Nate Cook

Best Practices When Using the Lock Statement

Nate Cook

Dec 3, 2018 • 11 Min read • 73,978 Views

Dec 3, 2018 • 11 Min read • 73,978 Views

C#

The Lock Statement Is Easy to Misuse

233

- The Lock Statement Is Easy to Misuse
- Lock on a Reference Type, Not a Value Type
- Avoid Locking on Anything Publicly Accessible
- Check State at the Beginning of the Lock Block
- Avoid Excessive Locking

The Lock Statement Is Easy to Misuse

When synchronizing access to data in multithreaded C# applications, it is common to use the `lock` statement—in part due to its simple syntax. However, its simplicity and ubiquity have a downside: it is tempting to use the `lock` statement without really considering what it does and what its requirements are in order for it to have the intended effect. Without delving into the more complex forms of thread coordination mentioned in the previous guide in this series,

want to be aware. Let's take a look at some recommendations with respect to the `lock` statement. Knowing such best practices will give you greater confidence when taking advantage of this useful C# keyword.

Lock on a Reference Type, Not a Value Type

One mistake you might make at the beginning is to try to lock on a variable that refers to a value type rather than an object type. For example:

csharp

```
static int myLocker;

static void WriteToFile()
{
  lock (myLocker)
  {
    ...
  }
}
```

Fortunately, the C# compiler will bring this to your attention in the form of a compiler error, telling you that "'int' is not a reference type as required by the lock statement." As you might recall though, the `lock` statement is syntactic sugar for a `try`/`finally` statement with `Monitor.Enter` and `Monitor.Exit`. You

csharp

```csharp
static int myLocker;

static void WriteToFile()
{
  Monitor.Enter(myLocker);
  try
  {
    ...
  }
  finally
  {
    Monitor.Exit(myLocker);
  }
}
```

But no, the above compiles just fine. At runtime however, the call to `Monitor.Exit` throws a `SynchronizationLockException`. Why?

While `lock` is a special C# keyword that allows the compiler to perform additional checks for you, `Monitor.Enter` and `Monitor.Exit` are normal .NET methods that accept any variable of type `object`. The C# language allows for automatic "boxing" (i.e. wrapping) of value types (such as integers, booleans, or structs) into reference types of type `object`, which is what allows us to pass in value types into many .NET methods. Automatic boxing however creates a new object each time boxing is necessary, so the object is different for each invocation of the `Monitor` methods. So, when `Monitor.Exit` attempts to find the lock for the box object

If the previous paragraph didn't make sense to you, don't worry. All of the problems in question can be avoided by ***always*** locking on a reference type and not a value type. Reference types are basically anything that is not a value type; examples include classes and delegates. To keep things even simpler, you can simply instantiate an object with `object myLocker = new object();`. In fact, it is common practice to do so, as we'll see in the next section.

Avoid Locking on Anything Publicly Accessible

The object that you choose to use with the `lock` statement doesn't have much to do, so it can be tempting to lock on some pre-existing object you have available to you. For example, let's say that you are writing a multithreaded web crawler console application and your application has a singleton buffer of some sort that represents a buffer of the data you are going to write to a file.

csharp

```
1  public class CustomBuffer { ... }
2
3  public static class Singletons
4  {
5    public static CustomBuffer LinksBuffer { get; p
6  }
```

When you write the buffer to a file, you need to synchronize access to the file; so, you decide to use `lock`. It might be tempting in this context to lock on the singleton buffer object, e.g.:

csharp

```csharp
static void WriteLinksBufferToFile()
{
  lock (Singletons.LinksBuffer)
  {
    ...
  }
}
```

Now, this might work initially and it has the advantage that you don't have to create a separate variable just for the `lock`. But let's say another developer comes along later who is working on a completely different aspect of the application. This developer is perhaps less conscientious than you are, so when they need to lock, they use the first thing they find. Say they *also* choose to lock on the same `LinksBuffer` singleton, even though what they're synchronizing has nothing to do with links. Can you see why that would cause a potential problem?

