16. Paper: Programming with Threads

The attached paper by Andrew Birrell, *Introduction to Programming with Threads*, originally appeared as report 35 of the Systems Research Center, Digital Equipment Corp., Jan. 1989. A somewhat revised version appears as chapter 4 of *Systems Programming with Modula-3*, Greg Nelson ed., Prentice-Hall, 1991, pp 88-118. The current version has been revised to use the C# language (similar to Java) and incorporate some new material.

Read it as an adjunct to the lecture on practical concurrency. It explains how to program with threads, mutexes, and condition variables, and it contains a lot of good advice and examples.

An Introduction to Programming with C# Threads

Andrew D. Birrell

[Revised May, 2005]

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use it. The tutorial sections provide advice on the best ways to use the primitives, give warnings about what can go wrong and offer hints about how to avoid these pitfalls. The paper is aimed at experienced programmers who want to acquire practical expertise in writing concurrent programs. The programming language used is C#, but most of the tutorial applies equally well to other languages with thread support, such as Java.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Threads, Concurrency, Multi-processing, Synchronization

CONTENTS

1. INTRODUCTION

Almost every modern operating system or programming environment provides support for concurrent programming. The most popular mechanism for this is some provision for allowing multiple lightweight "threads" within a single address space, used from within a single program.

Programming with threads introduces new difficulties even for experienced programmers. Concurrent programming has techniques and pitfalls that do not occur in sequential programming. Many of the techniques are obvious, but some are obvious only with hindsight. Some of the pitfalls are comfortable (for example, deadlock is a pleasant sort of bug—your program stops with all the evidence intact), but some take the form of insidious performance penalties.

The purpose of this paper is to give you an introduction to the programming techniques that work well with threads, and to warn you about techniques or interactions that work out badly. It should provide the experienced sequential programmer with enough hints to be able to build a substantial multi-threaded program that works—correctly, efficiently, and with a minimum of surprises.

This paper is a revision of one that I originally published in 1989 [2]. Over the years that paper has been used extensively in teaching students how to program with threads. But a lot has changed in 14 years, both in language design and in computer hardware design. I hope this revision, while presenting essentially the same ideas as the earlier paper, will make them more accessible and more useful to a contemporary audience.

A "thread" is a straightforward concept: a single sequential flow of control. In a high-level language you normally program a thread using procedure calls or method calls, where the calls follow the traditional stack discipline. Within a single thread, there is at any instant a single point of execution. The programmer need learn nothing new to use a single thread.

Having "multiple threads" in a program means that at any instant the program has multiple points of execution, one in each of its threads. The programmer can mostly view the threads as executing simultaneously, as if the computer were endowed with as many processors as there are threads. The programmer is required to decide when and where to create multiple threads, or to accept such decisions made for him by implementers of existing library packages or runtime systems. Additionally, the programmer must occasionally be aware that the computer might not in fact execute all his threads simultaneously.

Having the threads execute within a "single address space" means that the computer's addressing hardware is configured so as to permit the threads to read and write the same memory locations. In a traditional high-level language, this usually corresponds to the fact that the off-stack (global) variables are shared among all the threads of the program. In an object-oriented language such as C# or Java, the static variables of a class are shared among all the threads, as are the instance variables of any objects that the threads share.* Each thread executes on a separate call stack with its own separate local variables. The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer.*

* There is a mechanism in C# (and in Java) for making static fields thread-specific and not shared, but I'm going to ignore that feature in this paper.

Thread facilities are always advertised as being "lightweight". This means that thread creation, existence, destruction and synchronization primitives are cheap enough that the programmer will use them for all his concurrency needs.

Please be aware that I am presenting you with a selective, biased and idiosyncratic collection of techniques. Selective, because an exhaustive survey would be too exhausting to serve as an introduction—I will be discussing only the most important thread primitives, omitting features such as per-thread context information or access to other mechanisms such as NT kernel mutexes or events. Biased, because I present examples, problems and solutions in the context of one particular set of choices of how to design a threads facility—the choices made in the C# programming language and its supporting runtime system. Idiosyncratic, because the techniques presented here derive from my personal experience of programming with threads over the last twenty five years (since 1978)—I have not attempted to represent colleagues who might have different opinions about which programming techniques are "good" or "important". Nevertheless, I believe that an understanding of the ideas presented here will serve as a sound basis for programming with concurrent threads.

Throughout the paper I use examples written in C# [14]. These should be readily understandable by anyone familiar with modern object-oriented languages, including Java [7]. Where Java differs significantly from C#, I try to point this out. The examples are intended to illustrate points about concurrency and synchronization—don't try to use these actual algorithms in real programs.

Threads are not a tool for automatic parallel decomposition, where a compiler will take a visibly sequential program and generate object code to utilize multiple processors. That is an entirely different art, not one that I will discuss here.

2. WHY USE CONCURRENCY?

Life would be simpler if you didn't need to use concurrency. But there are a variety of forces pushing towards its use. The most obvious is the use of multi-processors. With these machines, there really are multiple simultaneous points of execution, and threads are an attractive tool for allowing a program to take advantage of the available hardware. The alternative, with most conventional operating systems, is to configure your program as multiple separate processes, running in separate address spaces. This tends to be expensive to set up, and the costs of communicating between address spaces are often high, even in the presence of shared segments. By using a lightweight multi-threading facility, the programmer can utilize the processors cheaply. This seems to work well in systems having up to about 10 processors, rather than 1000 processors.

* The CLR (Common Language Runtime) used by C# applications introduces the additional concept of "Application Domain", allowing multiple programs to execute in a single hardware address space, but that doesn't affect how your program uses threads.

A second area where threads are useful is in driving slow devices such as disks, networks, terminals and printers. In these cases an efficient program should be doing some other useful work while waiting for the device to produce its next event (such as the completion of a disk transfer or the receipt of a packet from the network). As we will see later, this can be programmed quite easily with threads by adopting an attitude that device requests are all sequential (i.e., they suspend execution of the invoking thread until the request completes), and that the program meanwhile does other work in other threads. Exactly the same remarks apply to higher level slow requests, such as performing an RPC call to a network server.

A third source of concurrency is human users. When your program is performing some lengthy task for the user, the program should still be responsive: exposed windows should repaint, scroll bars should scroll their contents, and cancel buttons should click and implement the cancellation. Threads are a convenient way of programming this: the lengthy task executes in a thread that's separate from the thread processing incoming GUI events; if repainting a complex drawing will take a long time, it will need to be in a separate thread too. In a section 6, I discuss some techniques for implementing this.

A final source of concurrency appears when building a distributed system. Here we frequently encounter shared network servers (such as a web server, a database, or a spooling print server), where the server is willing to service requests from multiple clients. Use of multiple threads allows the server to handle clients' requests in parallel, instead of artificially serializing them (or creating one server process per client, at great expense).

Sometimes you can deliberately add concurrency to your program in order to reduce the latency of operations (the elapsed time between calling a method and the method returning). Often, some of the work incurred by a method call can be deferred, since it does not affect the result of the call. For example, when you add or remove something in a balanced tree you could happily return to the caller before re-balancing the tree. With threads you can achieve this easily: do the re-balancing in a separate thread. If the separate thread is scheduled at a lower priority, then the work can be done at a time when you are less busy (for example, when waiting for user input). Adding threads to defer work is a powerful technique, even on a uni-processor. Even if the same total work is done, reducing latency can improve the responsiveness of your program and the happiness of your users.

3. THE DESIGN OF A THREAD FACILITY

We can't discuss how to program with threads until we agree on the primitives provided by a multi-threading facility. The various systems that support threads offer quite similar facilities, but there is a lot of diversity in the details. In general, there are four major mechanisms: thread creation, mutual exclusion, waiting for events, and some arrangement for getting a thread out of an unwanted long-term wait. To make the discussions in this paper concrete, they're based on the C# thread facility: the "System.Threading" namespace plus the C# "**lock**" statement.

When you look at the "System.Threading" namespace, you will (or should) feel daunted by the range of choices facing you: "Monitor" or "Mutex"; "Wait" or "AutoResetEvent"; "Interrupt" or "Abort"? Fortunately, there's a simple answer: use the "**lock**" statement, the "Monitor" class, and the "Interrupt" method. Those are the features that I'll use for most of the rest of the paper. For now, you should ignore the rest of "System.Threading", though I'll outline it for you section 9.

Throughout the paper, the examples assume that they are within the scope of "**using** System; **using** System.Threading;"

3.1. Thread creation

In C# you create a thread by creating an object of type "Thread", giving its constructor a "ThreadStart" delegate*, and calling the new thread's "Start" method. The new thread starts executing asynchronously with an invocation of the delegate's method. When the method returns, the thread dies. You can also call the "Join" method of a thread: this makes the calling thread wait until the given thread terminates. Creating and starting a thread is often called "forking".

For example, the following program fragment executes the method calls "foo.A()" and "foo.B()" in parallel, and completes only when both method calls have completed. Of course, method "A" might well access the fields of "foo".

```
Thread t = new Thread(new ThreadStart(foo.A));
t.Start();
foo.B();
t.Join();
```

In practice, you probably won't use "Join" very much. Most forked threads are permanent dæmon threads, or have no results, or communicate their results by some synchronization arrangement other than "Join". It's fine to fork a thread but never have a corresponding call of "Join".

3.2. Mutual exclusion

The simplest way that threads interact is through access to shared memory. In an object-oriented language, this is usually expressed as access to variables which are static fields of a class, or instance fields of a shared object. Since threads are running in parallel, the programmer must explicitly arrange to avoid errors arising when more than one thread is accessing the shared variables. The simplest tool for doing this is a primitive that offers mutual exclusion (sometimes called critical sections), specifying for a particular region of code that only one thread can execute there at any time. In the C# design, this is achieved with the class "Monitor" and the language's "**lock**" statement:

```
lock (expression) embedded-statement
```

* A C# "delegate" is just an object constructed from an object and one of its methods. In Java you would instead explicitly define and instantiate a suitable class.

