Chapter 20

Thread Synchronization

In Chapter 19, we discussed the details of multithreaded programming using the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). One topic we specifically avoided, however, was thread synchronization, which prevents race conditions while avoiding deadlocks. Thread synchronization is the topic of this chapter.

\*\*\*COMP: Insert 20mindmap.pdf

Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_Mindmaps:20mindmap.pdf

We begin with a multithreaded example with no thread synchronization around shared data—resulting in a race condition in which data integrity is lost. This discussion serves as the introduction for why we need thread synchronization. It is followed by coverage of myriad mechanisms and best practices for doing it.

Prior editions of this book included a significant section on additional multithreading patterns and another on various timer callback mechanisms. With the introduction of the async/await pattern, however, those approaches have essentially been replaced unless you are programming with frameworks prior to C# 5.0/.NET 4.5. However, pre-C# 5.0 material is still available from this book’s website: http://IntelliTect.com/EssentialCSharp.

This entire chapter uses the TPL, so the samples cannot be compiled on frameworks prior to Microsoft .NET Framework 4. However, unless specifically identified as a Microsoft .NET Framework 4 API, the only reason for the Microsoft .NET Framework 4 restriction is the use of the System.Threading.Tasks.Task class to execute the asynchronous operation. Modifying the code to instantiate a System.Threading.Thread and use a Thread.Join() to wait for the thread to execute will allow the vast majority of samples to compile on earlier frameworks.

That being said, the specific API for starting tasks throughout this chapter is the .NET 4.5 (or later) System.Threading.Tasks.Task.Run(). As we discussed in Chapter 19, this method is preferred over System.Threading.Tasks.Task.Factory.StartNew() because it is simpler and sufficient for the majority of scenarios. If you are limited to .NET 4, you can replace Task.Run() with Task.Factory.StartNew() without any additional modifications. (For this reason, the chapter does not explicitly highlight such code as .NET 4.5–specific code when only this method is used.)

Why Synchronization?

Running a new thread is a relatively simple programming task. What makes multithreaded programming difficult, however, is identifying which data multiple threads can safely access simultaneously. The program must synchronize such data to prevent simultaneous access, thereby creating the “safety.” Consider Listing 22.1.

Listing 22.1: Unsynchronized State

using System;

using System.Threading.Tasks;

public class Program

{

const int \_Total = int.MaxValue;

static long \_Count = 0;

public static void Main()

{

// Use Task.Factory.StartNew for .NET 4.0

Task task = Task.Run(() => Decrement());

// Increment

for(int i = 0; i < \_Total; i++)

{

\_Count++;

}

task.Wait();

Console.WriteLine("Count = {0}", \_Count);

}

static void Decrement()

{

// Decrement

for(int i = 0; i < \_Total; i++)

{

\_Count--;

}

}

}

One possible result of Listing 22.1 appears in Output 20.1.

Output 20.1

Count = 113449949

The important thing to note about Listing 22.1 is that the output is not 0. It would have been if Decrement() was called directly (sequentially). However, when calling Decrement() asynchronously, a race condition occurs because the individual steps within \_Count++ and \_Count-- statements intermingle. (As discussed in the Beginner Topic titled “Multithreading Jargon” in Chapter 19, a single statement in C# will likely involve multiple steps.) Consider the sample execution in Table 20.1.

Table 20.1: Sample Pseudocode Execution

|  |  |  |
| --- | --- | --- |
| Main Thread | Decrement Thread | Count |
| ... | ... | ... |
| Copy the value 0 out of \_Count. |  | 0 |
| Increment the copied value (0), resulting in 1. |  | 0 |
| Copy the resultant value (1) into \_Count. |  | 1 |
| Copy the value 1 out of \_Count. |  | 1 |
|  | Copy the value 1 out of \_Count. | 1 |
| Increment the copied value (1), resulting in 2. |  | 1 |
| Copy the resultant value (2) into \_Count. |  | 2 |
|  | Decrement the copied value (1), resulting in 0. | 2 |
|  | Copy the resultant value (0) into \_Count. | 0 |
| ... | ... | ... |

Table 20.1 shows a parallel execution (or a thread context switch) by the transition of instructions appearing from one column to the other. The value of \_Count after a particular line has completed appears in the last column. In this sample execution, \_Count++ executes twice and \_Count-- occurs once. However, the resultant \_Count value is 0, not 1. Copying a result back to \_Count essentially wipes out any \_Count value changes that have occurred since the read of \_Count on the same thread.

The problem in Listing 22.1 is a race condition, where multiple threads have simultaneous access to the same data elements. As this sample execution demonstrates, allowing multiple threads to access the same data elements is likely to undermine data integrity, even on a single-processor computer. To remedy this potential problem, the code needs synchronization around the data. Code or data synchronized for simultaneous access by multiple threads is **thread-safe**.

There is one important point to note about atomicity of reading and writing to variables. The runtime guarantees that a type whose size is no bigger than a native (pointer-size) integer will not be read or written partially. With a 64-bit operating system, therefore, reads and writes to a long (64 bits) will be atomic. However, reads and writes to a 128-bit variable such as decimal may not be atomic. Therefore, write operations to change a decimal variable may be interrupted after copying only 32 bits, resulting in the reading of an incorrect value, known as a **torn read**.

Beginner Topic

Multiple Threads and Local Variables

Note that it is not necessary to synchronize local variables. Local variables are loaded onto the stack, and each thread has its own logical stack. Therefore, each local variable has its own instance for each method call. By default, local variables are not shared across method calls; likewise, they are not shared among multiple threads.

However, this does not mean local variables are entirely without concurrency issues—after all, code could easily expose the local variable to multiple threads.[[1]](#footnote-1) A parallel for loop that shares a local variable between iterations, for example, will expose the variable to concurrent access and a race condition (see Listing 22.2).

