Concurrency

What Every Dev Must Know About Multithreaded Apps

Vance Morrison

This article discusses:

- Multithreading and the shared memory threading model
- Races and how concurrent access can break invariants
- Locks as the the standard solution to races
- When locks are needed
- How to use locks and understand costs
- How locks can get in each other's way

This article uses the following technologies:
.NET Framework

Contents

T
en years ago only hard-core systems programmers worried about the intricacies of writing correct code in the presence of more than one thread of execution. Most programmers stuck with sequential programs to avoid the problem altogether. Now, however, multiprocessor machines are becoming commonplace. Soon, programs that are not multithreaded will be at a disadvantage because they will not be using a large part of the available computing power.

Unfortunately, writing correct, multithreaded programs is not easy. Programmers are simply not accustomed to the idea that other threads might be changing memory out from underneath them. Worse, when a mistake is made, the program will continue to work most of the time. Only under stressful (production) conditions will the bugs manifest, and only rarely will there be enough information at the point of failure to effectively debug the application. Figure 1 summarizes the important differences between sequential and multithreaded programs. As it shows, it really pays to get multithreaded programs right the first time.

I have three goals in this article. First, I'll show that multithreaded programs are not that arcane. The fundamental requirement for writing correct programs is the same whether the program is sequential or multithreaded: all code in the program must protect any invariants that other parts of the program need. Second, I'll show you that while this principle is simple enough, preserving program invariants is much more difficult in the multithreaded case. Examples that are trivial in a sequential environment have surprising subtlety in a multithreaded environment. Finally, I'll also show you how to deal with the subtle problems that can creep into your multithreaded programs. This guidance amounts to strategies for being very methodical about protecting program invariants, which, as the table in Figure 2 shows, is more complicated in the multithreaded case. There are a number of reasons that this is more complicated when using multithreading and I will explain them in the following sections.

Threads and Memory

At its heart, multithreaded programming seems simple enough. Instead of having just one processing unit doing work sequentially, you have two or more executing simultaneously. Because the processors might be real hardware or might be implemented by time-multiplexing a single processor, the term "thread" is used instead of processor. The tricky part of multithreaded programming is how threads communicate with one another.

The most commonly deployed multithreaded communication model is called the shared memory model. In this model all threads have access to the same pool of shared memory, as shown in Figure 3. This model has the advantage that multithreaded programs are programmed in much the same way as sequential programs. That advantage, however, is also its biggest problem. The model does not distinguish between memory that is being used strictly for thread local use (like most locally declared variables), and memory that is being used to communicate with other threads (like some global variables and heap memory). Since memory that is potentially shared needs to be treated much more carefully than memory that is local to a thread, it's very easy to make mistakes.

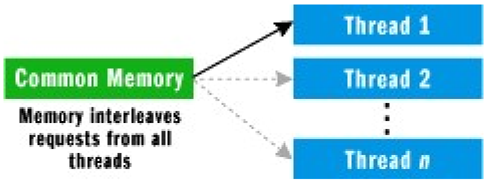

Figure 3 Shared Memory Threading Model

Races

Imagine a program that processes requests and has a global counter, totalRequests, that is incremented after every request is completed. As you can see, the code that does this for a sequential program is trivial:

```
totalRequests = totalRequests + 1
```

If, however, the program has multiple threads servicing requests and updating totalRequests, there is a problem. The compiler might compile the increment operation into the following machine code:

```
MOV EAX, [totalRequests]  // load memory for totalRequests into register
INC EAX                   // update register
MOV [totalRequests], EAX  // store updated value back to memory
```

Consider what would happen if two threads ran this code simultaneously. As shown in Figure 4, both threads will load the same value for totalRequests, both will increment it, and both will store it back to totalRequests. At the end of that sequence two threads have processed a request; however, the value in totalRequests will only be one larger than it was previously. Clearly, this isn't what you want. Bugs like this, which occur because of bad timing between threads, are called races.