The argument of the “**lock**” statement can be any object: in C# every object inherently implements a mutual exclusion lock. At any moment, an object is either “locked” or “unlocked”, initially unlocked. The “**lock**” statement locks the given object, then executes the contained statements, then unlocks the object. A thread executing inside the “**lock**” statement is said to “hold” the given object’s lock. If another thread attempts to lock the object when it is already locked, the second thread blocks (enqueued on the object’s lock) until the object is unlocked.

The most common use of the “**lock**” statement is to protect the instance fields of an object by locking that object whenever the program is accessing the fields. For example, the following program fragment arranges that only one thread at a time can be executing the pair of assignment statements in the “SetKV” method.

```
class KV {
    string k, v;
    public void SetKV(string nk, string nv) {
        lock (this) { this.k = nk; this.v = nv; }
    }
    …
}
```

However, there are other patterns for choosing which object’s lock protects which variables. In general, you achieve mutual exclusion on a set of variables by associating them (mentally) with a particular object. You then write your program so that it accesses those variables only from a thread which holds that object’s lock (i.e., from a thread executing inside a “**lock**” statement that locked the object). This is the basis of the notion of *monitors*, first described by Tony Hoare [9]. The C# language and its runtime make no restrictions on your choice of which object to lock, but to retain your sanity you should choose an obvious one. When the variables are instance fields of an object, that object is the obvious one to use for the lock (as in the “SetKV” method, above. When the variables are static fields of a class, a convenient object to use is the one provided by the C# runtime to represent the type of the class. For example, in the following fragment of the “KV” class the static field “head” is protected by the object “**typeof**(KV)”. The “**lock**” statement inside the “AddToList” instance method provides mutual exclusion for adding a “KV” object to the linked list whose head is “head”: only one thread at a time can be executing the statements that use “head”. In this code the instance field “next” is also protected by “**typeof**(KV)”.

```
static KV head = null;
KV next = null;

public void AddToList() {
    lock (typeof(KV)) {
        System.Diagnostics.Debug.Assert(this.next == null);
        this.next = head; head = this;
    }
}
```

3.3. Waiting for a condition

You can view an object’s lock as a simple kind of resource scheduling mechanism. The resource being scheduled is the shared memory accessed inside the “**lock**” statement, and the scheduling policy is one thread at a time. But often the programmer needs to express more complicated scheduling policies. This requires use of a mechanism that allows a thread to block until some condition is true. In thread systems pre-dating Java, this mechanism was generally called “condition variables” and corresponded to a separately allocated object [4,13]. In Java and C# there is no separate type for this mechanism. Instead every object inherently implements one condition variable, and the “Monitor” class provides static “Wait”, “Pulse” and “PulseAll” methods to manipulate an object’s condition variable.

```
public sealed class Monitor {
    public static bool Wait(Object obj) { … }
    public static void Pulse(Object obj) { … }
    public static void PulseAll(Object obj) { … }
    …
}
```

A thread that calls “Wait” must already hold the object’s lock (otherwise, the call of “Wait” will throw an exception). The “Wait” operation atomically unlocks the object and blocks the thread[*]. A thread that is blocked in this way is said to be “waiting on the object”. The “Pulse” method does nothing unless there is at least one thread waiting on the object, in which case it awakens at least one such waiting thread (but possibly more than one). The “PulseAll” method is like “Pulse”, except that it awakens *all* the threads currently waiting on the object. When a thread is awoken inside “Wait” after blocking, it re-locks the object, then returns. Note that the object’s lock might not be immediately available, in which case the newly awoken thread will block until the lock is available.

If a thread calls “Wait” when it has acquired the object’s lock multiple times, the “Wait” method releases (and later re-acquires) the lock that number of times.

It’s important to be aware that the newly awoken thread might not be the next thread to acquire the lock: some other thread can intervene. This means that the state of the variables protected by the lock could change between your call of “Pulse” and the thread returning from “Wait”. This has consequences that I’ll discuss in section 4.6.

In systems pre-dating Java, the “Wait” procedure or method took two arguments: a lock and a condition variable; in Java and C#, these are combined into a single argument, which is simultaneously the lock and the wait queue. In terms of the earlier systems, this means that the “Monitor” class supports only one condition variable per lock[†].

[*] This atomicity guarantee avoids the problem known in the literature as the “wake-up waiting” race [18].

[†] However, as we’ll see in section 5.2, it’s not very difficult to add the extra semantics yourself, by defining your own “Condition Variable” class.

The object's lock protects the shared data that is used for the scheduling decision. If some thread A wants the resource, it locks the appropriate object and examines the shared data. If the resource is available, the thread continues. If not, it unlocks the object and blocks, by calling "Wait". Later, when some other thread B makes the resource available it awakens thread A by calling "Pulse" or "PulseAll". For example, we could add the following "GetFromList" method to the class "KV". This method waits until the linked list is non-empty, and then removes the top item from the list.

```
public static KV GetFromList() {
    KV res;
    lock (typeof(KV)) {
        while (head == null) Monitor.Wait(typeof(KV));
        res = head;  head = res.next;
        res.next = null; // for cleanliness
    }
    return res;
}
```

And the following revised code for the "AddToList" method could be used by a thread to add an object onto "head" and wake up a thread that was waiting for it.

```
public void AddToList() {
    lock (typeof(KV)) {
        /* We're assuming this.next == null */
        this.next = head; head = this;
        Monitor.Pulse(typeof(KV));
    }
}
```

3.4. Interrupting a thread

The final part of the thread facility that I'm going to discuss is a mechanism for interrupting a particular thread, causing it to back out of a long-term wait. In the C# runtime system this is provided by the thread's "Interrupt" method:

```
public sealed class Thread {
    public void Interrupt() { … }
    …
}
```

If a thread "t" is blocked waiting on an object (i.e., it is blocked inside a call of "Monitor.Wait"), and another thread calls "t.Interrupt()", then "t" will resume execution by re-locking the object (after waiting for the lock to become unlocked, if necessary) and then throwing "ThreadInterruptedException". (The same is true if the thread has called "Thread.Sleep" or "t.Join".) Alternatively, if "t" is not waiting on an object (and it's not sleeping or waiting inside "t.Join"), then the fact that "Interrupt" has been called is recorded and the thread will throw "ThreadInterruptedException" next time it waits or sleeps.

For example, consider a thread "t" that has called KV's "GetFromList" method, and is blocked waiting for a KV object to become available on the linked list. It seems attractive that if some other thread of the computation decides the "GetFromList" call is no longer interesting (for example, the user clicked CANCEL with his mouse), then "t" should return from "GetFromList". If the thread handling the CANCEL request happens to know the object on which "t" is waiting, then it could just set a flag and call "Monitor.Pulse" on that object. However, much more often the actual call of "Monitor.Wait" is hidden under several layers of abstraction, completely invisible to the thread that's handling the CANCEL request. In this situation, the thread handling the CANCEL request can achieve its goal by calling "t.Interrupt()". Of course, somewhere in the call stack of "t" there should be a handler for "ThreadInterruptedException". Exactly what you should do with the exception depends on your desired semantics. For example, we could arrange that an interrupted call of "GetFromList" returns "**null**":

```
public static KV GetFromList() {
    KV res = null;
    try {
        lock (typeof(KV)) {
            while (head == null) Monitor.Wait(typeof(KV));
            res = head; head = head.next; res.next = null;
        }
    } catch (ThreadInterruptedException) { }
    return res;
}
```

Interrupts are complicated, and their use produces complicated programs. We will discuss them in more detail in section 7.

4. USING LOCKS: ACCESSING SHARED DATA

The basic rule for using mutual exclusion is straightforward: in a multi-threaded program all shared mutable data must be protected by associating it with some object's lock, and you must access the data only from a thread that is holding that lock (i.e., from a thread executing within a "**lock**" statement that locked the object).

4.1. Unprotected data

The simplest bug related to locks occurs when you fail to protect some mutable data and then you access it without the benefits of synchronization. For example, consider the following code fragment. The field "table" represents a table that can be filled with object values by calling "Insert". The "Insert" method works by inserting a non-null object at index "i" of "table", then incrementing "i". The table is initially empty (all "**null**").

```
class Table {
      Object[ ] table = new Object[1000];
      int i = 0;

      public void Insert(Object obj) {
            if (obj != null) {
(1)—              table[i] = obj;
(2)—              i++;
            }
      }

      …
}     // class Table
```

Now consider what might happen if thread A calls "Insert(x)" concurrently with thread B calling "Insert(y)". If the order of execution happens to be that thread A executes (1), then thread B executes (1), then thread A executes (2), then thread B executes (2), confusion will result. Instead of the intended effect (that "x" and "y" are inserted into "table", at separate indexes), the final state would be that "y" is correctly in the table, but "x" has been lost. Further, since (2) has been executed twice, an empty (**null**) slot has been left orphaned in the table. Such errors would be prevented by enclosing (1) and (2) in a "**lock**" statement, as follows.

```
      public void Insert(Object obj) {
            if (obj != null) {
                  lock(this) {
(1)—                    table[i] = obj;
(2)—                    i++;
                  }
            }
      }
```

The "**lock**" statement enforces serialization of the threads' actions, so that one thread executes the statements inside the "**lock**" statement, then the other thread executes them.

The effects of unsynchronized access to mutable data can be bizarre, since they will depend on the precise timing relationship between your threads. In most environments this timing relationship is non-deterministic (because of real-time effects such as page faults, or the use of real-time timer facilities or because of actual asynchrony in a multi-processor system). On a multi-processor the effects can be especially hard to predict and understand, because they depend on details of the computer's memory consistency and caching algorithms.

It would be possible to design a language that lets you explicitly associate variables with particular locks, and then prevents you accessing the variables unless the thread holds the appropriate lock. But C# (and most other languages) provides no support for this: you can choose any object whatsoever as the lock for a particular set of variables. An alternative way to avoid unsynchronized access is to use static or dynamic analysis tools. For example, there are experimental tools [19] that check at runtime which locks are held while accessing each variable, and that warn you if an inconsistent set of locks (or no lock at all) is used. If you have such tools available, seriously consider using them. If not, then you need considerable programmer discipline and careful use of searching and browsing tools. Unsynchronized, or improperly synchronized, access becomes increasingly likely as your locking granularity becomes finer and your locking rules become correspondingly more complex. Such problems will arise less often if you use very simple, coarse grained, locking. For example, use the object instance's lock to protect all the instance fields of a class, and use "**typeof**(theClass)" to protect the static fields. Unfortunately, very coarse grained locking can cause other problems, described below. So the best advice is to make your use of locks be as simple as possible, but no simpler. If you are tempted to use more elaborate arrangements, be entirely sure that the benefits are worth the risks, not just that the program looks nicer.