Listing 22.2: Unsynchronized Local Variables

using System;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

int x = 0;

Parallel.For(0, int.MaxValue, i =>

{

x++;

x--;

});

Console.WriteLine("Count = {0}", x);

}

}

In this example, x (a local variable) is accessed within a parallel for loop, so multiple threads will modify it simultaneously, creating a race condition very similar to that in Listing 22.1. The output is unlikely to yield the value 0 even though x is incremented and decremented the same number of times.

\*\*\*COMP: Insert “Begin 4.0” tab

Synchronization Using Monitor

To synchronize multiple threads so that they cannot execute particular sections of code simultaneously, you can use a **monitor** to block the second thread from entering a protected code section before the first thread has exited that section. The monitor functionality is part of a class called System.Threading.Monitor, and the beginning and end of protected code sections are marked with calls to the static methods Monitor.Enter() and Monitor.Exit(), respectively.

Listing 22.3 demonstrates synchronization using the Monitor class explicitly. As this listing shows, it is important that all code between calls to Monitor.Enter() and Monitor.Exit() be surrounded with a try/finally block. Without this block, an exception could occur within the protected section and Monitor.Exit() may never be called, thereby blocking other threads indefinitely.

Listing 22.3: Synchronizing with a Monitor Explicitly

using System;

using System.Threading;

using System.Threading.Tasks;

public class Program

{

readonly static object \_Sync = new object();

const int \_Total = int.MaxValue;

static long \_Count = 0;

public static void Main()

{

// Use Task.Factory.StartNew for .NET 4.0

Task task = Task.Run(() => Decrement());

// Increment

for(int i = 0; i < \_Total; i++)

{

bool lockTaken = false;

try

{

Monitor.Enter(\_Sync, ref lockTaken);

\_Count++;

}

finally

{

if (lockTaken)

{

Monitor.Exit(\_Sync);

}

}

}

task.Wait();

Console.WriteLine($"Count = {\_Count}");

}

static void Decrement()

{

for(int i = 0; i < \_Total; i++)

{

bool lockTaken = false;

try

{

Monitor.Enter(\_Sync, ref lockTaken);

\_Count--;

}

finally

{

if(lockTaken)

{

Monitor.Exit(\_Sync);

}

}

}

}

}

The results of Listing 22.3 appear in Output 20.2.

Output 20.2

Count = 0

Note that calls to Monitor.Enter() and Monitor.Exit() are associated with each other by sharing the same object reference passed as the parameter (in this case, \_Sync). The Monitor.Enter() overload method that takes the lockTaken parameter was added to the framework only in .NET 4.0. Before that, no such lockTaken parameter was available and there was no way to reliably catch an exception that occurred between the Monitor.Enter() and the try block. Placing the try block immediately following the Monitor.Enter() call was reliable in release code because the just-in-time compiler, or JIT, prevented any such asynchronous exception from sneaking in. However, anything other than a try block immediately following the Monitor.Enter(), including any instructions that the compiler might have injected within debug code, could prevent the JIT from reliably returning execution within the try block. Therefore, if an exception did occur, it would leak the lock (the lock remained acquired) rather than executing the finally block and releasing it—likely causing a deadlock when another thread tried to acquire the lock. In summary, in versions of the framework prior to .NET 4.0, you should always follow Monitor.Enter() with a try/finally {Monitor.Exit(\_Sync))} block.

Monitor also supports a Pulse() method for allowing a thread to enter the ready queue, indicating it is up next for execution. This is a common means of synchronizing producer–consumer patterns so that no “consume” occurs until there has been a “produce.” The producer thread that owns the monitor (by calling Monitor.Enter()) calls Monitor.Pulse() to signal the consumer thread (which may already have called Monitor.Enter()) that an item is available for consumption and that it should get ready. For a single Pulse() call, only one thread (the consumer thread, in this case) can enter the ready queue. When the producer thread calls Monitor.Exit(), the consumer thread takes the lock (Monitor.Enter() completes) and enters the critical section to begin consuming the item. Once the consumer processes the waiting item, it calls Exit(), thus allowing the producer (currently blocked with Monitor.Enter()) to produce again. In this example, only one thread can enter the ready queue at a time, ensuring that there is no consumption without production, and vice versa.

\*\*\*COMP: Insert “End 4.0” tab

Using the lock Keyword

Because of the frequent need for synchronization using Monitor in multithreaded code, and because the try/finally block can easily be forgotten, C# provides a special keyword to handle this locking synchronization pattern. Listing 22.4 demonstrates the use of the lock keyword, and Output 20.3 shows the results.

Listing 22.4: Synchronization Using the lock Keyword

using System;

using System.Threading;

using System.Threading.Tasks;

public class Program

{

readonly static object \_Sync = new object();

const int \_Total = int.MaxValue;

static long \_Count = 0;

public static void Main()

{

// Use Task.Factory.StartNew for .NET 4.0

Task task = Task.Run(() => Decrement());

// Increment

for(int i = 0; i < \_Total; i++)

{

lock(\_Sync)

{

\_Count++;

}

}

task.Wait();

Console.WriteLine($"Count = {\_Count}");

}

static void Decrement()

{

for(int i = 0; i < \_Total; i++)

{

lock(\_Sync)

{

\_Count--;

}

}

}

}

Output 20.3

Count = 0

By locking the section of code accessing \_Count (using either lock or Monitor), you make the Main() and Decrement() methods thread-safe, meaning they can be safely called from multiple threads simultaneously. (Prior to C# 4.0, the concept was the same but the compiler-emitted code depended on the lockTaken-less Monitor.Enter() method, and the Monitor.Enter() called was emitted before the try block.)

The price of synchronization is a reduction in performance. Listing 22.5, for example, takes an order of magnitude longer to execute than Listing 22.1 does, which demonstrates lock’s relatively slow execution compared to the execution of incrementing and decrementing the count.