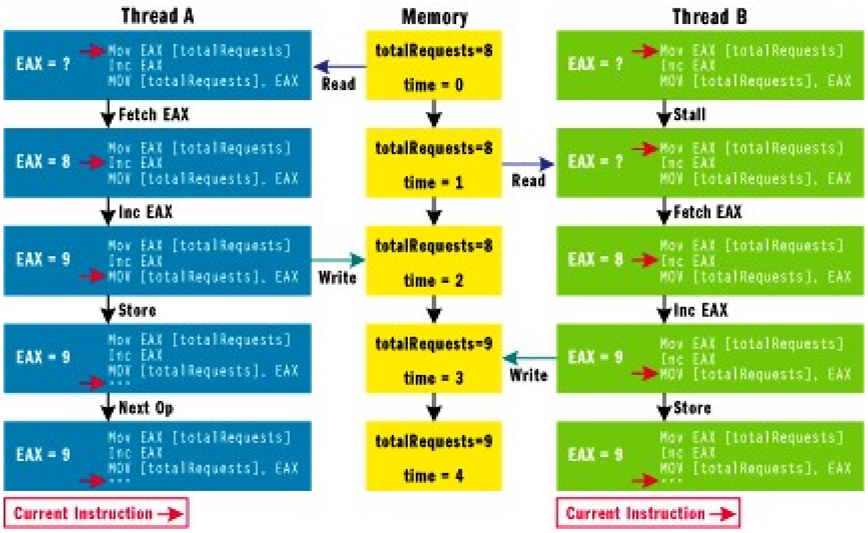

Figure 4 Anatomy of a Race

While this example may seem trivial, the general structure for the problem is the same as for more complicated real-life races. There are four conditions needed for a race to be possible.

The first condition is that there are memory locations that are accessible from more than one thread. Typically, locations are global/static variables (as is the case of totalRequests) or are heap memory reachable from global/static variables.

The second condition is that there is a property associated with these shared memory locations that is needed for the program to function correctly. In this case, the property is that totalRequests accurately represents the total number of times any thread has executed any part of the increment statement. Typically, the property needs to hold true (that is, totalRequests must hold an accurate count) before an update occurs for the update to be correct.

The third condition is that the property does not hold during some part of the actual update. In this particular case, from the time totalRequests is fetched until the time it is stored, totalRequests does not satisfy the invariant.

The fourth and final condition that must occur for a race to happen is that another thread accesses the memory when the invariant is broken, thereby causing incorrect behavior.

Locks

The most common way of preventing races is to use locks to prevent other threads from accessing memory associated with an invariant while it is broken. This removes the fourth condition I mentioned, thus making a race impossible.

The most common kind of lock goes by many different names. It is sometimes called a monitor, a critical section, a mutex, or a binary semaphore, but regardless of the name, it provides the same basic functionality. The lock provides Enter and Exit methods, and once a thread calls Enter, all attempts by other threads to call Enter will cause the other threads to block (wait) until a call to Exit is made. The thread that called Enter is the owner of the lock, and it is considered a programming error if Exit is called by a thread that is not the owner of the lock. Locks provide a mechanism for ensuring that only one thread can execute a particular region of code at any given time.

In the Microsoft® .NET Framework locks are implemented by the System.Threading.Monitor class. The Monitor class is a bit unusual because it does not define instances. This is because lock functionality is effectively implemented by System.Object, and so any object can be a lock. Here is how to use a lock to fix the race associated with totalRequests:

```
static object totalRequestsLock = new Object();  // executed at program
                                                 // init
...
System.Threading.Monitor.Enter(totalRequestsLock);
totalRequests = totalRequests + 1;
System.Threading.Monitor.Exit(totalRequestsLock);
```

While this code does fix the race, it can introduce another problem. If an exception happens while the lock is held, then Exit will not be called. This will cause all other threads that try to run this code to block forever. In many programs, any exception would be considered fatal to the program and thus what happens in that case is not interesting. However, for programs that want to be able to recover from exceptions, the solution can be made more robust by putting the call to Exit in a finally clause:

```
System.Threading.Monitor.Enter(totalRequestsLock);
try {
    totalRequests = totalRequests + 1;
} finally {
    System.Threading.Monitor.Exit(totalRequestsLock);
}
```