4.2. Invariants

When the data protected by a lock is at all complicated, many programmers find it convenient to think of the lock as protecting the *invariant* of the associated data. An invariant is a boolean function of the data that is true whenever the associated lock is not held. So any thread that acquires the lock knows that it starts out with the invariant true. Each thread has the responsibility to restore the invariant before releasing the lock. This includes restoring the invariant before calling "Wait", since that also releases the lock.

For example, in the code fragment above (for inserting an element into a table), the invariant is that "i" is the index of the first "**null**" element in "table", and all elements beyond index "i" are "**null**". Note that the variables mentioned in the invariant are accessed only while "this" is locked. Note also that the invariant is not true after the first assignment statement but before the second one—it is only guaranteed when the object is unlocked.

Frequently the invariants are simple enough that you barely think about them, but often your program will benefit from writing them down explicitly. And if they are too complicated to write down, you're probably doing something wrong. You might write down the invariants informally, as in the previous paragraph, or you might use some formal specification language. It's often sensible to have your program explicitly check its invariants. It's also generally a good idea to state explicitly, in the program, which lock protects which fields.

Regardless of how formally you like to think of invariants, you need to be aware of the concept. Releasing the lock while your variables are in a transient inconsistent state will inevitably lead to confusion if it is possible for another thread to acquire the lock while you're in this state.

4.3. Deadlocks involving only locks

In some thread systems [4] your program will deadlock if a thread tries to lock an object that it has already locked. C# (and Java) explicitly allows a thread to lock an object multiple times in a nested fashion: the runtime system keeps track of which thread has locked the object, and how often. The object remains locked (and therefore concurrent access by other threads remains blocked) until the thread has unlocked the object the same number of times.

This "re-entrant locking" feature is a convenience for the programmer: from within a "**lock**" statement you can call another of your methods that also locks the same object, with no risk of deadlock. However, the feature is double-edged: if you call the other method at a time when the monitor invariants are not true, then the other method will likely misbehave. In systems that prohibit re-entrant locking such misbehavior is prevented, being replaced by a deadlock. As I said earlier, deadlock is usually a more pleasant bug than returning the wrong answer.

There are numerous more elaborate cases of deadlock involving just locks, for example:

```
thread A locks object M1;
thread B locks object M2;
thread A blocks trying to lock M2;
thread B blocks trying to lock M1.
```

The most effective rule for avoiding such deadlocks is to have a partial order for the acquisition of locks in your program. In other words, arrange that for any pair of objects {M1, M2}, each thread that needs to have M1 and M2 locked simultaneously does so by locking the objects in the same order (for example, M1 is always locked before M2). This rule completely avoids deadlocks involving only locks (though as we will see later, there are other potential deadlocks when your program uses the "Monitor.Wait" method).

There is a technique that sometimes makes it easier to achieve this partial order. In the example above, thread A probably wasn't trying to modify exactly the same set of data as thread B. Frequently, if you examine the algorithm carefully you can partition the data into smaller pieces protected by separate locks. For example, when thread B tried to lock M1, it might actually want access to data disjoint from the data that thread A was accessing under M1. In such a case you might protect this disjoint data by locking a separate object, M3, and avoid the deadlock. Note that this is just a technique to enable you to have a partial order on the locks (M1 before M2 before M3, in this example). But remember that the more you pursue this hint, the more complicated your locking becomes, and the more likely you are to become confused about which lock is protecting which data, and end up with some unsynchronized access to shared data. (Did I mention that having your program deadlock is almost always a preferable risk to having your program give the wrong answer?)

4.4. Poor performance through lock conflicts

Assuming that you have arranged your program to have enough locks that all the data is protected, and a fine enough granularity that it does not deadlock, the remaining locking problems to worry about are all performance problems.

Whenever a thread is holding a lock, it is potentially stopping another thread from making progress—if the other thread blocks trying to acquire the lock. If the first thread can use all the machine's resources, that is probably fine. But if

the first thread, while holding the lock, ceases to make progress (for example by blocking on another lock, or by taking a page fault, or by waiting for an i/o device), then the total throughput of your program is degraded. The problem is worse on a multi-processor, where no single thread can utilize the entire machine; here if you cause another thread to block, it might mean that a processor goes idle. In general, to get good performance you must arrange that lock conflicts are rare events. The best way to reduce lock conflicts is to lock at a finer granularity; but this introduces complexity and increases the risk of unsynchronized access to data. There is no way out of this dilemma—it is a trade-off inherent in concurrent computation.

The most typical example where locking granularity is important is in a class that manages a set of objects, for example a set of open buffered files. The simplest strategy is to use a single global lock for all the operations: open, close, read, write, and so forth. But this would prevent multiple writes on separate files proceeding in parallel, for no good reason. So a better strategy is to use one lock for operations on the global list of open files, and one lock per open file for operations affecting only that file. Fortunately, this is also the most obvious way to use the locks in an object-oriented language: the global lock protects the global data structures of the class, and each object's lock is used to protect the data specific to that instance. The code might look something like the following.

```
class F {
    static F head = null;   // protected by typeof(F)
    string myName;          // immutable
    F next = null;          // protected by typeof(F)
    D data;                 // protected by "this"

    public static F Open(string name) {
        lock (typeof(F)) {
            for (F f = head; f != null; f = f.next) {
                if (name.Equals(f.myName)) return f;
            }
            // Else get a new F, enqueue it on "head" and return it.
            return …;
        }
    }

    public void Write(F f, string msg) {
        lock (this) {
            // Access "f.data"
        }
    }

}
```

There is one important subtlety in the above example. The way that I chose to implement the global list of files was to run a linked list through the "next" instance field. This resulted in an example where part of the instance data must be protected by the global lock, and part by the per-object instance lock. This is just one of a wide variety of situations where you might choose to protect different fields of an object with different locks, in order to get better efficiency by accessing them simultaneously from different threads.

Unfortunately, this usage has some of the same characteristics as unsynchronized access to data. The correctness of the program relies on the ability to access different parts of the computer's memory concurrently from different threads, without the accesses interfering with each other. The Java memory model specifies that this will work correctly as long as the different locks protect different variables (e.g., different instance fields). The C# language specification, however, is currently silent on this subject, so you should program conservatively. I recommend that you assume accesses to object references, and to scalar values of 32 bits or more (e.g., "**int**" or "**float**") can proceed independently under different locks, but that accesses to smaller values (like "**bool**") might not. And it would be most unwise to access different elements of an array of small values such as "**bool**" under different locks.

There is an interaction between locks and the thread scheduler that can produce particularly insidious performance problems. The scheduler is the part of the thread implementation (often part of the operating system) that decides which of the non-blocked threads should actually be given a processor to run on. Generally the scheduler makes its decision based on a *priority* associated with each thread. (C# allows you to adjust a thread's priority by assigning to the thread's "Priority" property[*].) Lock conflicts can lead to a situation where some high priority thread never makes progress at all, despite the fact that its high priority indicates that it is more urgent than the threads actually running.

This can happen, for example, in the following scenario on a uni-processor. Thread A is high priority, thread B is medium priority and thread C is low priority. The sequence of events is:

```
C is running (e.g., because A and B are blocked somewhere);
C locks object M;
B wakes up and pre-empts C
    (i.e., B runs instead of C since B has higher priority);
B embarks on some very long computation;
A wakes up and pre-empts B (since A has higher priority);
A tries to lock M, but can't because it's still locked by C;
A blocks, and so the processor is given back to B;
B continues its very long computation.
```

The net effect is that a high priority thread (A) is unable to make progress even though the processor is being used by a medium priority thread (B). This state is stable until there is processor time available for the low priority thread C to complete its work and unlock M. This problem is known as "priority inversion".

[*] Recall that thread priority is *not* a synchronization mechanism: a high priority thread can easily get overtaken by a lower priority thread, for example if the high priority threads hits a page fault.

The programmer can avoid this problem by arranging for C to raise its priority before locking M. But this can be quite inconvenient, since it involves considering for each lock which other thread priorities might be involved. The best solution to this problem lies in the operating system's thread scheduler. Ideally, it should artificially raise C's priority while that's needed to enable A to eventually make progress. The Windows NT scheduler doesn't quite do this, but it does arrange that even low priority threads do make progress, just at a slower rate. So C will eventually complete its work and A will make progress.

4.5. Releasing the lock within a "**lock**" statement

There are times when you want to unlock the object in some region of program nested inside a "**lock**" statement. For example, you might want to unlock the object before calling down to a lower level abstraction that will block or execute for a long time (in order to avoid provoking delays for other threads that want to lock the object). C# (but not Java) provides for this usage by offering the raw operations "Enter(m)" and "Exit(m)" as static methods of the "Monitor" class. You must exercise extra care if you take advantage of this. First, you must be sure that the operations are correctly bracketed, even in the presence of exceptions. Second, you must be prepared for the fact that the state of the monitor's data might have changed while you had the object unlocked. This can be tricky if you called "Exit" explicitly (instead of just ending the "**lock**" statement) at a place where you were embedded in some flow control construct such as a conditional clause. Your program counter might now depend on the previous state of the monitor's data, implicitly making a decision that might no longer be valid. So I discourage this paradigm, to reduce the tendency to introduce quite subtle bugs.

Some thread systems, though not C#, allow one other use of separate calls of "Enter(m)" and "Exit(m)", in the vicinity of forking. You might be executing with an object locked and want to fork a new thread to continue working on the protected data, while the original thread continues without further access to the data. In other words, you would like to transfer the holding of the lock to the newly forked thread, atomically. You can achieve this by locking the object with "Enter(m)" instead of a "**lock**" statement, and later calling "Exit(m)" in the forked thread. This tactic is quite dangerous—it is difficult to verify the correct functioning of the monitor. I recommend that you don't do this even in systems that (unlike C#) allow it.

