interact, they send messages to one another, to request or to provide information to one another. Various primitives are supplied to encourage certain communication constructs and patterns, such as interleaving results among many processes, waiting for one of many to produce data of interest, and so on. Using a system like CSP appreciably raises the level of abstraction from thinking about shared memory and informal state transitions to independent actors that communicate through well-defined interfaces.

The CSP idea has shown up in many subsequent systems. In the 1980s, actor languages evolved the ideas from CSP, mostly in the context of LISP and Scheme, for the purpose of supporting richer AI programming such as in the Act1 and Act2 systems (see Further Reading, Lieberman). It turns out that modeling agents in an AI system as independent processes that communicate through messages is not only a convenient way of implementing a system, but also leads to increased parallelism that is bounded only by the number of independent agents running at once and their communication dependencies. Actors in such a system also sometimes are called "active objects" because they are usually ordinary objects but use CSP-like techniques transparently for function calls. The futures abstraction mentioned earlier is also typically used pervasively. Over time, programming systems like Ada and Erlang (see Further Reading, Armstrong) have pushed the envelope of message passing, incrementally pushing more and more usage from academia into industry.

Many CSP-like concurrency facilities have been modeled mathematically. This has subsequently led to the development of the pi-calculus, among others, to formalize the notion of independently communicating agents. This has taken the form of a calculus, which has had recent uses outside of the domain of computer science (see Further Reading, Sangiorgi, Walker).

Windows and the .NET Framework offer only limited support for fine-grained message passing. CLR AppDomains can be used for fine-grained isolation, possibly using CLR Remoting to communicate between objects in separate domains. But the programming model is not nearly as nice as the aforementioned systems in which message passing is first class. Distributed programming systems such as Windows Communication Foundation (WCF) offer message passing support, but are more broadly used for coarse-grained parallel communication. The Coordination and Concurrency

Runtime (CCR), downloadable as part of Microsoft's Robotics SDK (available on MSDN), offers fine-grained message as a first-class construct in the programming model.

As noted in Chapter 1, Introduction, the ideal architecture for building concurrent systems demands a hybrid approach. At a coarse-grain, asynchronous agents are isolated and communicate in a mostly loosely coupled fashion; message passing is great for this. Then at a fine-grain, parallel computations share memory and use data and task parallel techniques.

Where Are We?

In this chapter, we've covered a fair bit of material. We first built up a good understanding of synchronization and time as they relate to concurrent programming and many related topics. Synchronization is important and relevant to all kinds of concurrent programming, no matter whether it is performance or responsiveness motivated, in the form of fine- or coarse-grained concurrency, shared-memory or message-passing based, written in native or managed code, and so on.

Although we haven't yet experimented with enough real mechanisms to build a concurrent program, we're well on our way. The following section, Mechanisms, spans seven chapters and focuses on the building blocks you'll use to build native and managed concurrent Windows programs. We'll start with the schedulable unit of concurrency on Windows: threads.

FURTHER READING

M. Aiken, M. Fahndrich, C. Hawblitzel, G. Hunt, J. R. Larus. Deconstructing Process Isolation. *Microsoft Research Technical Report,* MSR-TR-2006-43 (2006).

J. Armstrong. *Programming Erlang: Software for a Concurrent World* (The Pragmatic Programmers, 2007).

C. Boyapati, B. Liskov, L. Shrira. Ownership Types for Object Encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)* (2003).

P. Brinch Hansen. Structured Multiprogramming. *Communications of the ACM,* Vol. 15, No. 7 (1972).

J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, S. Midkiff. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1999).

E. W. Dijkstra. Co-operating Sequential Processes. In *Programming Languages* (Academic Press, 1965).

E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM,* Vol. 8, No. 9 (1965).

F. Drejhammar, C. Schulte. Implementation Strategies for Single Assignment Variables. *Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS)* (2004).

R. H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS),* Vol. 7, Issue 4 (1985).

M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM Transactions on Programming Languages and Systems,* 12 (3) (1990).

R. Hieb, R. Kent Dybvig. Continuations and Concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1990).

C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM,* Vol. 17, No. 10 (1974).