This pattern is common enough that both C# and Visual Basic® .NET have a special statement construct to support it. The following C# code is equivalent to the try/finally statement just shown:

```
lock(totalRequestsLock) {
    totalRequests = totalRequests + 1;
}
```

Personally, I am ambivalent about the lock statement. On the one hand, it is convenient shorthand. However, it can lull programmers into a false sense of confidence that they are writing robust code. Remember, the locked region was introduced because an important program invariant did not hold there. If an exception is thrown in that region, it is likely that the invariant was broken at the time the exception was thrown. Allowing the program to continue without trying to fix the invariant is a bad idea.

In the totalRequests example, there is no useful cleanup to do so the lock statement is appropriate. The lock statement would also be appropriate if everything its body did was read-only. In general, however, more cleanup work needs to be done if an exception happens. In this case a lock statement does not add much value since an explicit try/finally statement will be needed anyway.

Using Locks Properly

Most programmers have had some interaction with races and have seen simple examples of how to use locks to prevent them. Without additional explanation, however, the examples themselves don't drive home important principles needed to use locks effectively in real-world programs.

The first important observation is that locks provide mutual exclusion for regions of code, but generally programmers want to protect regions of memory. In the totalRequests example, the goal is to make certain an invariant holds true on totalRequests (a memory location). However, to do so, you actually place a lock around a region of code (the increment of totalRequests). This provides mutual exclusion over totalRequests because it is the only code that references totalRequests. If there was other code that updates totalRequests without entering the lock, then you would not have mutual exclusion for the memory, and consequently the code would have race conditions.

This leads to the following principle. For a lock to provide mutual exclusion for a region of memory, no writes to that memory can occur without entering the same lock. In a properly designed program, associated with every lock is a region of memory for which it provides mutual exclusion. Unfortunately, there is no obvious artifact of the code that makes this association clear, and yet this information is absolutely critical for anyone reasoning about the multithreaded behavior of the program.

As a consequence of this, every lock should have a specification associated with it that documents the precise region of memory (set of data structures) for which it provides mutual exclusion. In the totalRequests example, totalRequestsLock protects the totalRequests variable and nothing else. In real programs, locks tend to protect larger regions, such as a whole data structure, several related data structures, or all memory reachable from a data structure. Sometimes a lock only protects part of a data structure (like the hash bucket chain of a hash table), but whatever the region is, it is critical that the programmer document it. With the specification written down, it is possible to methodically validate that the lock is entered before the associated memory is updated. Most races are caused by the failure to consistently enter the correct lock before accessing the related memory, so this audit is well worth the time.

Once each lock has a precise specification of what memory it protects, check whether any of the regions protected by different locks overlap. While overlap is not strictly incorrect, it should be avoided because it is not useful for memory associated with two different locks to overlap. Consider what would happen when memory common to both locks needs to be updated. Which one should be taken? The possibilities include the following:

Enter just one of the locks arbitrarily This convention doesn't work because it no longer provides mutual exclusion. It is now possible for two different update sites to choose different locks and thus update the same memory location simultaneously.

Always enter both locks This provides mutual exclusion, but it is twice as expensive, and it offers no advantage over having just one lock for the location.

Consistently enter one of the locks This is the same as saying that only one lock protects the particular location.

How Many Locks?

To illustrate the next point, the example is slightly more complicated. Instead of having just one totalRequests tally, suppose you had two different tallies for high- and low-priority requests. totalRequests is not stored directly, but is calculated as follows:

```
totalRequests = highPriRequests + lowPriRequests;
```

The program needs the invariant that the sum of the two global variables be equal to the number of times any thread has serviced a request. Unlike the previous example, this invariant involves two memory locations. This immediately brings up the question of whether you need one lock or two. The answer to this depends on your design goals.