As a result of the second developer's decision to lock on the same object, we now have unrelated code waiting on each other. An inefficiency (and perhaps a bug that is difficult to troubleshoot) has been inadvertently introduced. Such a problem

Let's consider another example that is even more insidious. Let's say that you get rid of the `Singletons` class altogether and instantiate a `CustomBuffer` in your web crawler code that is writing links to a file. You declare the `CustomBuffer` as `private`, so you believe it is safe to lock on your instance of `CustomBuffer` because no other code will have access to it. Let's also suppose that the `CustomBuffer` class now lives in a separate assembly and you don't have access to the source code. And, unbeknownst to you, somewhere inside `CustomBuffer` is the following bit of code:

csharp

```
lock (this)
{
  ...
}
```

We now inadvertently have the exact same problem as before: unrelated areas of the application are using the same object for locking! That's because the `this` instance is publicly accessible, at minimum to the declarer (your web crawler code in this case). For that reason, you should avoid locking on `this`, despite its almost irresistible convenience. Furthermore, avoid locking on any object that does literally anything other than locking. Doing so is the only way to

dedicated, private variable of type `object` called `myLocker` or something along those lines is considered to be a best practice. The object's usage is then unambiguous and you and other developers are unlikely to accidentally misuse it in the future. Keep it simple! The following approach is highly recommended and commonly used. Use such a variable *only* for locking.

csharp

```
1  private static object myLocker = new object();
```

Check State at the Beginning of the Lock Block

While working on multithreaded code, it can be easy, as a developer, to forget that *when* something happens is not always under our control. So, when using the `lock` statement, remember that we do not know how long a particular thread will sit at the `lock` statement before entering the block of code. If thread B has a lock when thread A encounters the `lock` statement, it could be seconds or even minutes before thread B releases the lock and allows thread A to acquire the same.

For that reason, it is often the case that

block. In fact, you may need to re-evaluate something that you ***just*** evaluated immediately before the lock statement. For example, consider the case where you have some amount of initialization code that needs to happen only once by whatever thread gets there first. The following approach would be incomplete:

csharp

```csharp
static bool isInitialized;
static object initLock = new object();

static void InitializeIfNeeded()
{
  if (!isInitialized)
  {
    lock (initLock)
    {
      // init code here

      isInitialized = true;
    }
  }
}
```

While partially correct, the above approach may allow multiple initializations, especially if initialization is lengthy. Initialization could be actively in progress by one thread when the second thread encounters the lock. The correct approach would be something like the following:

csharp

```csharp
static bool isInitialized;
static object initLock = new object();

```

```
7    {
8      lock (initLock)
9      {
10       if (!isInitialized)
11       {
12         // init code here
13
14         isInitialized = true;
15       }
16     }
17   }
18 }
```

The second check of `isInitialized` appears duplicative, almost as if it were a typo. But it is absolutely necessary since a thread has no idea what happened between the time the `lock` statement was encountered and the time the lock is eventually acquired. The first outer check of `isInitialized` is, therefore, an optimization; the authoritative check happens ***inside*** the lock. So again, always consider whether you need to check the state of your application after you enter the `lock` statement's block. Often, the answer is yes.

Avoid Excessive Locking

The final pitfall to be aware of when using the `lock` statement is simply using it when you don't need to! Locking in itself is not very expensive from a "number of CPU cycles" perspective but

can't do anything while it is waiting, any unnecessary pauses will significantly affect the overall execution time of your application's work. Consider the following example:

csharp

```csharp
static object myLocker = new object();
static ConcurrentDictionary<string, string> keyVa

static void RemoveAllData()
{
  lock (myLocker)
  {
    keyValueData.Clear();
  }
}
```

In the above example, our lock is redundant because a `ConcurrentDictionary` has its own code to synchronize access to its data. In fact, any collection in the `System.Collections.Concurrent` namespace has mechanisms to ensure access to its data is synchronized. Such collections are considered to be what's called "thread-safe", so you can use them in multithreaded contexts without worrying about race conditions. Any additional locking on your part when accessing thread-safe classes is, therefore, unnecessary. When you use .NET Framework classes for the first time, it's a good idea to check the documentation for information about its thread safety (or lack

help your applications reach their maximum performance.

One Tool in Your Toolbox

The `lock` statement is an extremely useful tool for C# developers writing multithreaded applications. Any amount of asynchronous programming can be challenging, so it is nice to have the `lock` statement's simple syntax available to you. But even the simplest of tools is not without its caveats. By following the above best practices, you will avoid many common problems while having your applications run exactly as you intended.

When it comes to asynchronous programming, the `lock` statement is by no means the only tool available for C# developers. Check out our other C# guides related to async programming for information on some of the others!

SOLUTIONS

Pluralsight Skills (/product/skills)

Pluralsight Flow (/product/flow)

Government (/industries/government)

Gift of Pluralsight (/gift-of-pluralsight)

View Pricing (/pricing)

Contact Sales (/product/contact-sales)

Skill up for free (/product/skills/free)