Even when lock is insignificant in comparison with the work it synchronizes, programmers should avoid indiscriminate synchronization so as to avoid the possibility of deadlocks and unnecessary synchronization on multiprocessor computers that could instead be executing code in parallel. The general best practice for object design is to synchronize mutable static state and not any instance data. (There is no need to synchronize something that never changes.) Programmers who allow multiple threads to access a particular object must provide synchronization for the object. Any class that explicitly deals with threads is likely to want to make instances thread-safe to some extent.

\*\*\*COMP: Insert “Begin 7.1” tab

Beginner Topic

Task Return with No await

In Listing 22.1, although Task.Run(() => Decrement()) returns a Task, the await operator is not used. The reason for this is that until C# 7.1, Main() doesn’t support the use of async. Given C# 7.1, however, the code can be refactored to use the async/await pattern, as shown in Listing 22.5.

Listing 22.5: async Main() with C# 7.1

using System;

using System.Threading.Tasks;

public class Program

{

readonly static object \_Sync = new object();

const int \_Total = int.MaxValue;

static long \_Count = 0;

public static async Task Main()

{

// Use Task.Factory.StartNew for .NET 4.0

Task task = Task.Run(() => Decrement());

// Increment

for(int i = 0; i < \_Total; i++)

{

lock(\_Sync)

{

\_Count++;

}

}

await task;

Console.WriteLine($"Count = {\_Count}");

}

static void Decrement()

{

for(int i = 0; i < \_Total; i++)

{

lock(\_Sync)

{

\_Count--;

}

}

}

}

\*\*\*COMP: Insert “End 7.1” tab

\*\*\*COMP: Beginner Topic ends after listing

Choosing a lock Object

Whether or not the lock keyword or the Monitor class is explicitly used, it is crucial that programmers carefully select the lock object.

In the previous examples, the synchronization variable, \_Sync, is declared as both private and read-only. It is declared as read-only to ensure that the value is not changed between calls to Monitor.Enter() and Monitor.Exit(). This allows correlation between entering and exiting the synchronized block.

Similarly, the code declares \_Sync as private so that no synchronization block outside the class can synchronize the same object instance, causing the code to block.

If the data is public, the synchronization object may be public so that other classes can synchronize using the same object instance. However, this makes it harder to avoid deadlock. Fortunately, the need for this pattern is rare. For public data, it is instead preferable to leave synchronization entirely outside the class, allowing the calling code to take locks with its own synchronization object.

It’s important that the synchronization object not be a value type. If the lock keyword is used on a value type, the compiler will report an error. (In the case of accessing the System.Threading.Monitor class explicitly [not via lock], no such error will occur at compile time. Instead, the code will throw an exception with the call to Monitor.Exit(), indicating there was no corresponding Monitor.Enter() call.) The issue is that when using a value type, the runtime makes a copy of the value, places it in the heap (boxing occurs), and passes the boxed value to Monitor.Enter(). Similarly, Monitor.Exit() receives a boxed copy of the original variable. The result is that Monitor.Enter() and Monitor.Exit() receive different synchronization object instances so that no correlation between the two calls occurs.

Why to Avoid Locking on this, typeof(type), and string

One seemingly reasonable pattern is to lock on the this keyword for instance data in a class and on the type instance obtained from typeof(type) (e.g., typeof(MyType)) for static data. Such a pattern provides a synchronization target for all states associated with a particular object instance when this is used and for all static data for a type when typeof(type) is used. The problem is that the synchronization target that this (or typeof(type)) points to could participate in the synchronization target for an entirely different synchronization block created in an unrelated block of code. In other words, although only the code within the instance itself can block using the this keyword, the caller that created the instance can pass that instance to a synchronization lock.

The result is that two different synchronization blocks that synchronize two entirely different sets of data could block each other. Although perhaps unlikely, sharing the same synchronization target could have an unintended performance impact and, in extreme cases, could even cause a deadlock. Instead of locking on this or even typeof(type), it is better to define a private, read-only field on which no one will block except for the class that has access to it.

Another lock type to avoid is string because of string interning. If the same string constant appears within multiple locations, it is likely that all locations will refer to the same instance, making the scope of the lock much broader than expected.

In summary, you should use a per-synchronization context instance of type object for the lock target.

Guidelines

**AVOID** locking on this, System.Type, or a string.

**DO** declare a separate, read-only synchronization variable of type object for the synchronization target.

Advanced Topic

Avoid Synchronizing with MethodImplAttribute

One synchronization mechanism that was introduced in .NET 1.0 was the MethodImplAttribute. Used in conjunction with the MethodImplOptions.Synchronized method, this attribute marks a method as synchronized so that only one thread can execute the method at a time. To achieve this, the JIT essentially treats the method as though it was surrounded by lock(this) or, in the case of a static method, locks on the type. Such an implementation means that, in fact, the method and all other methods on the same class, decorated with the same attribute and enum parameter, are synchronized—rather than each method being synchronized relative to itself. In other words, given two or more methods on the same class decorated with the attribute, only one of them will be able to execute at a time, and the one executing will block all calls by other threads to itself or to any other method in the class with the same decoration. Furthermore, since the synchronization is on this (or even worse, on the type), it suffers the same detriments as lock(this) (or worse, for the static) discussed in the preceding section. As a result, it is a best practice to avoid the attribute altogether.

Guidelines

**AVOID** using the MethodImplAttribute for synchronization.

\*\*\*COMP: Advanced Topic ends after Guidelines

Declaring Fields as volatile

On occasion, the compiler or CPU may optimize code in such a way that the instructions do not occur in the exact order they are coded, or some instructions are optimized out. Such optimizations are innocuous when code executes on one thread. However, with multiple threads, such optimizations may have unintended consequences because the optimizations may change the order of execution of a field’s read or write operations relative to an alternate thread’s access to the same field.

One way to stabilize this behavior is to declare fields using the volatile keyword. This keyword forces all reads and writes to the volatile field to occur at the exact location the code identifies instead of at some other location that the optimization produces. The volatile modifier identifies that the field is susceptible to modification by the hardware, operating system, or another thread. As such, the data is “volatile,” and the keyword instructs the compilers and runtime to handle it more exactly. (See http://bit.ly/CSharpReorderingOptimizations for further details.)