The main advantage of having two locks, one for highPriRequests and another for lowPriRequests, is that it allows more concurrency. If one thread is trying to update highPriRequests and another thread is trying to update lowPriRequests, but there is only one lock, then one thread would have to wait for the other. If there were two locks, then each thread could proceed without contention. In the example, this improvement in concurrency is trivial since a single lock would be taken relatively rarely and would not be held for a long time. Imagine, however, if the lock was protecting a table that was used frequently during the processing of requests. In this case, it may pay to lock only parts of the table (such as the hash bucket entries) so that several threads can access the table simultaneously.

The main disadvantage of having two locks is the complexity involved. The program clearly has more parts associated with it so there is more opportunity for programmer error. This complexity grows quickly with the number of locks in the system, so it is best to have few locks that protect large regions of memory and only split them when lock contention is shown to be a bottleneck on performance.

At the extreme, a program could have a single lock that protects all memory that is reachable from more than one thread. This design would work quite well for the request-processing example if a thread can process a request without needing to access shared data. If processing a request requires a thread to make many updates to shared memory, then a single lock will become a bottleneck. In this case, the single large region of memory protected by the single lock needs to be split into several non-overlapping subsets, each protected by its own lock.

Taking Locks on Reads

Thus far, I've presented the convention that a lock should always be entered before writing a memory location, but I have not discussed what should happen when memory is read. The read case is a little more complicated because it depends on the programmer's expectations. Consider the example again. Let's imagine that you have decided to read highPriRequests and lowPriRequests to compute totalRequests:

```
totalRequests = highPriRequests + lowPriRequests;
```

In this case, the expectation is that the values in these two memory locations add up to an accurate total number of requests. This will only be true if they are not changing while this computation is proceeding. If each tally has its own lock, both locks need to be entered before the sum can be computed.

In contrast, the code that increments highPriRequests only needs to take a single lock. This is because the only invariant used by the update code is that highPriRequests be an accurate tally; lowPriRequests is not involved whatsoever. In general, when code needs a program invariant, all locks associated with any memory involved with the invariant must be taken.

A useful analogy that helps illustrate this point is shown in Figure 5. Think of the computer's memory as a slot machine with thousands of windows, one for each memory location. When you start your program it is like pulling the handle of the slot machine. Memory locations start spinning as other threads change the values of memory. When a thread enters a lock, the locations associated with the lock stop spinning because the code consistently follows the convention that a lock is taken before any update is attempted. The thread can repeat this process, taking more locks and causing more memory to freeze until all the memory locations needed by the thread are stabilized. The thread is now in a position to perform the operation without interference from other threads.

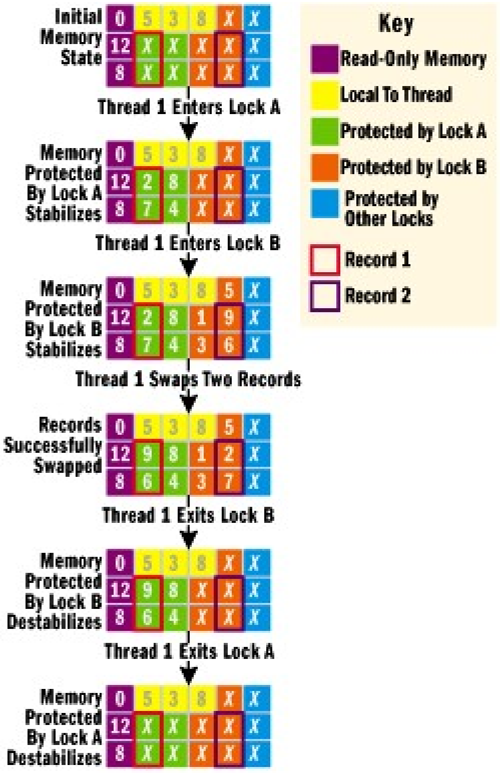

Figure 5 Five Steps to Swap Values Protected by Locks

This analogy is useful in changing the programmer's mindset from believing that nothing changes unless it is explicitly changed to believing that everything changes unless locks are used to prevent it. Adopting this new mindset is the most important piece of advice when building multithreaded applications.

What Memory Needs Lock Protection?