C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM,* Vol. 21, No. 8 (1978).

C. A. R. Hoare. *Communicating Sequential Processes* (Prentice Hall, 1985).

C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., M. E. Zosel. *The High Performance FORTRAN Handbook* (MIT Press, 1994).

L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM,* Vol. 17, No. 8 (1974).

L. Lamport. Verification and Specification of Concurrent Programs. *A Decade of Concurrency: Reflections and Perspectives, Lecture Notes in Computer Science,* Number 803 (1993).

H. Lieberman. Concurrent Object-oriented Programming in Act 1. *Object-oriented Concurrent Programming* (MIT Press, 1987).

G. L. Peterson. Myths About the Mutual Exclusion Problem. *Inf. Proc. Lett.*, 12, 115–116 (1981).

M. Raynal. *Algorithms for Mutual Exclusion* (MIT Press, 1986).

D. Sangiorgi, D. Walker. *The Pi-Calculus: A Theory of Mobile Processes* (Cambridge University Press, 2003).

N. Shavit, D. Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (1995).

B. Stroustrup. *The C++ Programming Language,* Third Edition (Addison-Wesley, 1997).

PART II
Mechanisms

This page intentionally left blank

3 Threads

INDIVIDUAL PROCESSES ON Windows are sequential by default. Even on a multiprocessor machine, a program (by default) will only use one of them at a time. Running multiple processes at once creates concurrency at a very coarse level. Microsoft Word could be repaginating a document on one processor, while Internet Explorer downloads and renders a Web page on another, all while Windows Indexer is rebuilding search indexes on a third processor. This happens because each application is run inside its own distinct process with (one hopes) little interference between the two (again, one hopes), yielding better responsiveness and overall performance by virtue of running completely concurrently with one another.

The programs running inside of each process, however, are free to introduce additional concurrency. This is done by creating *threads* to run different parts of the program running inside a single program at once. Each Windows process is actually comprised of a single thread by default, but creating more than one in a program enables the OS to schedule many onto separate processors simultaneously. Coincidently, each .NET program is actually multithreaded from the start because the CLR garbage collector uses a separate *finalizer* thread to reclaim resources. As a developer, you are free to create as many additional threads as you want.

Using multiple threads for a single program can be done to run entirely independent parts of a program at once. This is classic **agents style** concurrency and, historically, has been used frequently in server-side programs. Or, you can use threads to break one big task into multiple smaller pieces that can execute concurrently. This is **parallelism** and is increasingly important as commodity hardware continues to increase the number of available processors. Refer back to Chapter 1, Introduction, for a detailed explanation of this taxonomy.

Threads are the fundamental units of schedulable concurrency on the Windows platform and are available to native and managed code alike. This chapter takes a look at the essentials of scheduling and managing concurrency on Windows using threads. The APIs used to access threading in native and managed code are slightly different, but the fundamental architecture and OS support are the same. But before we go into the details, let's precisely define what a thread is and of what it consists. After that, we'll move on to how programs use them.

Threading from 10,001 Feet

A thread is in some sense just a virtual processor. Each runs some program's code as though it were independent from all other virtual processors in the system. There can be fewer, equal, or more threads than real processors on a system at any given moment due (in part) to the multitasking nature of Windows, wherein a user can run many programs at once, and the OS ensures that all such threads get a fair chance at running on the available hardware.

Given that this could be as much a simple definition of an OS process as a thread, clearly there has to be some interesting difference. And there is (on Windows, at least). Processes are the fundamental unit of concurrency on many UNIX OSs because they are generally lighter-weight than Windows processes. A Windows process always consists of at least one thread that runs the program code itself. But one process also may execute multiple threads during the course of its lifetime, each of which shares access to a set of process-wide resources. In short, having many threads in a single process allows one process to do many things at once. The resources shared among threads include a single virtual memory address space, permitting threads to share data and communicate easily by reading from and writing to common addresses and objects in memory. Shared resources also include things associated with the Windows process, such as the handle table and security token information.