In general, the use of the volatile modifier is rare and fraught with complications that will likely lead to incorrect usage. Using lock is preferred to the volatile modifier unless you are absolutely certain about the volatile usage.

Using the System.Threading.Interlocked Class

The mutual exclusion pattern described so far provides the minimum set of tools for handling synchronization within a process (application domain). However, synchronization with System.Threading.Monitor is a relatively expensive operation, and an alternative solution that the processor supports directly targets specific synchronization patterns.

Listing 22.6 sets \_Data to a new value as long as the preceding value was null. As indicated by the method name, this pattern is the compare/exchange pattern. Instead of manually placing a lock around behaviorally equivalent compare and exchange code, the Interlocked.CompareExchange() method provides a built-in method for a synchronous operation that does the same check for a value (null) and updates the first parameter if the value is equal to the second parameter. Table 20.2 shows other synchronization methods supported by Interlocked.

Listing 22.6: Synchronization Using System.Threading.Interlocked

public class SynchronizationUsingInterlocked

{

private static object? \_Data;

// Initialize data if not yet assigned

static void Initialize(object newValue)

{

// If \_Data is null, then set it to newValue

Interlocked.CompareExchange(

ref \_Data, newValue, null);

}

// ...

}

\*\*\*COMP: Insert “Begin 2.0” tab

Table 20.2: Interlocked’s Synchronization-Related Methods

|  |  |
| --- | --- |
| Method Signature | Description |
| public static T CompareExchange<T>(  T location,  T value,  T comparand  ); | Checks location for the value in comparand. If the values are equal, it sets location to value and returns the original data stored in location. |
| public static T Exchange<T>(  T location,  T value  ); | Assigns location with value and returns the previous value. |
| public static int Decrement(  ref int location  ); | Decrements location by 1. It is equivalent to the prefix -- operator, except Decrement() is thread-safe. |
| public static int Increment(  ref int location  ); | Increments location by 1. It is equivalent to the prefix ++ operator, except Increment() is thread-safe. |
| public static int Add(  ref int location,  int value  ); | Adds value to location and assigns location the result. It is equivalent to the += operator |
| public static long Read(  ref long location  ); | Returns a 64-bit value in a single atomic operation. |

\*\*\*COMP: Insert “End 2.0” tab

Most of these methods are overloaded with additional data type signatures, such as support for long. Table 20.2 provides the general signatures and descriptions.

Note that you can use Increment() and Decrement() in place of the synchronized ++ and -- operators from Listing 22.5, and doing so will yield better performance. Also note that if a different thread accessed \_Count using a non-interlocked method, the two accesses would not be synchronized correctly.

Event Notification with Multiple Threads

One area where developers often overlook synchronization is when firing events. The unsafe thread code for publishing an event is similar to Listing 22.7.

Listing 22.7: Firing an Event Notification

// Not thread-safe

if(OnTemperatureChanged != null)

{

// Call subscribers

OnTemperatureChanged(

this, new TemperatureEventArgs(value) );

}

This code is valid as long as there is no race condition between this method and the event subscribers. However, the code is not atomic, so multiple threads could introduce a race condition. It is possible that between the time when OnTemperatureChange is checked for null and when the event is actually fired, OnTemperatureChange could be set to null, thereby throwing a NullReferenceException. In other words, if multiple threads could potentially access a delegate simultaneously, it is necessary to synchronize the assignment and firing of the delegate.

The C# 6.0 solution to this dilemma is trivial. All that is necessary is to use the null-conditional operator:

OnTemperature?.Invoke(

this, new TemperatureEventArgs( value ) );

The null-conditional operator is specifically designed to be atomic, so this invocation of the delegate is, in fact, atomic. The key, obviously, is to remember to make use of the null-conditional operator.

Although it requires more code, thread-safe delegate invocation prior to C# 6.0 isn’t especially difficult, either. This approach works because the operators for adding and removing listeners are thread-safe and static (operator overloading is done with static methods). To correct Listing 22.7 and make it thread-safe, assign a copy, check the copy for null, and fire the copy (see Listing 22.8).

Listing 22.8: Thread-Safe Event Notification

// ...

TemperatureChangedHandler localOnChange =

OnTemperatureChanged;

if(localOnChanged != null)

{

// Call subscribers

localOnChanged(

this, new TemperatureEventArgs(value) );

}

// ...

Given that a delegate is a reference type, it is perhaps surprising that assigning a local variable and then firing with the local variable is sufficient for making the null check thread-safe. As localOnChange points to the same location that OnTemperatureChange points to, you might think that any changes in OnTemperatureChange would be reflected in localOnChange as well.

However, this is not the case: Any calls to OnTemperatureChange += <listener> will not add a new delegate to OnTemperatureChange but rather will assign it an entirely new multicast delegate without having any effect on the original multicast delegate to which localOnChange also points. This makes the code thread-safe because only one thread will access the localOnChange instance, and OnTemperatureChange will be an entirely new instance if listeners are added or removed.

Synchronization Design Best Practices

Along with the complexities of multithreaded programming come several best practices for handling those complexities.

Avoiding Deadlock

With the introduction of synchronization comes the potential for deadlock. Deadlock occurs when two or more threads wait for one another to release a synchronization lock. For example, suppose Thread 1 requests a lock on \_Sync1, and then later requests a lock on \_Sync2 before releasing the lock on \_Sync1. At the same time, Thread 2 requests a lock on \_Sync2, followed by a lock on \_Sync1, before releasing the lock on \_Sync2. This sets the stage for the deadlock. The deadlock actually occurs if both Thread 1 and Thread 2 successfully acquire their initial locks (\_Sync1 and \_Sync2, respectively) before obtaining their second locks.

For a deadlock to occur, four fundamental conditions must be met:

1. Mutual exclusion: One thread (Thread A) exclusively owns a resource such that no other thread (Thread B) can acquire the same resource.