4.6. Lock-free programming

As we have seen, using locks is a delicate art. Acquiring and releasing locks slows your program down, and some inappropriate uses of locks can produce dramatically large performance penalties, or even deadlock. Sometimes programmers respond to these problems by trying to write concurrent programs that are correct without using locks. This practice is general called "lock-free programming". It requires taking advantage of the atomicity of certain primitive operations, or using lower-level primitives such as memory barrier instructions.

With modern compilers and modern machine architectures, this is an exceedingly dangerous thing to do. Compilers are free to re-order actions within the specified formal semantics of the programming language, and will often do so. They do this for simple reasons, like moving code out of a loop, or for more subtle ones, like optimizing use of a processor's on-chip memory cache or taking advantage of otherwise idle machine cycles. Additionally, multi-processor machine architectures have amazingly complex rules for when data gets moved between processor caches and main memory, and how this is synchronized between the processors. (Even if a processor hasn't ever referenced a variable, the variable might be in the processor's cache, if the variable is in the same cache line as some other variable that the processor *did* reference.)

It is, certainly, possible to write correct lock-free programs, and there is much current research on how to do this [10]. But it is very difficult, and it is most unlikely that you'll get it right unless you use formal techniques to verify that part of your program. You also need to be very familiar with your programming language's memory model.[15][*] Looking at your program and guessing (or arguing informally) that it is correct is likely to result in a program that works correctly almost all the time, but very occasionally mysteriously gets the wrong answer.

If you are still tempted to write lock-free code despite these warnings, please first consider carefully whether your belief that the locks are too expensive is accurate. In practice, very few parts of a program or system are actually critical to its overall performance. It would be sad to replace a correctly synchronized program with an almost-correct lock-free one, and then discover that even when your program doesn't break, its performance isn't significantly better.

If you search the web for lock-free synchronization algorithms, the most frequently referenced one is called "Peterson's algorithm"[16]. There are several descriptions of it on university web sites, often accompanied by informal "proofs" of its correctness. In reliance on these, programmers have tried using this algorithm in practical code, only to discover that their program has developed subtle race conditions. The resolution to this paradox lies in observing that the "proofs" rely, usually implicitly, on assumptions about the atomicity of memory operations. Peterson's algorithm as published relies on a memory model known as "sequential consistency"[12]. Unfortunately, no modern multi-processor provides sequential consistency, because the performance penalty on its memory sub-system would be too great. Don't do this.

Another lock-free technique that people often try is called "double-check locking". The intention is to initialize a shared variable shortly before its first use, in such a way that the variable can subsequently be accessed without using a "**lock**" statement. For example, consider code like the following.

[*] The Java language specification has a reasonably precise memory model [7, chapter 17], which says, approximately, that object references and scalar quantities no bigger than 32 bits are accessed atomically. C# does not yet have an accurately defined memory model, which makes lock-free programming even more risky.

```
Foo theFoo = null;

public Foo GetTheFoo() {
    if (theFoo == null) {
        lock (this) {
            if (theFoo == null) theFoo = new Foo();
        }
    }
    return theFoo;
}
```

The programmer's intention here is that the first thread to call "GetTheFoo" will cause the requisite object to be created and initialized, and all subsequent calls of "GetTheFoo" will return this same object. The code is tempting, and it would be correct with the compilers and multi-processors of the 1980's.[*] Today, in most languages and on almost all machines, it is wrong. The failings of this paradigm have been widely discussed in the Java community [1]. There are two problems.

First, if "GetTheFoo" is called on processor A and then later on processor B, it's possible that processor B will see the correct non-null object reference for "theFoo", but will read incorrectly cached values of the instance variables inside "theFoo", because they arrived in B's cache at some earlier time, being in the same cache line as some other variable that's cached on B.

Second, it's legitimate for the compiler to re-order statements within a "**lock**" statement, if the compiler can prove that they don't interfere. Consider what might happen if the compiler makes the initialization code for "new Foo()" be inline, and then re-orders things so that the assignment to "theFoo" happens before the initialization of the instance variable's of "theFoo". A thread running concurrently on another processor might then see a non-null "theFoo" before the object instance is properly initialized.

There are many ways that people have tried to fix this [1]. They're all wrong. The only way you can be sure of making this code work is the obvious one, where you wrap the entire thing in a "**lock**" statement:

```
Foo theFoo = null;

public Foo GetTheFoo() {
    lock (this) {
        if (theFoo == null) theFoo = new Foo();
        return theFoo;
    }
}
```

[*] This discussion is quite different from the corresponding one in the 1989 version of this paper [2]. The speed discrepancy between processors and their main memory has increased so much that it has impacted high-level design issues in writing concurrent programs. Programming techniques that previously were correct have become incorrect.

In fact, today's C# is implemented in a way that makes double-check locking work correctly. But relying on this seems like a very dangerous way to program.

5. USING WAIT AND PULSE: SCHEDULING SHARED RESOURCES

When you want to schedule the way in which multiple threads access some shared resource, and the simple one-at-a-time mutual exclusion provided by locks is not sufficient, you'll want to make your threads block by waiting on an object (the mechanism called "condition variables" in other thread systems).

Recall the "GetFromList" method of my earlier "KV" example. If the linked list is empty, "GetFromList" blocks until "AddToList" generates some more data:

```
lock (typeof(KV)) {
    while (head == null) Monitor.Wait(typeof(KV));
    res = head;  head = res.next; res.next = null;
}
```

This is fairly straightforward, but there are still some subtleties. Notice that when a thread returns from the call of "Wait" its first action after re-locking the object is to check once more whether the linked list is empty. This is an example of the following general pattern, which I strongly recommend for all your uses of condition variables:

```
while (!expression) Monitor.Wait(obj);
```

You might think that re-testing the expression is redundant: in the example above, "AddToList" made the list non-empty before calling "Pulse". But the semantics of "Pulse" do not guarantee that the awoken thread will be the next to lock the object. It is possible that some other consumer thread will intervene, lock the object, remove the list element and unlock the object, before the newly awoken thread can lock the object.* A secondary benefit of this programming rule is that it would allow the implementation of "Pulse" to (rarely) awaken more than one thread; this can simplify the implementation of "Wait", although neither Java nor C# actually give the threads implementer this much freedom.

But the main reason for advocating use of this pattern is to make your program more obviously, and more robustly, correct. With this style it is immediately clear that the expression is true before the following statements are executed. Without it, this fact could be verified only by looking at all the places that might pulse the object. In other words, this programming convention allows you to verify correctness by local inspection, which is always preferable to global inspection.

*The condition variables described here are not the same as those originally described by Hoare [9]. Hoare's design would indeed provide a sufficient guarantee to make this re-testing redundant. But the design given here appears to be preferable, since it permits a much simpler implementation, and the extra check is not usually expensive.

A final advantage of this convention is that it allows for simple programming of calls to "Pulse" or "PulseAll"—extra wake-ups are benign. Carefully coding to ensure that only the correct threads are awoken is now only a performance question, not a correctness one (but of course you must ensure that at least the correct threads are awoken).

5.1. Using "PulseAll"

The "Pulse" primitive is useful if you know that at most one thread can usefully be awoken. "PulseAll" awakens all threads that have called "Wait". If you always program in the recommended style of re-checking an expression after return from "Wait", then the correctness of your program will be unaffected if you replace calls of "Pulse" with calls of "PulseAll".

One use of "PulseAll" is when you want to simplify your program by awakening multiple threads, even though you know that not all of them can make progress. This allows you to be less careful about separating different wait reasons into different queues of waiting threads. This use trades slightly poorer performance for greater simplicity. Another use of "PulseAll" is when you really need to awaken multiple threads, because the resource you have just made available can be used by several other threads.

A simple example where "PulseAll" is useful is in the scheduling policy known as shared/exclusive locking (or readers/writers locking). Most commonly this is used when you have some shared data being read and written by various threads: your algorithm will be correct (and perform better) if you allow multiple threads to read the data concurrently, but a thread modifying the data must do so when no other thread is accessing the data.

The following methods implement this scheduling policy*. Any thread wanting to read your data calls "AcquireShared", then reads the data, then calls "ReleaseShared". Similarly any thread wanting to modify the data calls "AcquireExclusive", then modifies the data, then calls "ReleaseExclusive". When the variable "i" is greater than zero, it counts the number of active readers. When it is negative there is an active writer. When it is zero, no thread is using the data. If a potential reader inside "AcquireShared" finds that "i" is less than zero, it must wait until the writer calls "ReleaseExclusive".

```
class RW {
    int i = 0;                    // protected by "this"

    public void AcquireExclusive() {
        lock (this) {
            while (i != 0) Monitor.Wait(this);
```

* The C# runtime includes a class to do this for you, "ReaderWriterLock". I pursue this example here partly because the same issues arise in lots of more complex problems, and partly because the specification of "ReaderWriterLock" is silent on how or whether its implementation addresses the issues that we're about to discuss. If you care about these issues, you might find that your own code will work better than "ReaderWriterLock".

```
                i = -1;
            }
        }

        public void AcquireShared() {
            lock (this) {
                while (i < 0) Monitor.Wait(this);
                i++;
            }
        }

        public void ReleaseExclusive() {
            lock (this) {
                i = 0;
                Monitor.PulseAll(this);
            }
        }

        public void ReleaseShared() {
            lock (this) {
                i--;
                if (i == 0) Monitor.Pulse(this);
            }
        }

    }   // class RW
```

Using “PulseAll” is convenient in “ReleaseExclusive”, because a terminating writer does not need to know how many readers are now able to proceed. But notice that you could re-code this example using just “Pulse”, by adding a counter of how many readers are waiting, and calling “Pulse” that many times in “ReleaseExclusive”. The “PulseAll” facility is just a convenience, taking advantage of information already available to the threads implementation. Notice that there is no reason to use “PulseAll” in “ReleaseShared”, because we know that at most one blocked writer can usefully make progress.

This particular encoding of shared/exclusive locking exemplifies many of the problems that can occur when waiting on objects, as we will see in the following sections. As we discuss these problems, I will present revised encodings of this locking paradigm.