Most people get their first taste of threading by accident. Developers use a framework such as ASP.NET that calls their code on multiple threads simultaneously or write some GUI event code in Windows Forms, MFC, or Windows Presentation Foundation, in which there is a strong notion of particular data structures belonging to particular threads. (We discuss this fact and its implications in Chapter 16, Graphical User Interfaces.) These developers often learn about concurrency "the hard way" by accidentally writing unreliable code that crashes or by creating an unresponsive GUI by doing I/O on the GUI thread. Faced with such a situation, people are quick to learn some basic rules of thumb, often without deeply understanding the reasons behind them. This can give people a bad first impression of threads. But while concurrency is certainly difficult, threads are the key to exploiting new hardware, and so it's important to develop a deeper understanding.

What Is a Windows Thread?

We already discussed threads at a high level in previous chapters, but let's begin painting a more detailed picture.

Conceptually speaking, a thread is an execution context that represents in-progress work being performed by a program. A thread isn't a simple, physical thing. Windows must allocate and maintain a kernel object for each thread, along with a set of auxiliary data structures. But as a thread executes, some portion of its logical state is also comprised of hardware state, such as data in the processor's registers. A thread's state is, therefore, distributed among software and hardware, at least when it's running. Given a thread that is running, a processor can continue running it, and given a thread that is not running, the OS has all the information it needs so that it can schedule the thread to run on the hardware again.

Each thread is mapped onto a processor by the Windows thread scheduler, enabling the in-progress work to actually execute. Each thread has an instruction pointer (IP) that refers to the current executing instruction. "Execution" consists of the processor fetching the next instruction, decoding it, and issuing it, one instruction after another, from the thread's code, incrementing the IP after ordinary instructions or adjusting it in other ways as branches and function calls occur. During the execution of some compiled code, program data will be routinely moved into and out of registers from the attached main memory. While these registers physically reside on the processor, some of this volatile state also abstractly belongs to the thread too. If the thread must be paused for any reason, this state will be captured and saved in memory so it can be later restored. Doing this enables the same IP fetch, decode, and issue process to proceed for the thread later as though it were never interrupted. The process of saving or restoring this state from and to the hardware is called a **context switch.**

During a context switch, the volatile processor state, which logically belongs to the thread, is saved in something called a **context.** The context switching behavior is performed entirely by the OS kernel, although the context data structure is available to user-mode in the form of a `CONTEXT` structure. Similarly, when the thread is rescheduled onto a processor, this state must be restored so the processor can begin fetching and executing the thread's instructions again. We'll look at this process in more detail later. Note that contexts arise in a few other places too. For example, when an exception occurs, the OS takes a snapshot of the current context so that exception handling code can inspect the IP and other state when determining how to react. Contexts are also useful when writing debugging and diagnostics tools.

As the processor invokes various function call instructions, a region of memory called the **stack** is used to pass arguments from the caller to the callee (i.e., the function being called), to allocate local variables, to save register values, and to capture return addresses and values. Code on a thread can allocate and store arbitrary data on the stack too. Each thread, therefore, has its own region of stack memory in the process's virtual address space. In truth, each thread actually has two stacks: a user-mode and a kernel-mode stack. Which gets used depends on whether the thread is actively running code in user- or kernel-mode, respectively. Each thread has a well-defined lifetime. When a new process is created, Windows also creates a thread that begins executing that process's entry-point code. A process doesn't execute anything, its threads do. After the magic of a process's first thread being created—handled by the OS's process creation routine—any code inside that process can go ahead and create additional threads. Various system services create threads without you being involved, such as the CLR's garbage collector. When a new thread is created, the OS is told what code to begin executing and away it goes: it handles the bookkeeping, setting the processor's IP, and the code is then subsequently free to create additional threads, and so on.

Eventually a thread will exit. This can happen in a variety of ways—all of which we'll examine soon—including simply returning from the entry-point used to begin the thread's life an unhandled exception, or directly calling one of the platform's thread termination APIs.