2. Hold and wait: One thread (Thread A) with a mutual exclusion is waiting to acquire a resource held by another thread (Thread B).

3. No preemption: The resource held by a thread (Thread A) cannot be forcibly removed (Thread A needs to release its own locked resource).

4. Circular wait condition: Two or more threads form a circular chain such that they lock on the same two or more resources, and each waits on the resource held by the next thread in the chain.

Removing any one of these conditions will prevent the deadlock.

A scenario likely to cause a deadlock is when two or more threads request exclusive ownership on the same two or more synchronization targets (resources) and the locks are requested in different orders. This situation can be avoided when developers are careful to ensure that multiple lock acquisitions always occur in the same order. Another cause of a deadlock is locks that are not **reentrant**. When a lock from one thread can block the same thread—that is, when it re-requests the same lock—the lock is not reentrant. For example, if Thread A acquires a lock and then re-requests the same lock but is blocked because the lock is already owned (by itself), the lock is not reentrant and the additional request will deadlock.

The code generated by the lock keyword (with the underlying Monitor class) is reentrant. However, as we shall see in the “More Synchronization Types” section, some lock types are not reentrant.

When to Provide Synchronization

As we discussed earlier, all static data should be thread-safe. Therefore, synchronization needs to surround static data that is mutable. Generally, programmers should declare private static variables and then provide public methods for modifying the data. Such methods should internally handle the synchronization if multithreaded access is possible.

In contrast, instance state is not expected to include synchronization. Synchronization may significantly decrease performance and increase the chance of a lock contention or deadlock. With the exception of classes that are explicitly designed for multithreaded access, programmers sharing objects across multiple threads are expected to handle their own synchronization of the data being shared.

Avoiding Unnecessary Locking

Without compromising data integrity, programmers should avoid unnecessary synchronization where possible. For example, you should use immutable types between threads so that no synchronization is necessary (this approach has proved invaluable in functional programming languages such as F#). Similarly, you should avoid locking on thread-safe operations such as simple reads and writes of values smaller than a native (pointer-size) integer, as such operations are automatically atomic.

Guidelines

**DO NOT** request exclusive ownership on the same two or more synchronization targets in different orders.

**DO** ensure that code that concurrently holds multiple locks always acquires them in the same order.

**DO** encapsulate mutable static data in public APIs with synchronization logic.

**AVOID** synchronization on simple reading or writing of values no bigger than a native (pointer-size) integer, as such operations are automatically atomic.

More Synchronization Types

In addition to System.Threading.Monitor and System.Threading.Interlocked, several more synchronization techniques are available.

Using System.Threading.Mutex

\*\*\*COMP: Insert “Begin 2.0” tab

System.Threading.Mutex is similar in concept to the System.Threading.Monitor class (without the Pulse() method support), except that the lock keyword does not use it, and Mutexes can be named so that they support synchronization across multiple processes. Using the Mutex class, you can synchronize access to a file or some other cross-process resource. Since Mutex is a cross-process resource, .NET 2.0 added support to allow for setting the access control via a System.Security.AccessControl.MutexSecurity object. One use for the Mutex class is to limit an application so that it cannot run multiple times simultaneously, as Listing 22.9 demonstrates.

Listing 22.9: Creating a Single Instance Application

using System;

using System.Threading;

using System.Reflection;

public class Program

{

public static void Main()

{

// Obtain the mutex name from the full

// assembly name

string mutexName =

Assembly.GetEntryAssembly()!.FullName!;

// firstApplicationInstance indicates whether this is the first

// application instance

using Mutex mutex = new Mutex(false, mutexName,

out bool firstApplicationInstance);

if(!firstApplicationInstance)

{

Console.WriteLine(

"This application is already running.");

}

else

{

Console.WriteLine("ENTER to shut down");

Console.ReadLine();

}

}

}

The results from running the first instance of the application appear in Output 20.4.

Output 20.4

ENTER to shut down

The results of the second instance of the application while the first instance is still running appear in Output 20.5.

Output 20.5

This application is already running.

In this case, the application can run only once on the machine, even if it is launched by different users. To restrict the instances to once per user, add System.Environment.UserName (which requires the Microsoft .NET Framework or .NET Standard 2.0) as a suffix when assigning the mutexName.

Mutex derives from System.Threading.WaitHandle, so it includes the WaitAll(), WaitAny(), and SignalAndWait() methods. These methods allow it to acquire multiple locks automatically—something Monitor does not support.

\*\*\*COMP: Insert “End 2.0” tab

WaitHandle

The base class for Mutex is a System.Threading.WaitHandle. This is a fundamental synchronization class used by the Mutex, EventWaitHandle, and Semaphore synchronization classes. The key methods on a WaitHandle are the WaitOne() methods. These methods block execution until the WaitHandle instance is signaled or set. The WaitOne() methods include several overloads allowing for an indefinite wait: void WaitOne(), a millisecond-timed wait; bool WaitOne(int milliseconds); and bool WaitOne(TimeSpan timeout), a TimeSpan wait. The versions that return a Boolean will return a value of true whenever the WaitHandle is signaled before the timeout.

In addition to the WaitHandle instance methods, there are two key static members: WaitAll() and WaitAny(). Like their instance cousins, these static members support timeouts. In addition, they take a collection of WaitHandles, in the form of an array, so that they can respond to signals coming from within the collection.

Note that WaitHandle contains a handle (of type SafeWaitHandle) that implements IDisposable. As such, care is needed to ensure that WaitHandles are disposed when they are no longer needed.

\*\*\*COMP: Insert “Begin 4.0” tab

Reset Events: ManualResetEvent and ManualResetEventSlim

One way to control uncertainty about when particular instructions in a thread will execute relative to instructions in another thread is with reset events. In spite of the term events, reset events have nothing to do with C# delegates and events. Instead, reset events are a way to force code to wait for the execution of another thread until the other thread signals. They are especially useful for testing multithreaded code because it is possible to wait for a particular state before verifying the results.