5.2. Spurious wake-ups

If you keep your use of “Wait” very simple, you might introduce the possibility of awakening threads that cannot make useful progress. This can happen if you use “PulseAll” when “Pulse” would be sufficient, or if you have threads waiting on a single object for multiple different reasons. For example, the shared/exclusive locking methods above have readers and writers both waiting on “this”. This means that when we call “PulseAll” in “ReleaseExclusive”, the effect will be to awaken both classes of blocked threads. But if a reader is first to lock the object, it will increment “i” and prevent an awoken potential writer from making progress until the reader later calls “ReleaseShared”. The cost of this is extra time spent in the thread scheduler, which is typically an expensive place to be. If your problem is such that these spurious wake-ups will be common, then you really want two places to wait—one for readers and one for writers. A terminating reader need only call “Pulse” on the object where writers are waiting; a terminating writer would call “PulseAll” on one of the objects, depending on which was non-empty.

Unfortunately, in C# (and in Java) for each lock we can only wait on one object, the same one that we’re using as the lock. To program around this we need to use a second object, and its lock. It is surprisingly easy to get this wrong, generally by introducing a race where some number of threads have committed to waiting on an object, but they don’t have enough of a lock held to prevent some other thread calling “PulseAll” on that object, and so the wake-up gets lost and the program deadlocks. I believe the following “CV” class, as used in the following revised “RW” example, gets this all right (and you should be able to re-use this exact “CV” class in other situations).[*]

```
class CV {
    Object m;                   // The lock associated with this CV
    public CV(Object m) {       // Constructor
        lock(this) this.m = m;
    }

    public void Wait() {        // Pre: this thread holds “m” exactly once
        bool enter = false;
        // Using the “enter” flag gives clean error handling if m isn’t locked
        try {
            lock (this) {
                Monitor.Exit(m); enter = true; Monitor.Wait(this);
            }
        } finally {
            if (enter) Monitor.Enter(m);
        }
    }

    public void Pulse() {
        lock (this) Monitor.Pulse(this);
    }

    public void PulseAll() {
        lock (this) Monitor.PulseAll(this);
    }

}   // class CV
```

[*] This is more difficult in Java, which does not provide “Monitor.Enter” and “Monitor.Exit”.

We can now revise "RW" to arrange that only waiting readers wait on the main "RW" object, and that waiting writers wait on the auxiliary "wQueue" object. (Initializing "wQueue" is a little tricky, since we can't reference "**this**" when initializing an instance variable.)

```
class RW {
    int i = 0;                    // protected by "this"
    int readWaiters = 0;          // protected by "this"
    CV wQueue = null;

    public void AcquireExclusive() {
        lock (this) {
            if (wQueue == null) wQueue = new CV(this);
            while (i != 0) wQueue.Wait();
            i = -1;
        }
    }

    public void AcquireShared() {
        lock (this) {
            readWaiters++;
            while (i < 0) Monitor.Wait(this);
            readWaiters--;
            i++;
        }
    }

    public void ReleaseExclusive() {
        lock (this) {
            i = 0;
            if (readWaiters > 0) {
                Monitor.PulseAll(this);
            } else {
                if (wQueue != null) wQueue.Pulse();
            }
        }
    }

    public void ReleaseShared() {
        lock (this) {
            i--;
            if (i == 0 && wQueue != null) wQueue.Pulse();
        }
    }

}   // class RW
```

5.3. Spurious lock conflicts

Another potential source of excessive scheduling overhead comes from situations where a thread is awakened from waiting on an object, and before doing useful work the thread blocks trying to lock an object. In some thread designs, this is a problem on most wake-ups, because the awakened thread will immediately try to acquire the lock associated with the condition variable, which is currently held by the thread doing the wake-up. C# avoids this problem in simple cases: calling "Monitor.Pulse" doesn't actually let the awakened thread start executing. Instead, it is transferred to a "ready queue" on the object. The ready queue consists of threads that are ready and willing to lock the object. When a thread unlocks the object, as part of that operation it will take one thread off the ready queue and start it executing.

Nevertheless there is still a spurious lock conflict in the "RW" class. When a terminating writer inside "ReleaseExclusive" calls "wQueue.Pulse(this)", it still has "this" locked. On a uni-processor this would often not be a problem, but on a multi-processor the effect is liable to be that a potential writer is awakened inside "CV.Wait", executes as far as the "**finally**" block, and then blocks trying to lock "m"—because that lock is still held by the terminating writer, executing concurrently. A few microseconds later the terminating writer unlocks the "RW" object, allowing the new writer to continue. This has cost us two extra re-schedule operations, which is a significant expense.

Fortunately there is a simple solution. Since the terminating writer does not access the data protected by the lock after the call of "wQueue.Pulse", we can move that call to after the end of the "**lock**" statement, as follows. Notice that accessing "i" is still protected by the lock. A similar situation occurs in "ReleaseShared".

```
public void ReleaseExclusive() {
    bool doPulse = false;
    lock (this) {
        i = 0;
        if (readWaiters > 0) {
            Monitor.PulseAll(this);
        } else {
            doPulse = (wQueue != null);
        }
    }
    if (doPulse) wQueue.Pulse();
}

public void ReleaseShared() {
    bool doPulse = false;
    lock (this) {
        i--;
        if (i == 0) doPulse = (wQueue != null);
    }
```

```
        if (doPulse) wQueue.Pulse();
    }
```

There are potentially even more complicated situations. If getting the best performance is important to your program, you need to consider carefully whether a newly awakened thread will necessarily block on some other object shortly after it starts running. If so, you need to arrange to defer the wake-up to a more suitable time. Fortunately, most of the time in C# the ready queue used by “Monitor.Pulse” will do the right thing for you automatically.

5.4. Starvation

Whenever you have a program that is making scheduling decisions, you must worry about how fair these decisions are; in other words, are all threads equal or are some more favored? When you are locking an object, this consideration is dealt with for you by the threads implementation—typically by a first-in-first-out rule for each priority level. Mostly, this is also true when you’re using “Monitor.Wait” on an object. But sometimes the programmer must become involved. The most extreme form of unfairness is “starvation”, where some thread will *never* make progress. This can arise in our reader-writer locking example (of course). If the system is heavily loaded, so that there is always at least one thread wanting to be a reader, the existing code will starve writers. This would occur with the following pattern.

```
Thread A calls “AcquireShared”; i := 1;
Thread B calls “AcquireShared”; i := 2;
Thread A calls “ReleaseShared”; i := 1;
Thread C calls “AcquireShared”; i := 2;
Thread B calls “ReleaseShared”; i := 1; … etc.
```

Since there is always an active reader, there is never a moment when a writer can proceed; potential writers will always remain blocked, waiting for “i” to reduce to 0. If the load is such that this is really a problem, we need to make the code yet more complicated. For example, we could arrange that a new reader would defer inside “AcquireShared” if there was a blocked potential writer. We could do this by adding a counter for blocked writers, as follows.

```
int writeWaiters = 0;

public void AcquireExclusive() {
    lock (this) {
        if (wQueue == null) wQueue = new CV(this);
        writeWaiters++;
        while (i != 0) wQueue.Wait();
        writeWaiters--;
        i = -1;
    }
}
```

```
public void AcquireShared() {
    lock (this) {
        readWaiters++;
        if (writeWaiters > 0) {
            wQueue.Pulse();
            Monitor.Wait(this);
        }
        while (i < 0) Monitor.Wait(this);
        readWaiters--;
        i++;
    }
}
```

There is no limit to how complicated this can become, implementing ever more elaborate scheduling policies. The programmer must exercise restraint, and only add such features if they are really required by the actual load on the resource.

5.5. Complexity

As you can see, worrying about these spurious wake-ups, lock conflicts and starvation makes the program more complicated. The first solution of the reader/writer problem that I showed you had 16 lines inside the method bodies; the final version had 39 lines (including the “CV” class), and some quite subtle reasoning about its correctness. You need to consider, for each case, whether the potential cost of ignoring the problem is enough to merit writing a more complex program. This decision will depend on the performance characteristics of your threads implementation, on whether you are using a multi-processor, and on the expected load on your resource. In particular, if your resource is mostly *not* in use then the performance effects will not be a problem, and you should adopt the simplest coding style. But sometimes they are important, and you should only ignore them after explicitly considering whether they are required in your particular situation.

5.6. Deadlock

You can introduce deadlocks by waiting on objects, even although you have been careful to have a partial order on acquiring locks. For example, if you have two resources (call them (1) and (2)), the following sequence of actions produces a deadlock.

```
Thread A acquires resource (1);
Thread B acquires resource (2);
Thread A wants (2), so it calls “Monitor.Wait” to wait for (2);
Thread B wants (1), so it calls “Monitor.Wait” to wait for (1).
```

Deadlocks such as this are not significantly different from the ones we discussed in connection with locks. You should arrange that there is a partial order on the resources managed with condition variables, and that each thread wishing to acquire multiple resources does so according to this order. So, for example, you might decide that (1) is ordered before (2). Then thread B would not be permitted to try to acquire (1) while holding (2), so the deadlock would not occur.

One interaction between locks and waiting on objects is a subtle source of deadlock. Consider the following (extremely simplified) two methods.

```
class GG {
    static Object a = new Object();
    static Object b = new Object();
    static bool ready = false;

    public static void Get() {
        lock (a) {
            lock (b) {
                while (!ready) Monitor.Wait(b);
            }
        }
    }

    public static void Give() {
        lock (a) {
            lock (b) {
                ready = true;
                Monitor.Pulse(b);
            }
        }
    }

}   // class GG
```

If "ready" is "**false**" and thread A calls "Get", it will block on the call of "Monitor.Wait(b)". This unlocks "b", but leaves "a" locked. So if thread B calls "Give", intending to cause a call of "Monitor.Pulse(b)", it will instead block trying to lock "a", and your program will have deadlocked. Clearly, this example is trivial, since the lock of "a" does not protect any data (and the potential for deadlock is quite apparent anyway), but the overall pattern does occur.