You've seen how to use locks to protect program invariants, but I haven't been precise about what memory needs such protection. An easy (and correct) answer would be that all memory needs to be protected by a lock, but this would be overkill for most apps.

Memory can be made safe for multithreaded use in one of several ways. First, memory that is only accessed by a single thread is safe because other threads are unaffected by it. This includes most local variables and all heap-allocated memory before it is published (made reachable to other threads). Once memory is published, however, it falls out of this category and some other technique must be used.

Second, memory that is read-only after publication does not need a lock because any invariants associated with it must hold for the rest of the program (since the value does not change).

Third, memory that is actively updated from multiple threads generally uses locks to ensure that only one thread has access while a program invariant is broken.

Finally, in certain specialized cases where the program invariant is relatively weak, it is possible to perform updates that can be done without locks. Typically, specialized compare-and-exchange instructions are used. These techniques are best thought of as special, lightweight implementations of locks.

All memory used in a program should fall into one of these four cases. Moreover, the final case is significantly more subtle and error prone, and thus should only be used when performance requirements justify the extra care and risk involved (I will be devoting a future article to this case). Ignoring that case for now, your general rule should be that all program memory should fall into one of three buckets: thread exclusive, read-only, or lock protected.

Methodical Locking

In practice, most nontrivial multithreaded programs are riddled with races. Much of the problem is that programmers are simply unclear about when locks are needed, which I've hopefully now cleared up. This understanding alone is not sufficient, however. It is frightfully easy to make mistakes because it only takes one missed lock to introduce races. You need strong methodologies that help avoid simple-but-common mistakes. However, even the best current techniques require significant care to apply them well.

One of the simplest and most useful techniques for methodical locking is the concept of a monitor. The basic idea is to piggyback on the data abstraction that is already present in an object-oriented design. Consider the example of a hash table. In a well-designed class, it will already be the case that clients only access its internal state by calling its instance methods. If a lock is taken on entry to any instance method, and released on exit, there is a systematic way of ensuring that all accesses to internal data (instance fields) only occur when the lock is held, as shown in Figure 6. Classes that follow this protocol are called monitors.

It is no accident that the name of a lock in the .NET Framework is System.Threading.Monitor. This type was specifically tailored to support the monitor concept. The reason .NET locks operate on System.Object is to make the creation of monitors easier. Effectively, every object has a built-in lock that can be used to protect its instance data. By embedding the body of every instance method in a lock(this) statement, a monitor can be formed. There is even a special attribute, [MethodImpl(MethodImplAttributes.Synchronized)], that can be placed on instance methods to automatically insert the lock(this) statement. Further, .NET locks are reentrant, meaning that the thread that has entered the lock can enter the lock again without blocking. This allows methods to call other methods on the same class without causing the deadlock that would normally occur.

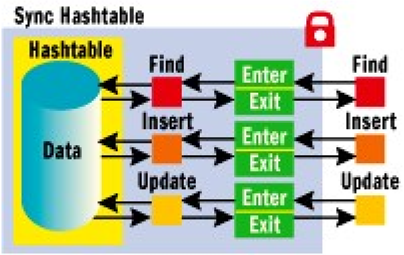

Figure 6 Using a Monitor class

While useful (and easy to write using .NET), monitors are far from a panacea for locking. If they are used indiscriminately, they can result in either too little or too much locking. Consider an application that uses the hash table shown in Figure 6 to implement a higher-level operation called Debit, which transfers money from one account to another. The Debit method uses the Find method on Hashtable to retrieve the two accounts, and the Update method to actually perform the transfer. Because Hashtable is a monitor, the calls to Find and Update are guaranteed to be atomic. Unfortunately, the Debit method needs more than this atomicity guarantee. If another thread updates one of the accounts between the two calls to Update that Debit makes, Debit can be incorrect. The monitor did a fine job of protecting Hashtable within a single call, but an invariant needed over several calls was missed because too little locking was done.