The reset event types are System.Threading.ManualResetEvent and the Microsoft .NET Framework 4–added lightweight version, System.Threading.ManualResetEventSlim. (As discussed in the upcoming Advanced Topic titled “Favor ManualResetEvent and Semaphores over AutoResetEvent,” there is a third type, System.Threading.AutoResetEvent, but programmers should avoid it in favor of one of the first two.) The key methods on the reset events are Set() and Wait() (called WaitOne() on ManualResetEvent). Calling the Wait() method will cause a thread to block until a different thread calls Set() or until the wait period times out. Listing 22.10 demonstrates how this works, and Output 20.6 shows the results.

Listing 22.10: Waiting for ManualResetEventSlim

using System;

using System.Threading;

using System.Threading.Tasks;

public class Program

{

static ManualResetEventSlim \_MainSignaledResetEvent;

static ManualResetEventSlim \_DoWorkSignaledResetEvent;

public static void DoWork()

{

Console.WriteLine("DoWork() started....");

\_DoWorkSignaledResetEvent.Set();

\_MainSignaledResetEvent.Wait();

Console.WriteLine("DoWork() ending....");

}

public static void Main()

{

using(\_MainSignaledResetEvent =

new ManualResetEventSlim())

using (\_DoWorkSignaledResetEvent =

new ManualResetEventSlim())

{

Console.WriteLine(

"Application started....");

Console.WriteLine("Starting task....");

// Use Task.Factory.StartNew for .NET 4.0

Task task = Task.Run(() => DoWork());

// Block until DoWork() has started

\_DoWorkSignaledResetEvent.Wait();

Console.WriteLine(

" Waiting while thread executes...");

\_MainSignaledResetEvent.Set();

task.Wait();

Console.WriteLine("Thread completed");

Console.WriteLine(

"Application shutting down....");

}

}

}

Output 20.6

Application started....

Starting thread....

DoWork() started....

Waiting while thread executes...

DoWork() ending....

Thread completed

Application shutting down....

Listing 22.10 begins by instantiating and starting a new Task. Table 20.3 shows the execution path in which each column represents a thread. In cases where code appears on the same row, it is indeterminate which side executes first.

Table 20.3: Execution Path with ManualResetEvent Synchronization

|  |  |
| --- | --- |
| Main() | DoWork() |
| ... |  |
| Console.WriteLine(  "Application started...."); |  |
| Task task = new Task(DoWork); |  |
| Console.WriteLine(  "Starting thread...."); |  |
| task.Start(); |  |
| \_DoWorkSignaledResetEvent.Wait(); | Console.WriteLine(  "DoWork() started...."); |
| \_DoWorkSignaledResetEvent.Set(); |
| Console.WriteLine(  "Thread executing..."); | \_MainSignaledResetEvent.Wait(); |
| \_MainSignaledResetEvent.Set(); |
| task.Wait(); | Console.WriteLine(  "DoWork() ending...."); |
| Console.WriteLine(  "Thread completed"); |  |
| Console.WriteLine(  "Application exiting...."); |  |

Calling a reset event’s Wait() method (for a ManualResetEvent, it is called WaitOne()) blocks the calling thread until another thread signals and allows the blocked thread to continue. Instead of blocking indefinitely, Wait()/WaitOne() overrides include a parameter, either in milliseconds or as a TimeSpan object, for the maximum amount of time to block. When specifying a timeout period, the return from WaitOne() will be false if the timeout occurs before the reset event is signaled. ManualResetEvent.Wait() also includes a version that takes a cancellation token, allowing for cancellation requests as discussed in Chapter 19.

The difference between ManualResetEventSlim and ManualResetEvent is that the latter uses kernel synchronization by default whereas the former is optimized to avoid trips to the kernel except as a last resort. Thus, ManualResetEventSlim is more performant even though it could possibly use more CPU cycles. For this reason, you should use ManualResetEventSlim in general unless waiting on multiple events or across processes is required.

Notice that reset events implement IDisposable, so they should be disposed when they are no longer needed. In Listing 22.10, we do this via a using statement. (CancellationTokenSource contains a ManualResetEvent, which is why it, too, implements IDisposable.)

Although not exactly the same, System.Threading.Monitor’s Wait() and Pulse() methods provide similar functionality to reset events in some circumstances.

Advanced Topic

Favor ManualResetEvent and Semaphores over AutoResetEvent

There is a third reset event, System.Threading.AutoResetEvent, that, like ManualResetEvent, allows one thread to signal (with a call to Set()) another thread that this first thread has reached a certain location in the code. The difference is that the AutoResetEvent unblocks only one thread’s Wait() call: After the first thread passes through the auto-reset gate, it goes back to locked. With the auto-reset event, it is all too easy to mistakenly code the producer thread with more iterations than the consumer thread. Therefore, use of Monitor’s Wait()/Pulse() pattern or use a semaphore (if fewer than n threads can participate in a particular block) is generally preferred.

In contrast to an AutoResetEvent, the ManualResetEvent won’t return to the unsignaled state until Reset() is called explicitly.

Semaphore/SemaphoreSlim and CountdownEvent

Semaphore and SemaphoreSlim have the same performance differences as ManualResetEvent and ManualResetEventSlim. Unlike ManualResetEvent/ManualResetEventSlim, which provide a lock (like a gate) that is either open or closed, semaphores restrict only N calls to pass within a critical section simultaneously. The semaphore essentially keeps a count of the pool of resources. When this count reaches zero, it blocks any further access to the pool until one of the resources is returned, making it available for the next blocked request that is queued.

CountdownEvent is much like a semaphore, except it achieves the opposite synchronization. That is, rather than protecting further access to a pool of resources that are all used up, the CountdownEvent allows access only once the count reaches zero. Consider, for example, a parallel operation that downloads a multitude of stock quotes. Only when all of the quotes are downloaded can a particular search algorithm execute. The CountdownEvent may be used for synchronizing the search algorithm, decrementing the count as each stock is downloading, and then releasing the search to start once the count reaches zero.