The Windows thread scheduler takes care of tracking all of the threads in the system and working with the processor(s) to schedule execution of them. Once a thread has been created, it is placed into a queue of runnable threads and the scheduler will eventually let it run, though perhaps not right away, depending on system load. Windows uses preemptive scheduling for threads, which allows it to forcibly stop a thread from running on a certain processor in order to run some other code when appropriate. Preemption causes a context switch, as explained previously. This happens when a higher priority thread becomes runnable or after a certain period of time (called a **quantum** or a **timeslice**) has elapsed. In either case, the switch only occurs if there aren't enough processors to accommodate both threads in question running simultaneously; the scheduler will always prefer to fully utilize the processors available.

Threads can **block** for a number of reasons: explicit I/O, a hard page fault (i.e., caused by reading or writing virtual memory that has been paged out to disk by the OS), or by using one of the many synchronization primitives detailed in Chapters 5, Windows Kernel Synchronization and 6, Data and Control Synchronization. While a thread blocks, it consumes no processor time or power, allowing other runnable threads to make forward progress in its stead. The act of blocking, as you might imagine, modifies the thread data structure so that the OS thread scheduler knows it has become ineligible for execution and then triggers a context switch. When the condition that unblocks the thread arises, it becomes eligible for execution again, which places it back into the queue of runnable threads, and the scheduler will later schedule it to run using its ordinary thread scheduling algorithms. Sometimes awakened threads are given priority to run again, something called a **priority boost,** particularly if the thread has awakened in response to a GUI event such as a button click. This topic will come up again later.

There are five basic mechanisms in Windows that routinely cause nonlocal transfer of control to occur. That is to say, a processor's IP jumps somewhere very different from what the program code would suggest should happen. The first is a context switch, which we've already seen. The second is exception handling. An exception causes the OS to run various exception filters and handlers in the context of the current executing thread, and, if a handler is found, the IP ends up inside of it.

The next mechanism that causes nonlocal transfer of control is the hardware interrupt. An interrupt occurs when a significant hardware event of interest occurs, like some device I/O completing, a timer expiring, etc., and provides an interrupt dispatch routine the chance to respond. In fact, we've already seen an example of this: preemption based context switches are initiated from a timer based interrupt. While an interrupt borrows the currently executing thread's kernel-mode stack, this is usually not noticeable: the code that runs typically does a small amount of work very quickly and won't run user-mode code at all.

(For what it's worth, in the initial SMP versions of Windows NT, all interrupts ran on processor number 0 instead of on the processor executing the affected thread. This was obviously a scalability bottleneck and required large amounts of interprocessor communication and was remedied for Windows 2000. But I've been surprised by how many people still believe this is how interrupt handling on Windows works, which is why I mention it here.)

Software based interrupts are commonly used in kernel and system code too, bringing us to the fourth and fifth methods: deferred procedure calls (DPCs) and asynchronous procedure calls (APCs). A DPC is just some callback that the OS kernel queues to run later on. DPCs run at a higher Interrupt Request Level (IRQL) than hardware interrupts, which simply means they do not hold up the execution of other higher priority hardware based interrupts should one happen in the middle of the DPC running. If anything meaty has to occur during a hardware interrupt, it usually gets done by the interrupt handler queuing a DPC to execute the hard work, which is guaranteed to run before the thread returns back to user-mode. In fact, this is how preemption based context switches occur. An APC is similar, but can execute user-mode callbacks and only run when the thread has no other useful work to do, indicated by the thread entering something called an **alertable wait.** When, specifically, the thread will perform an alertable wait is unknowable, and it may never occur. Therefore, APCs are normally used for less critical and less time sensitive work, or for cases in which performing an alertable wait is a necessary part of the programming model that users program against. Since APCs also can be queued programmatically from user-mode, we'll return to this topic in Chapter 5, Windows Kernel Synchronization. Both DPCs and APCs can be scheduled across processors to run asynchronously and always run in the context of whatever the thread is doing at the time they execute.

Threads have a plethora of other interesting aspects that we'll examine throughout this chapter and the rest of the book, such as priorities, thread local storage, and a lot of API surface area. Each thread belongs to a single process that has other interesting and relevant data shared among all of its threads—such as the handle table and a virtual memory page table—but the above definition gives us a good roadmap for exploring at a deeper level.

Before all of that, let's review what makes a managed CLR thread different from a native thread. It's a question that comes up time and time again.