Fixing the race in the Debit method with a monitor can lead to too much locking. What is needed is a lock that protects all the memory that the Debit method uses and can be held for the duration of the method. If you used a monitor to do this, it would be like the Accounts class shown in Figure 7. Each high-level operation like Debit or Credit will take the lock on Accounts before performing its operation, thus providing the needed mutual exclusion. Creating the Accounts monitor fixes the race, but now calls into question the value of the lock on Hashtable. If all accesses to Accounts (and therefore Hashtable) take the Accounts lock, then mutual exclusion for accesses to Hashtable (which is part of Accounts) is already assured. If that is so, then there is no need for the overhead of having a lock for Hashtable. Too much locking was done.

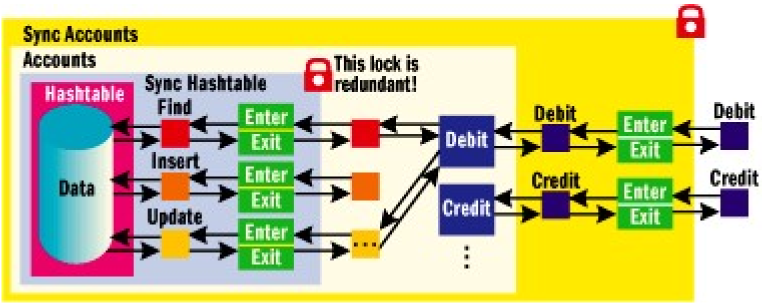

Figure 7 Monitors Needed Only at Top Level

Another important weakness with the monitor concept is that it provides no protection if the class gives out updatable pointers to its data. For example, it is fairly common for methods like the Find on Hashtable to return an object that the caller can then update. Since these updates can happen outside any call to Hashtable , they are not protected by a lock, breaking the protection the monitor was supposed to provide. Finally, monitors simply don't address the more complicated situation when multiple locks need to be taken.

Monitors are a useful concept, but they are just a tool for implementing a well thought out locking design. Sometimes the memory a lock protects naturally coincides with a data abstraction, and a monitor is the perfect mechanism to implement the locking design. However, other times, one lock will protect many data structures, or only parts of a data structure, in which case a monitor is inappropriate. There is no getting around the hard work of precisely defining what locks a system needs and what memory each lock protects. Now let's consider several guidelines that can help you in this design.

Generally speaking, most reusable code (like container classes) should not have locks built into it because it can only protect itself, and it is likely that whatever code uses it will need a stronger lock anyway. The exception to this rule is when it is important that the code works even with high-level program errors. The global memory heap and security-sensitive code are examples of exceptional cases.

It is less error prone and more efficient to have just a few locks that protect large amounts of memory. A single lock that protects many data structures is a good design if it allows the desired amount of concurrency. If the work each of the threads will do doesn't require many updates to shared memory, I would consider taking this principle to the extreme and having one lock that protects all shared memory.

This makes the program almost as simple as a sequential program. It works well for applications whose worker threads don't interact much.

In cases where threads read shared structures frequently, but write to them infrequently, reader-writer locks such as System.Threading.ReaderWriterLock can be used to keep the number of locks in the system low. This type of lock has one method for entering for reading and one for entering for writing. The lock will allow multiple readers to enter concurrently, but writers get exclusive access. Since readers now don't block one another, the system can be made simpler (containing fewer locks) and still achieve the necessary concurrency.

Unfortunately, even with these guidelines to help, designing highly concurrent systems is fundamentally harder than writing sequential systems. Lock usage can often be at odds with normal object-oriented program abstraction because locking is really another independent dimension of the program that has its own design criteria (other such dimensions include lifecycle management, transactional behavior, real-time constraints, and exception-handling behavior). Sometimes the needs of locking and the needs of data abstraction align, such as when they both are used to control access to instance data. There are other times, however, when they are in conflict. (Monitors don't usefully nest, and pointers can cause monitors to "leak".)

There is no good resolution to this conflict. The bottom line is that multithreaded programs are more complicated. The trick is to control the complexity. You've already seen one strategy: try to use just a few locks at the top level of your data abstraction hierarchy. Even this strategy can be at odds with modularity because many data structures will likely be protected by one lock. This means that there is no obvious data structure on which to hang the lock. Typically, the lock needs to be a global variable (never a really great idea for read/write data), or part of the most global data structure involved. In the latter case, it must be possible to get at that structure from any other structure that needs the lock. Sometimes this is a hardship because extra parameters might need to be added to some methods, and the design might get a bit messy, but it's better than the alternatives.