\*\*\*COMP: Insert “Begin 5.0” tab

Notice that SemaphoreSlim and CountdownEvent were introduced with Microsoft .NET Framework 4. In .NET 4.5, the former includes a SemaphoreSlim.WaitAsync() method so that the Task-based Asynchronous Pattern (TAP) can be used when waiting to enter the semaphore.

\*\*\*COMP: Insert “End 5.0” tab

Concurrent Collection Classes

Another series of classes introduced with Microsoft .NET Framework 4 is the concurrent collection classes. These classes are especially designed to include built-in synchronization code so that they can support simultaneous access by multiple threads without concern for race conditions. A list of the concurrent collection classes appears in Table 20.4.

Table 20.4: Concurrent Collection Classes

|  |  |
| --- | --- |
| Collection Class | Description |
| BlockingCollection<T> | Provides a blocking collection that enables producer/consumer scenarios in which producers write data into the collection while consumers read the data. This class provides a generic collection type that synchronizes add and remove operations without concern for the back-end storage (whether a queue, stack, list, or something else). BlockingCollection<T> provides blocking and bounding support for collections that implement the IProducerConsumerCollection<T> interface. |
| ConcurrentBag<T>\* | A thread-safe unordered collection of T type objects. |
| ConcurrentDictionary<TKey, TValue> | A thread-safe dictionary; a collection of keys and values. |
| ConcurrentQueue<T>\* | A thread-safe queue supporting first in, first out (FIFO) semantics on objects of type T. |
| ConcurrentStack<T>\* | A thread-safe stack supporting first in, last out (FILO) semantics on objects of type T. |

\* Collection classes that implement IProducerConsumerCollection<T>.

A common pattern enabled by concurrent collections is support for thread-safe access by producers and consumers. Classes that implement IProducerConsumerCollection<T> (identified by \* in Table 20.4) are specifically designed to provide such support. This enables one or more classes to be pumping data into the collection while a different set reads it out, removing it. The order in which data is added and removed is determined by the individual collection classes that implement the IProducerConsumerCollection<T> interface.

Although it is not built into the out-of-the-box .NET/Dotnet Core Frameworks, an additional immutable collection library is available as a NuGet package reference, called System.Collections.Immutable. The advantage of the immutable collection is that it can be passed freely between threads without concern for either deadlocks or interim updates. As immutable collections cannot be modified, interim updates won’t occur; thus such collections are automatically thread-safe (so there is no need to lock access). For more information, see http://itl.tc/SystemCollectionsImmutable.

Thread Local Storage

In some cases, using synchronization locks can lead to unacceptable performance and scalability restrictions. In other instances, providing synchronization around a particular data element may be too complex, especially when it is added after the original coding.

One alternative solution to synchronization is isolation, and one method for implementing isolation is **thread local storage**. With thread local storage, each thread has its own dedicated instance of a variable. As a result, there is no need for synchronization, as there is no point in synchronizing data that occurs within only a single thread’s context. Two examples of thread local storage implementations are ThreadLocal<T> and ThreadStaticAttribute.

ThreadLocal<T>

Use of thread local storage with Microsoft .NET Framework 4 or later involves declaring a field (or variable, in the case of closure by the compiler) of type ThreadLocal<T>. The result is a different instance of the field for each thread, as demonstrated in Listing 22.11 and Output 20.7. Note that a different instance exists even if the field is static.

Listing 22.11: Using ThreadLocal<T> for Thread Local Storage

using System;

using System.Threading;

public class Program

{

static ThreadLocal<double> \_Count =

new ThreadLocal<double>(() => 0.01134);

public static double Count

{

get { return \_Count.Value; }

set { \_Count.Value = value; }

}

public static void Main()

{

Thread thread = new Thread(Decrement);

thread.Start();

// Increment

for(double i = 0; i < short.MaxValue; i++)

{

Count++;

}

thread.Join();

Console.WriteLine("Main Count = {0}", Count);

}

static void Decrement()

{

Count = -Count;

for (double i = 0; i < short.MaxValue; i++)

{

Count--;

}

Console.WriteLine(

"Decrement Count = {0}", Count);

}

}

Output 20.7

Decrement Count = -32767.01134

Main Count = 32767.01134

As Output 20.7 demonstrates, the value of Count for the thread executing Main() is never decremented by the thread executing Decrement(). For Main()’s thread, the initial value is 0.01134 and the final value is 32767.01134. Decrement() has similar values, except that they are negative. As Count is based on the static field of type ThreadLocal<T>, the thread running Main() and the thread running Decrement() have independent values stored in \_Count.Value.

\*\*\*COMP: Insert “End 4.0” tab

Thread Local Storage with ThreadStaticAttribute

Decorating a static field with a ThreadStaticAttribute, as in Listing 22.12, is a second way to designate a static variable as an instance per thread. This technique has a few caveats relative to ThreadLocal<T>, but it also has the advantage of being available prior to Microsoft .NET Framework 4. (Also, since ThreadLocal<T> is based on the ThreadStaticAttribute, it would consume less memory and give a slight performance advantage given frequently enough repeated small iterations.)

Listing 22.12: Using ThreadStaticAttribute for Thread Local Storage

using System;

using System.Threading;

public class Program

{

[ThreadStatic]

static double \_Count = 0.01134;

public static double Count

{

get { return Program.\_Count; }

set { Program.\_Count = value; }

}

public static void Main()

{

Thread thread = new Thread(Decrement);

thread.Start();

// Increment

for(int i = 0; i < short.MaxValue; i++)

{

Count++;

}

thread.Join();

Console.WriteLine("Main Count = {0}", Count);

}

static void Decrement()

{

for(int i = 0; i < short.MaxValue; i++)

{

Count--;

}

Console.WriteLine("Decrement Count = {0}", Count);

}

}

The results of Listing 22.12 appear in Output 20.8.