What Is a CLR Thread?

A CLR thread is the same thing as a Windows thread—usually. Why, then, is it popular to refer to CLR threads as "managed threads," a very official term that makes them sound entirely different from Windows threads? The answer is somewhat complicated. At the simplest level, it effectively changes nothing for developers writing concurrent software that will run on the CLR. You can think of a thread running managed code as precisely the same thing as a thread running native code, as described above. They really aren't fundamentally different except for some esoteric and exotic situations that are more theoretical than practical.

First, the pragmatic difference: the CLR needs to track each thread that has ever run managed code in order for the CLR to do certain important jobs. The state associated with a Windows thread isn't sufficient. For example, the CLR needs to know about the object references that are live so that the garbage collector can determine which objects in the heap are still live. It does this in part by storing additional per-thread information such as how to find arguments and local variables on the stack. The CLR keeps other information on each managed thread, like event kernel objects that it uses for its own internal synchronization purposes, security, and execution context information, etc. All of these are simply implementation details.

Since the OS doesn't know anything about managed threads, the CLR has to convert OS threads to managed threads, which really just populates the thread's CLR-specific information. This happens in two places. When a new thread is created inside a managed program, it begins life as a managed thread (i.e., CLR-specific state is associated before it is even started). This is easy. If a thread already exists, however—that is it was created in native code and native-managed interoperability is being used—then the first time the thread runs managed code, the CLR will perform this conversion on-demand at the interoperability boundary.

Just to reiterate, all of this is transparent to you as a developer, so these points should make little difference. Knowing about them can come in useful, however, when understanding the CLR architecture and when debugging your programs.

Aside from that very down-to-earth explanation, the CLR has also decoupled itself from Windows threads from day one because there has always been the goal of allowing CLR hosts to override the default mapping of CLR threads directly to Windows threads. A CLR host, like SQL Server or ASP.NET, implements a set of interfaces, allowing it to override various policies, such as memory management, unhandled exception handling, reliability events of interest, and so on. (See Further Reading, Pratschner, for a more detailed overview of these capabilities.) One such overridable policy is the implementation of managed threads. When the CLR 2.0 was being developed, in fact, SQL Server 2005 experimented very seriously with mapping CLR threads to Windows fibers instead of threads, something they called **fiber-mode.** We'll explore in Chapter 9, Fibers, the advantages fibers offer over threads, and how the CLR intended to support them. SQL Server has had a lot of experience in the past employing fiber based user-mode scheduling. We will also discuss We will also discuss a problem called **thread affinity**, which is related to all of this: a piece of work can take a dependency on the identity of the physical OS thread or can create a dependency between the thread and the work itself, which inhibits the platform's ability to decouple the CLR and Windows threads.

Just before shipping the CLR 2.0, the CLR and SQL Server teams decided to eliminate fiber-mode completely, so this whole explanation now has little practical significance other than as a possibly interesting historical account. But, of course, who knows what the future holds? User-mode scheduling offers some promising opportunities for building massively concurrent programs for massively parallel hardware, so the distinction between a CLR thread and a Windows thread may prove to be a useful one. That's really the only reason you might care about the distinction and why I labeled the concern "theoretical" at the outset.

Unless explicitly stated otherwise in the pages to follow, all of the discussions in this chapter pertain to behavior when run normally (i.e., no host) or inside a host that doesn't override the threading behavior. Trying to explain the myriad of possibilities simultaneously would be nearly impossible because the hosting APIs truly enable a large amount of the CLR's behavior to be extended and customized by a host.

Explicit Threading and Alternatives

We'll start our discussion about concurrency mechanisms at the bottom of the architectural stack with the Windows thread management facilities in Win32 and in the .NET Framework. This is called **explicit threading** in this book because you must be explicit about the creation and use of threads. This is a very low-level way to write concurrent software. Sometimes thinking at this low level is unavoidable, particularly for systems-level programming and, sometimes, also in application and library. Thinking about and managing threads is tricky and can quickly steal the focus from solving real algorithmic domain and business problems. You'll find that explicit threading quickly can become intrusive and pervasive in your program's architecture and implementation. Alternatives exist.

Index