When complexity like this starts to show itself in a design, the correct response is to make it explicit, not to ignore it. If some method does not take locks itself, but expects its caller to provide that mutual exclusion, then that requirement is a precondition of calling that method and should be in its interface contract. If, on the other hand, a data abstraction might take a lock or call client (virtual) methods while holding a lock, that also needs to be part of the interface contract. Only by making these details explicit at interface boundaries can you make good decisions locally about the code. In a good design, most of these contracts are trivial. The callee expects the caller to have provided exclusion for the entire data structure involved, so specifying this is not a hardship.

In an ideal world, the complexity of concurrency design would be hidden in a class library. Unfortunately, there is also little that class library designers can do to make the library friendlier to multithreading. As the Hashtable example shows, locking just one data structure is rarely useful. It is only when the threading structure of program is designed that locks can be usefully added. Typically this makes it the application developer's responsibility. Only holistic frameworks like ASP.NET that define the threading structure end-user code plugs into are in a position to relieve their clients from the burden of careful lock design and analysis.

Deadlock

Another reason to avoid having many locks in the system is deadlock. Once a program has more than one lock, deadlock becomes a possibility. For example, if one thread tries to enter Lock A and then Lock

B, while simultaneously another thread tries to enter Lock B and then Lock A, it is possible for them to deadlock if each enters the lock that the other owns before attempting to enter the second lock.

From a pragmatic perspective, deadlocks are generally prevented in one of two ways. The first (and best) way to prevent deadlock, is to have few enough locks in the system that it is never necessary to take more than one lock at a time. If this is impossible, deadlock can also be prevented by having a convention on the order in which locks are taken. Deadlocks can only occur if there is a circular chain of threads such that each thread in the chain is waiting on a lock already acquired by the next in line. To prevent this, each lock in the system is assigned a "level", and the program is designed so that threads always take locks only in strictly descending order by level. This protocol makes cycles involving locks, and therefore deadlock, impossible. If this strategy does not work (can't find a set of levels), it is likely that the lock-taking behavior of the program is so input-dependent that it is impossible to guarantee that deadlock can not happen in every case. Typically, this type of code falls back on timeouts or some kind of deadlock detection scheme.

Deadlock is just one more reason to keep the number of locks in the system small. If this cannot be done, an analysis must be made to determine why more than one lock has to be taken simultaneously. Remember, taking multiple locks is only necessary when code needs to get exclusive access to memory protected by different locks. This analysis typically either yields a trivial lock ordering that will avoid deadlock or shows that complete deadlock avoidance is impossible.

The Cost of Locks

Another reason to avoid having many locks in a system is the cost of entering and leaving a lock. The lightest locks use a special compare/exchange instruction to check whether the lock is taken, and if it isn't, they enter the lock in a single atomic action. Unfortunately, this special instruction is relatively expensive (typically ten to hundreds of times longer than an ordinary instruction). There are two main reasons for this expense, and they both have to do with issues arising on a true multiprocessor system.

The first reason is that the compare/exchange instruction must ensure that no other processor is also trying to do the same thing. Fundamentally, this requires one processor to coordinate with all other processors in the system. This is a slow operation and accounts for the lower bound of the cost of a lock (dozens of cycles). The other reason for the expense is the effect inter-process communication has on the memory system. After a lock has been taken, the program is very likely to access memory that may have been recently modified by another thread. If this thread ran on another processor, it is necessary to ensure that all pending writes on all other processors have been flushed so that the current thread sees the updates. The cost of doing this largely depends on how the memory system works and how many writes need to be flushed. This cost can be pretty high in the worst case (possibly hundreds of cycles or more).

Thus the cost of a lock is can be significant. If a frequently called method needs to take a lock and only executes a hundred or so instructions, lock overhead is likely to be an issue. The program generally needs to be redesigned so that the lock can be held for a larger unit of work.