Output 20.8

Decrement Count = -32767

Main Count = 32767.01134

As in Listing 22.11, the value of Count for the thread executing Main() is never decremented by the thread executing Decrement(). For Main()’s thread, the initial value is a negative \_Total and the final value is 0. In other words, with ThreadStaticAttribute the value of Count for each thread is specific to the thread and not accessible across threads.

Notice that unlike with Listing 22.11, the value displayed for the decrement count does not have any decimal digits, indicating it was never initialized to 0.01134. Although the value of \_Count is assigned during declaration—private double \_Count = 0.01134 in this example—only the thread static instance associated with the thread running the static constructor will be initialized. In Listing 22.12, only the thread executing Main() will have a thread local storage variable initialized to 0.01134. The value of \_Count that Decrement() decrements will always be initialized to 0 (default(double) since \_Count is an double). Similarly, if a constructor initializes a thread local storage field, only the constructor calling that thread will initialize the thread local storage instance. For this reason, it is a good practice to initialize a thread local storage field within the method that each thread initially calls. However, this is not always reasonable, especially in connection with async, in which different pieces of computation might run on different threads, resulting in unexpectedly differing thread local storage values on each piece.

The decision to use thread local storage requires some degree of cost–benefit analysis. For example, consider using thread local storage for a database connection. Depending on the database management system, database connections are relatively expensive, so creating a connection for every thread could be costly. Similarly, locking a connection so that all database calls are synchronized places a significantly lower ceiling on scalability. Each pattern has its costs and benefits, and the best choice depends largely on the individual implementation.

Another reason to use thread local storage is to make commonly needed context information available to other methods without explicitly passing the data via parameters. For example, if multiple methods in the call stack require user security information, you can pass the data using thread local storage fields instead of as parameters. This technique keeps APIs cleaner while still making the information available to methods in a thread-safe manner. Such an approach requires that you ensure the thread local data is always set—a step that is especially important on Tasks or other thread pool threads because the underlying threads are reused.

\*\*\*COMP: Insert “Begin 5.0” tab

Timers

On occasion, it is necessary to delay code execution for a specific period of time or to register for a notification after a specific period of time. Examples include refreshing the screen at a specific period rather than immediately when frequent data changes occur. One approach to implementing timers is to leverage the async/await pattern of C# 5.0 and the Task.Delay() method added in .NET 4.5. As we pointed out in Chapter 19, one key feature of TAP is that the code executing after an async call will continue in a supported thread context, thereby avoiding any UI cross-threading issues. Listing 22.13 provides an example of how to use the Task.Delay() method.

Listing 22.13: Using Task.Delay() as a Timer

using System;

using System.Threading.Tasks;

public class Pomodoro

{

// ...

private static async Task TickAsync(

System.Threading.CancellationToken token)

{

for(int minute = 0; minute < 25; minute++)

{

DisplayMinuteTicker(minute);

for(int second = 0; second < 60; second++)

{

await Task.Delay(1000);

if(token.IsCancellationRequested) break;

DisplaySecondTicker();

}

if(token.IsCancellationRequested) break;

}

}

}

The call to Task.Delay(1000) will set a countdown timer that triggers after 1 second and executes the continuation code that appears after it.

Fortunately, in C# 5.0, TAP’s use of the synchronization context specifically addressed executing UI-related code exclusively on the UI thread. Prior to that, it was necessary to use specific timer classes that were UI-thread-safe—or could be configured as such. Timers such as System.Windows.Forms.Timer, System.Windows.Threading.DispatcherTimer, and System.Timers.Timer (if configured appropriately) are UI-thread-friendly. Others, such as System.Threading.Timer, are optimized for performance.

\*\*\*COMP: Insert “End 5.0” tab

Advanced Topic

Controlling the COM Threading Model with the STAThreadAttribute

With COM, four different apartment-threading models determine the threading rules relating to calls between COM objects. Fortunately, these rules—and the complexity that accompanied them—have disappeared from .NET as long as the program invokes no COM components. The general approach to handling COM interoperability issues is to place all .NET components within the main, single-threaded apartment by decorating a process’s Main method with the System.STAThreadAttribute. In so doing, it is not necessary to cross apartment boundaries to invoke the majority of COM components. Furthermore, apartment initialization does not occur unless a COM interop call is made. The caveat to this approach is that all other threads (including those of Task) will default to using a multithreaded apartment (MTA). In turn, care needs to be taken when invoking COM components from other threads besides the main one.

COM interop is not necessarily an explicit action by the developer. Microsoft implemented many of the components within the Microsoft .NET Framework by creating a runtime callable wrapper (RCW) rather than rewriting all the COM functionality within managed code. As a result, COM calls are often made unknowingly. To ensure that these calls are always made from a single-threaded apartment, it is generally a good practice to decorate the main method of all Windows Forms executables with the System.STAThreadAttribute.

Summary

In this chapter, we looked at various synchronization mechanisms and saw how a variety of classes are available to protect against race conditions. Coverage included the lock keyword, which leverages System.Threading.Monitor under the covers. Other synchronization classes include System.Threading.Interlocked, System.Threading.Mutext, System.Threading.WaitHandle, reset events, semaphores, and the concurrent collection classes.

In spite of all the progress made in improving multithreaded programming between early versions of .NET and today, synchronization of multithreaded programming remains complicated with numerous pitfalls. To avoid these sand traps, several best practices have been identified. They include consistently acquiring synchronization targets in the same order and wrapping static members with synchronization logic.

Before closing the chapter, we considered the Task.Delay() method, a .NET 4.5 introduced API for implementing a timer based on TAP.

The next chapter investigates another complex .NET technology: that of marshalling calls out of .NET and into unmanaged code using P/Invoke. In addition, it introduces a concept known as unsafe code, which C# uses to access memory pointers directly, as unmanaged code does (e.g., C++).

1. . While at the C# level it’s a local, at the Common Intermediate Language level it’s a field—and fields can be accessed from multiple threads. [↑](#footnote-ref-1)