Most often this problem occurs when you acquire a lock at one abstraction level of your program then call down to a lower level, which (unknown to the higher level) blocks. If this block can be freed only by a thread that is holding the higher level lock, you will deadlock. It is generally risky to call into a lower level abstraction while holding one of your locks, unless you understand fully the circumstances under which the called method might block. One solution here is to explicitly unlock the higher level lock before calling the lower level abstraction, as we discussed earlier; but as we discussed, this solution has its own dangers. A better solution is to arrange to end the "**lock**" statement before calling down. You can find further discussions of this problem, known as the "nested monitor problem", in the literature [8].

6. USING THREADS: WORKING IN PARALLEL

As we discussed earlier, there are several classes of situations where you will want to fork a separate thread: to utilize a multi-processor; to do useful work while waiting for a slow device; to satisfy human users by working on several actions at once; to provide network service to multiple clients simultaneously; and to defer work until a less busy time.

It is quite common to find straightforward application programs using several threads. For example, you might have one thread doing your main computation, a second thread writing some output to a file, a third thread waiting for (or responding to) interactive user input, and a fourth thread running in background to clean up your data structures (for example, re-balancing a tree). It's also quite likely that library packages you use will create their own threads internally.

When you are programming with threads, you usually drive slow devices through synchronous library calls that suspend the calling thread until the device action completes, but allow other threads in your program to continue. You will find no need to use older schemes for asynchronous operation (such as i/o completion routines). If you don't want to wait for the result of a device interaction, invoke it in a separate thread. If you want to have multiple device requests outstanding simultaneously, invoke them in multiple threads. In general the libraries provided with the C# environment provide appropriate synchronous calls for most purposes. You might find that legacy libraries don't do this (for example, when your C# program is calling COM objects); in those cases, it's usually a good idea to add a layer providing a synchronous calling paradigm, so that the rest of your program can be written in a natural thread-based style.

6.1. Using Threads in User Interfaces

If your program is interacting with a human user, you will usually want it to be responsive even while it is working on some request. This is particularly true of window-oriented interfaces. It is particularly infuriating to the user if his interactive display goes dumb (for example, windows don't repaint or scrollbars don't scroll) just because a database query is taking a long time. You can achieve responsiveness by using extra threads

In the C# Windows Forms machinery your program hears about user interface events by registering delegates as event-handlers for the various controls. When an event occurs, the control calls the appropriate event-handler. But the delegate is called synchronously: until it returns, no more events will be reported to your program, and that part of the user's desktop will appear frozen. So you must decide whether the requested action is short enough that you can safely do it synchronously, or whether you should do the work in a separate thread. A good rule of thumb is that if the event-handler can complete in a length of time that's not significant to a human (say, 30 milliseconds) then it can run synchronously. In all other cases, the event handler should just extract the appropriate parameter data from the user interface state (e.g., the contents of text boxes or radio buttons), and arrange for an asynchronous thread to do the work.

In making this judgment call you need to consider the worst case delay that your code might incur.

When you decide to move the work provoked by a user interface event into a separate thread, you need to be careful. You must capture a consistent view of the relevant parts of the user interface synchronously, in the event handler delegate, before transferring the work to the asynchronous worker thread. You must also take care that the worker thread will desist if it becomes irrelevant (e.g., the user clicks “Cancel”). In some applications you must serialize correctly so that the work gets done in the correct order. Finally, you must take care in updating the user interface with the worker’s results. It’s not legal for an arbitrary thread to modify the user interface state. Instead, your worker thread must use the “Invoke” method of a control to modify its state. This is because the various control instance objects are not thread-safe: their methods cannot be called concurrently. Two general techniques can be helpful. One is to keep exactly one worker thread, and arrange for your event handlers to feed it requests through a queue that you program explicitly. An alternative is to create worker threads as needed, perhaps with sequence numbers on their requests (generated by your event handlers).

Canceling an action that’s proceeding in an asynchronous worker thread can be difficult. In some cases it’s appropriate to use the “Thread.Interrupt” mechanism (discussed later). In other cases that’s quite difficult to do correctly. In those cases, consider just setting a flag to record the cancellation, then checking that flag before the worker thread does anything with its results. A cancelled worker thread can then silently die if it has become irrelevant to the user’s desires. In all cancellation cases, remember that it’s not usually necessary to do all the clean-up synchronously with the cancellation request. All that’s needed is that after you respond to the cancellation request, the user will never see anything that results from the cancelled activity.

6.2. Using Threads in Network Servers

Network servers are usually required to service multiple clients concurrently. If your network communication is based on RPC [3], this will happen without any work on your part, since the server side of your RPC system will invoke each concurrent incoming call in a separate thread, by forking a suitable number of threads internally to its implementation. But you can use multiple threads even with other communication paradigms. For example, in a traditional connection-oriented protocol (such as file transfer layered on top of TCP), you should probably fork one thread for each incoming connection. Conversely, if you are writing a client program and you don’t want to wait for the reply from a network server, invoke the server from a separate thread.

6.3. Deferring Work

The technique of adding threads in order to defer work is quite valuable. There are several variants of the scheme. The simplest is that as soon as your method has done enough work to compute its result, you fork a thread to do the remainder of the work, and then return to your caller in the original thread. This

reduces the latency of your method call (the elapsed time from being called to returning), in the hope that the deferred work can be done more cheaply later (for example, because a processor goes idle). The disadvantage of this simplest approach is that it might create large numbers of threads, and it incurs the cost of calling “Fork” each time. Often, it is preferable to keep a single housekeeping thread and feed requests to it. It’s even better when the housekeeper doesn't need any information from the main threads, beyond the fact that there is work to be done. For example, this will be true when the housekeeper is responsible for maintaining a data structure in an optimal form, although the main threads will still get the correct answer without this optimization. An additional technique here is to program the housekeeper either to merge similar requests into a single action, or to restrict itself to run not more often than a chosen periodic interval.

6.4. Pipelining

On a multi-processor, there is one specialized use of additional threads that is particularly valuable. You can build a chain of producer-consumer relationships, known as a *pipeline*. For example, when thread A initiates an action, all it does is enqueue a request in a buffer. Thread B takes the action from the buffer, performs part of the work, then enqueues it in a second buffer. Thread C takes it from there and does the rest of the work. This forms a three-stage pipeline. The three threads operate in parallel except when they synchronize to access the buffers, so this pipeline is capable of utilizing up to three processors. At its best, pipelining can achieve almost linear speed-up and can fully utilize a multi-processor. A pipeline can also be useful on a uni-processor if each thread will encounter some real-time delays (such as page faults, device handling or network communication).

For example, the following program fragment uses a simple three stage pipeline. The “Queue” class implements a straightforward FIFO queue, using a linked list. An action in the pipeline is initiated by calling the “PaintChar” method of an instance of the “PipelinedRasterizer” class. One auxiliary thread executes in “Rasterizer” and another in “Painter”. These threads communicate through instances of the “Queue” class. Note that synchronization for “QueueElem” objects is achieved by holding the appropriate “Queue” object’s lock.

```
class QueueElem {                          // Synchronized by Queue’s lock
    public Object v;                       // Immutable
    public QueueElem next = null;          // Protected by Queue lock

    public QueueElem(Object v) {
        this.v = v;
    }

}   // class QueueElem
```

```
class Queue {
    QueueElem head = null;              // Protected by “this”
    QueueElem tail = null;              // Protected by “this”

    public void Enqueue(Object v) {     // Append “v” to this queue
        lock (this) {
            QueueElem e = new QueueElem(v);
            if (head == null) {
                head = e;
                Monitor.PulseAll(this);
            } else {
                tail.next = e;
            }
            tail = e;
        }
    }

    public Object Dequeue() {           // Remove first item from queue
        Object res = null;
        lock (this) {
            while (head == null) Monitor.Wait(this);
            res = head.v;
            head = head.next;
        }
        return res;
    }

}   // class Queue

class PipelinedRasterizer {
    Queue rasterizeQ = new Queue();
    Queue paintQ = new Queue();
    Thread t1, t2;
    Font f;
    Display d;

    public void PaintChar(char c) {
        rasterizeQ.Enqueue(c);
    }

    void Rasterizer() {
        while (true) {
            char c = (char)(rasterizeQ.Dequeue());
            // Convert character to a bitmap …
            Bitmap b = f.Render(c);
            paintQ.Enqueue(b);
        }
    }
```

```
    void Painter() {
        while (true) {
            Bitmap b = (Bitmap)(paintQ.Dequeue());
            // Paint the bitmap onto the graphics device …
            d.PaintBitmap(b);
        }
    }

    public PipelinedRasterizer(Font f, Display d) {
        this.f = f;
        this.d = d;
        t1 = new Thread(new ThreadStart(this.Rasterizer));
        t1.Start();
        t2 = new Thread(new ThreadStart(this.Painter));
        t2.Start();
    }
}   // class PipelinedRasterizer
```

There are two problems with pipelining. First, you need to be careful about how much of the work gets done in each stage. The ideal is that the stages are equal: this will provide maximum throughput, by utilizing all your processors fully. Achieving this ideal requires hand tuning, and re-tuning as the program changes. Second, the number of stages in your pipeline determines statically the amount of concurrency. If you know how many processors you have, and exactly where the real-time delays occur, this will be fine. For more flexible or portable environments it can be a problem. Despite these problems, pipelining is a powerful technique that has wide applicability.

6.5. The impact of your environment

The design of your operating system and runtime libraries will affect the extent to which it is desirable or useful to fork threads. The libraries that are most commonly used with C# are reasonably thread-friendly. For example, they include synchronous input and output methods that suspend only the calling thread, not the entire program. Most object classes come with documentation saying to what extent it’s safe to call methods concurrently from multiple threads. You need to note, though, that very many of the classes specify that their static methods are thread-safe, and their instance methods are not. To call the instance methods you must either use your own locking to ensure that only one thread at a time is calling, or in many cases the class provides a “Synchronized” method that will create a synchronization wrapper around an object instance.

You will need to know some of the performance parameters of your threads implementation. What is the cost of creating a thread? What is the cost of keeping a blocked thread in existence? What is the cost of a context switch? What is the cost of a “**lock**” statement when the object is *not* locked? Knowing these, you will be able to decide to what extent it is feasible or useful to add extra threads to your program.

6.6. Potential problems with adding threads

You need to exercise a little care in adding threads, or you will find that your program runs slower instead of faster.