In addition to the raw overhead of entering and leaving locks, as the number of processors in the system grows, locks become the main impediment in using all the processors efficiently. If there are too few locks in the program, it is impossible to keep all the processors busy since they are waiting on memory that is locked by another processor. On the other hand, if there are many locks in the program, it is easy to have a "hot" lock that is being entered and exited frequently by many processors. This causes the memory-flushing overhead to be very high, and throughput again does not scale linearly with the number of processors. The only design that scales well is one in which worker threads can do significant work without interacting with shared data.

Inevitably, performance issues might make you want to avoid locks altogether. This can be done in certain constrained circumstances, but doing it correctly involves even more subtleties than getting mutual exclusion correct using locks. It should only be used as a last resort, and only after understanding the issues involved. I will be devoting a complete article to this in a future issue.

A Quick Word on Synchronization

While locks provide a way of keeping threads out of each other's way, they really don't provide a mechanism for them to cooperate (synchronize). I'll quickly cover the principles for synchronizing threads without introducing races. As you'll see, they are not much different from the principles for proper locking.

In the .NET Framework, synchronization functionality is implemented in the ManualResetEvent and AutoResetEvent classes. These classes provide Set, Reset, and WaitOne methods. The WaitOne method causes a thread to block as long as the event is in the reset state. When another thread calls the Set method, AutoResetEvents will allow one thread that called WaitOne to unblock, while ManualResetEvents will unblock all waiting threads.

Generally, events are used to signal that a more complicated program property holds. For example, a program might have a queue of work for a thread, and an event is used to signal to the thread that the queue is not empty. Note that this introduces a program invariant that the event should be set if and only if the queue is not empty. The rules for proper locking require that if code needs an invariant, there must be locks that provide exclusive access for all memory associated with that invariant. Applying this principle in a queue suggests that all access to the event and the queue should happen only after entering a common lock.

Unfortunately, this design can cause a deadlock. Take this scenario for example. Thread A enters the lock and needs to wait for the queue to be filled (while owning the queue's lock). Thread B, which is attempting to add an entry to the queue that Thread A needs, will try to enter the queue's lock before modifying the queue and thus block on Thread A. Deadlock!

In general, it's a bad idea to hold locks while waiting on events. After all, why lock out all other threads from a data structure when a thread is waiting on something else? You're just asking for deadlocks. A common practice is to release the lock and then wait on the event. However it's now possible for the event and the queue to be out of sync. We have broken the invariant that the event is an accurate indicator of when the queue is not empty. A typical solution is to weaken the invariant in this case to, "if the event is reset, then the queue is empty." This invariant is strong enough that it is still safe to wait on the event without risking waiting forever. This relaxed invariant means that when a thread returns from WaitOne, it cannot assume that the queue has an element in it. The waking thread has to enter the queue's lock and verify that the queue has an element. If not (say some other thread removed the entry), it must wait again. If fairness among threads is important, this solution has a problem, but it does work well for most purposes.

Conclusion

The fundamental principles underlying good coding in sequential and multithreaded programs are not so different. In both cases, the entire code base has to methodically protect the invariants that are needed elsewhere in the program. The difference is that, as Figure 2 shows, protecting program invariants is more complicated in the multithreaded case. As a result it requires a significantly higher degree of discipline to build a correct multithreaded program. Part of this discipline is ensuring through tools like monitors that all thread-shared, read-write data is protected by locks. The other part is carefully designing which memory is protected by which lock, and controlling the extra complexity that locks inevitably add to the program. As in a sequential program, good designs are typically the simplest, and for multithreaded programs this means having the fewest number of locks that achieve the needed concurrency. If you keep the locking design simple and methodically follow your locking design, you *can* write race-free multithreaded programs.

Vance Morrison is the compiler architect for the .NET runtime at Microsoft, where he has been involved in the design of the .NET runtime since its inception. He drove the design for the .NET Intermediate Language (IL) and was lead for the just-in-time (JIT) compiler team.