If you have significantly more threads ready to run than there are processors, you will usually find that your program's performance degrades. This is partly because most thread schedulers are quite slow at making general re-scheduling decisions. If there is a processor idle waiting for your thread, the scheduler can probably get it there quite quickly. But if your thread has to be put on a queue, and later swapped into a processor in place of some other thread, it will be more expensive. A second effect is that if you have lots of threads running they are more likely to conflict over locks or over the resources managed by your condition variables.

Mostly, when you add threads just to improve your program's structure (for example driving slow devices, or responding to user interface events speedily, or for RPC invocations) you will not encounter this problem; but when you add threads for performance purposes (such as performing multiple actions in parallel, or deferring work, or utilizing multi-processors), you will need to worry whether you are overloading the system.

But let me stress that this warning applies only to the threads that are ready to run. The expense of having threads blocked waiting on an object is usually less significant, being just the memory used for scheduler data structures and the thread stack. Well-written multi-threaded applications often have quite a large number of blocked threads (50 is not uncommon).

In most systems the thread creation and termination facilities are not cheap. Your threads implementation will probably take care to cache a few terminated thread carcasses, so that you don't pay for stack creation on each fork, but nevertheless creating a new thread will probably incur a total cost of about two or three re-scheduling decisions. So you shouldn't fork too small a computation into a separate thread. One useful measure of a threads implementation on a multi-processor is the smallest computation for which it is profitable to fork a thread.

Despite these cautions, be aware that my experience has been that programmers are as likely to err by creating too few threads as by creating too many.

7. USING INTERRUPT: DIVERTING THE FLOW OF CONTROL

The purpose of the "Interrupt" method of a thread is to tell the thread that it should abandon what it is doing, and let control return to a higher level abstraction, presumably the one that made the call of "Interrupt". For example, on a multi-processor it might be useful to fork multiple competing algorithms to solve the same problem, and when the first of them completes you abort the others. Or you might embark on a long computation (e.g., a query to a remote database server), but abort it if the user clicks a CANCEL button. Or you might want to clean up an object that uses some daemon threads internally.

For example, we could add a "Dispose" method to "PipelinedRasterizer" to terminate its two threads when we've finished using the "PipelinedRasterizer" object. Notice that unless we do this the "PipelinedRasterizer" object will never be garbage collected, because it is referenced by its own dæmon threads.[*]

```
class PipelinedRasterizer: IDisposable {

    public void Dispose() {
        lock(this) {
            if (t1 != null) t1.Interrupt();
            if (t2 != null) t2.Interrupt();
            t1 = null; t2 = null;
        }
    }

    void Rasterizer() {
        try {
            while (true) {
                char c = (char)(rasterizeQ.Dequeue());
                // Convert character to a bitmap …
                Bitmap b = f.Render(c);
                paintQ.Enqueue(b);
            }
        } catch (ThreadInterruptedException) { }
    }

    void Painter() {
        try {
            while (true) {
                Bitmap b = (Bitmap)(paintQ.Dequeue());
                // Paint the bitmap onto the graphics device …
                d.PaintBitmap(b);
            }
        } catch (ThreadInterruptedException) { }
    }

    …
}   // class PipelineRasterizer
```

There are times when you want to interrupt a thread that is performing a long computation but making no calls of "Wait". The C# documentation is a little vague about how to do this, but most likely you can achieve this effect if the long computation occasionally calls "Thread.Sleep(0)". Earlier designs such as Java and Modula included mechanisms designed specifically to allow a thread to poll to

[*] The garbage collector could notice that if the only reference to an object is from threads that are not accessible externally and that are blocked on a wait with no timeout, then the object and the threads can be disposed of. Sadly, real garbage collectors aren't that clever.

see whether it has an interrupt pending (i.e., whether a call of "Wait" would throw the "Interrupted" exception).

Modula also permitted two sorts of wait: alertable and non-alertable. This allowed an area of your program to be written without concern for the possibility of a sudden exception arising. In C# all calls of "Monitor.Wait" are interruptible (as are the corresponding calls in Java), and so to be correct you must either arrange that all calls of "Wait" are prepared for the "Interrupted" exception to be thrown, or you must verify that the Interrupt method will not be called on threads that are performing those waits. This shouldn't be too much of an imposition on your program, since you already needed to restore monitor invariants before calling "Wait". However you do need to be careful that if the "Interrupted" exception is thrown then you release any resources being held in the stack frames being unwound, presumably by writing appropriate "**finally**" statements.

The problem with thread interrupts is that they are, by their very nature, intrusive. Using them will tend to make your program less well structured. A straightforward-looking flow of control in one thread can suddenly be diverted because of an action initiated by another thread. This is another example of a facility that makes it harder to verify the correctness of a piece of program by local inspection. Unless alerts are used with great restraint, they will make your program unreadable, unmaintainable, and perhaps incorrect. I recommend that you very rarely use interrupts, and that the "Interrupt" method should be called only from the abstraction where the thread was created. For example, a package should not interrupt a caller's thread that happens to be executing inside the package. This convention allows you to view an interrupt as an indication that the thread should terminate completely, but cleanly.

There are often better alternatives to using interrupts. If you know which object a thread is waiting on, you can more simply prod it by setting a Boolean flag and calling "Monitor.Pulse". A package could provide additional entry points whose purpose is to prod a thread blocked inside the package on a long-term wait. For example, instead of implementing "PipelinedRasterizer.Dispose" with the "Interrupt" mechanism we could have added a "Dispose" method to the "Queue" class, and called that.

Interrupts are most useful when you don't know exactly what is going on. For example, the target thread might be blocked in any of several packages, or within a single package it might be waiting on any of several objects. In these cases an interrupt is certainly the best solution. Even when other alternatives are available, it might be best to use interrupts just because they are a single unified scheme for provoking thread termination.

Don't confuse "Interrupt" with the quite distinct mechanism called "Abort", which I'll describe later. Only "Interrupt" lets you interrupt the thread at a well-defined place, and it's the only way the thread will have any hope of restoring the invariants on its shared variables.

8. ADDITIONAL TECHNIQUES

Most of the programming paradigms for using threads are quite simple. I've described several of them earlier; you will discover many others as you gain experience. A few of the useful techniques are much less obvious. This section describes some of these less obvious ones.

8.1. Up-calls

Most of the time most programmers build their programs using layered abstractions. Higher level abstractions call only lower level ones, and abstractions at the same level do not call each other. All actions are initiated at the top level.

This methodology carries over quite well to a world with concurrency. You can arrange that each thread will honor the abstraction boundaries. Permanent dæmon threads within an abstraction initiate calls to lower levels, but not to higher levels. The abstraction layering has the added benefit that it forms a partial order, and this order is sufficient to prevent deadlocks when locking objects, without any additional care from the programmer.

This purely top-down layering is not satisfactory when actions that affect high-level abstractions can be initiated at a low layer in your system. One frequently encountered example of this is on the receiving side of network communications. Other examples are user input, and spontaneous state changes in peripheral devices.

Consider the example of a communications package dealing with incoming packets from a network. Here there are typically three or more layers of dispatch (corresponding to the data link, network and transport layers in OSI terminology). If you try to maintain a top-down calling hierarchy, you will find that you incur a context switch in each of these layers. The thread that wishes to receive data from its transport layer connection cannot be the thread that dispatches an incoming Ethernet packet, since the Ethernet packet might belong to a different connection, or a different protocol (for example, UDP instead of TCP), or a different protocol family altogether (for example, DECnet instead of IP). Many implementers have tried to maintain this layering for packet reception, and the effect has been uniformly bad performance—dominated by the cost of context switches.

The alternative technique is known as "up-calls" [6]. In this methodology, you maintain a pool of threads willing to receive incoming data link layer (e.g., Ethernet) packets. The receiving thread dispatches on Ethernet protocol type and calls *up* to the network layer (e.g., DECnet or IP), where it dispatches again and calls up to the transport layer (e.g., TCP), where there is a final dispatch to the appropriate connection. In some systems, this up-call paradigm extends into the application. The attraction here is high performance: there are no unnecessary context switches.

You do pay for this performance. As usual, the programmer's task has been made more complicated. Partly this is because each layer now has an up-call interface as well as the traditional down-call interface. But also the synchronization problem has become more delicate. In a purely top-down system it is fine to hold one layer's lock while calling a lower layer (unless the lower layer might block on an object waiting for some condition to become true and thus cause the sort of nested monitor deadlock we discussed earlier). But when you make an up-call you can easily provoke a deadlock involving just the locks—if an up-calling thread holding a lower level lock needs to acquire a lock in a higher level abstraction (since the lock might be held by some other thread that's making a down-call). In other words, the presence of up-calls makes it more likely that you will violate the partial order rule for locking objects. To avoid this, you should generally avoid holding a lock while making an up-call (but this is easier said than done).

8.2. Version stamps and caching

Sometimes concurrency can make it more difficult to use cached information. This can happen when a separate thread executing at a lower level in your system invalidates some information known to a thread currently executing at a higher level. For example, information about a disk volume might change—either because of hardware problems or because the volume has been removed and replaced. You can use up-calls to invalidate cache structures at the higher level, but this will not invalidate state held locally by a thread. In the most extreme example, a thread might obtain information from a cache, and be about to call an operation at the lower level. Between the time the information comes from the cache and the time that the call actually occurs, the information might have become invalid.

A technique known as "version stamps" can be useful here. In the low level abstraction you maintain a counter associated with the true data. Whenever the data changes, you increment the counter. (Assume the counter is large enough to never overflow.) Whenever a copy of some of the data is issued to a higher level, it is accompanied by the current value of the counter. If higher level code is caching the data, it caches the associated counter value too. Whenever you make a call back down to the lower level, and the call or its parameters depend on previously obtained data, you include the associated value of the counter. When the low level receives such a call, it compares the incoming value of the counter with the current truth value. If they are different it returns an exception to the higher level, which then knows to re-consider its call. (Sometimes, you can provide the new data with the exception). Incidentally, this technique is also useful when maintaining cached data across a distributed system.

8.3. Work crews (thread pools)

There are situations that are best described as "an embarrassment of parallelism", when you can structure your program to have vastly more concurrency than can be efficiently accommodated on your machine. For example, a compiler implemented using concurrency might be willing to use a separate thread to compile each method, or even each statement. In such situations, if you create one thread for each action you will end up with so many threads that the scheduler becomes quite inefficient, or so many that you have numerous lock conflicts, or so many that you run out of memory for the stacks.

Your choice here is either to be more restrained in your forking, or to use an abstraction that will control your forking for you. Such an abstraction was first described in Vandevoorde and Roberts' paper [20], and is available to C# programmers through the methods of the "ThreadPool" class:

```
public sealed class ThreadPool { … }
```

The basic idea is to enqueue requests for asynchronous activity and to have a fixed pool of threads that perform the requests. The complexity comes in managing the requests, synchronizing between them, and coordinating the results.

Beware, though, that the C# "ThreadPool" class uses entirely static methods – there is a single pool of threads for your entire application. This is fine if the tasks you give to the thread pool are purely computational; but if the tasks can incur delays (for example, by making a network RPC call) then you might well find the built-in abstraction inadequate.

An alternative proposal, which I have not yet seen in practice, is to implement "Thread.Create" and "Thread.Start" in such a way that they defer actually creating the new thread until there is a processor available to run it. This proposal has been called "lazy forking".

8.4. Overlapping locks

The following are times when you might use more than one lock for some data.

Sometimes when it's important to allow concurrent read access to some data, while still using mutual exclusion for write access, a very simple technique will suffice. Use two (or more) locks, with the rule that any thread holding just one lock can read the data, but if a thread wants to modify the data it must acquire both (or all) locks.

Another overlapping lock technique is often used for traversing a linked list. Have one lock for each element, and acquire the lock for the next element before releasing the one for the current element. This requires explicit use of the "Enter" and "Exit" methods, but it can produce dramatic performance improvements by reducing lock conflicts.

9. ADVANCED C# FEATURES

Throughout this paper I've restricted the discussion to a small subset of the "System.Threading" namespace. I strongly recommend that you restrict your programming to this subset (plus the "System.Threading.ReaderWriterLock" class) as much as you can. However, the rest of the namespace was defined for a purpose, and there are times when you'll need to use parts of it. This section outlines those other features.

There are variants of "Monitor.Wait" that take an additional argument. This argument specifies a timeout interval: if the thread isn't awoken by "Pulse", "PulseAll" or "Interrupt" within that interval, then the call of "Wait" returns anyway. In such a situation the call returns "**false**".

There is an alternative way to notify a thread that it should desist: you call the thread's "Abort" method. This is much more drastic and disruptive than "Interrupt", because it throws an exception at an arbitrary and ill-defined point (instead of just at calls of "Wait", "Sleep" or "Join"). This means that in general it will be impossible for the thread to restore invariants. It will leave your shared data in ill-defined states. The only reasonable use of "Abort" is to terminate an unbounded computation or a non-interruptible wait. If you have to resort to "Abort" you will need to take steps to re-initialize or discard affected shared variables.

Several classes in "System.Threading" correspond to objects implemented by the kernel. These include "AutoResetEvent", "ManualResetEvent", "Mutex", and "WaitHandle". The only real benefit you'll get from using these is that they can be used to synchronize between threads in multiple address spaces. There will also be times when you need them to synchronize with legacy code.

The "Interlocked" class can be useful for simple atomic increment, decrement or exchange operations. Remember that you can only do this if your invariant involves just a single variable. "Interlocked" won't help you when more than one variable is involved.

10. BUILDING YOUR PROGRAM

A successful program must be useful, correct, live (as defined below) and efficient. Your use of concurrency can impact each of these. I have discussed quite a few techniques in the previous sections that will help you. But how will you know if you have succeeded? The answer is not clear, but this section might help you towards discovering it.

The place where concurrency can affect usefulness is in the design of the interfaces to library packages. You should design your classes with the assumption that your callers will be using multiple threads. This means that you must ensure that all the methods are thread re-entrant (i.e., they can be called by multiple threads simultaneously), even if this means that each method immediately acquires a single shared lock. You must not return results in shared static variables, nor in shared allocated storage. Your methods should be synchronous, not returning until their results are available—if your caller wants to do other work meanwhile, he can do it in other threads. Even if you don't presently have any multi-threaded clients for a class, I strongly recommend that you follow these guidelines so that you will avoid problems in the future.

Not everyone agrees with the preceding paragraph. In particular, most of the instance methods in the libraries provided with C# (those in the CLR and the .Net platform SDK) are not thread-safe. They assume that an object instance is called from only one thread at a time. Some of the classes provide a method that will return a correctly synchronized object instance, but many do not. The reason for this design decision is that the cost of the "**lock**" statement was believed to be too high to use it where it might be unnecessary. Personally, I disagree with this: the cost of shipping an incorrectly synchronized program can be very much higher. In my opinion, the correct solution is to implement the "**lock**" statement in a way that is sufficiently cheap. There are known techniques to do this [5].

By "correct" I mean that if your program eventually produces an answer, it will be the one defined by its specification. Your programming environment is unlikely to provide much help here beyond what it already provides for sequential programs. Mostly, you must be fastidious about associating each piece of data with one (and only one) lock. If you don't pay constant attention to this, your task will be hopeless. If you use locks correctly, and you always use wait for objects in the recommended style (re-testing the Boolean expression after returning from "Wait"), then you are unlikely to go wrong.

By "live", I mean that your program will eventually produce an answer. The alternatives are infinite loops or deadlock. I can't help you with infinite loops. I believe that the hints of the preceding sections will help you to avoid deadlocks. But if you fail and produce a deadlock, it should be quite easy to detect.

By "efficient", I mean that your program will make good use of the available computer resources, and therefore will produce its answer quickly. Again, the hints in the previous sections should help you to avoid the problem of concurrency adversely affecting your performance. And again, your programming environment needs to give you some help. Performance bugs are the most insidious of problems, since you might not even notice that you have them. The sort of information you need to obtain includes statistics on lock conflicts (for example, how often threads have had to block in order to acquire this lock, and how long they then had to wait for an object) and on concurrency levels (for example, what was the average number of threads ready to execute in your program, or what percentage of the time were "n" threads ready).

In an ideal world, your programming environment would provide a powerful set of tools to aid you in achieving correctness, liveness and efficiency in your use of concurrency. Unfortunately in reality the most you're likely to find today is the usual features of a symbolic debugger. It's possible to build much more powerful tools, such as specification languages with model checkers to verify what your program does [11], or tools that detect accessing variables without the appropriate locks [19]. So far, such tools are not widely available, though that is something I hope we will be able to fix.

One final warning: don't emphasize efficiency at the expense of correctness. It is much easier to start with a correct program and work on making it efficient, than to start with an efficient program and work on making it correct.

11. CONCLUDING REMARKS

Writing concurrent programs has a reputation for being exotic and difficult. I believe it is neither. You need a system that provides you with good primitives and suitable libraries, you need a basic caution and carefulness, you need an armory of useful techniques, and you need to know of the common pitfalls. I hope that this paper has helped you towards sharing my belief.

Butler Lampson, Mike Schroeder, Bob Stewart, and Bob Taylor caused me to write this paper (in 1988), and Chuck Thacker persuaded me to revise it for C# (in 2003). If you found it useful, thank them.

REFERENCES

1. BACON, A. *et al.* "The 'double-checked locking is broken' declaration." At *http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html* (undated).
2. BIRRELL, A. "An Introduction to programming with threads." SRC Research Report 35. Digital Equipment Corporation (Jan. 1989).
3. BIRRELL, A., AND NELSON, B. "Implementing remote procedure calls." *ACM Trans. Comput. Syst.* **2**, 1 (Feb. 1984), 39-59.
4. BIRRELL, A., GUTTAG, J., HORNING, J. AND LEVIN, R. "Synchronization primitives for a multiprocessor: a formal specification." In *Proceedings of the 11th Symposium on Operating System Principles* (Nov. 1987), 94-102.
5. BURROWS, M. "Implementing unnecessary mutexes." In *Computer Systems: Theory, Technology, and Applications.* Springer (2004), 51-57.
6. CLARK, D. "The structuring of systems using up-calls." In *Proceedings of the 10th Symposium on Operating System Principles* (Dec. 1985), 171-180.
7. GOSLING, G. *et al. The Java language specification second edition.* Addison-Wesley (June 2000).
8. HADDON, B. "Nested monitor calls." *Operating Systems Review* **11**, 4 (Oct. 1977), 18-23.
9. HOARE, C.A.R. "Monitors: An operating system structuring concept." *Commun. ACM* **17**, 10 (Oct. 1974), 549-557.
10. HARRIS, T. AND FRASER, K. "Language support for lightweight transactions." In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Oct. 2003), 388-402.
11. LAMPORT, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley (July 2002).
12. LAMPORT, L. "How to make a multiprocessor computer that correctly executes multiprocess programs." *IEEE Trans. Computers* **28**, 9 (Sep. 1979) 690-691.
13. LAMPSON, B AND REDELL, D. "Experience with processes and monitors in Mesa." *Commun. ACM* **23**, 2 (Feb.1980), 105-117.
14. MICROSOFT CORPORATION. *C# language specifications.* Microsoft Press (2001).
15. MOSBERGER, D. "Memory consistency models." *Operating Systems Review* **27**, 1 (Jan. 1993) 18-26.
16. PETERSON, G. "Myths about the mutual exclusion problem." *Inform. Process. Lett.* **12**, 3 (June 1981), 115-116.
17. ROVNER, P. "Extending Modula-2 to build large, integrated systems." *IEEE Software* **3**, 6 (Nov. 1986), 46-57.
18. SALTZER, J. "Traffic control in a multiplexed computer system." Th., MAC-TR-30, MIT, Cambridge, Mass. (July 1966).
19. SAVAGE, S, *et al.* "Eraser: a dynamic race detector for concurrent programs." *ACM Trans. Comput. Syst.* **15**, 4 (Apr. 1997), 391-411.
20. VANDEVOORDE, M. AND ROBERTS, E. Workcrews: an abstraction for controlling parallelism. *SRC Research Report 42.* Digital Equipment Corporation (April 1989).