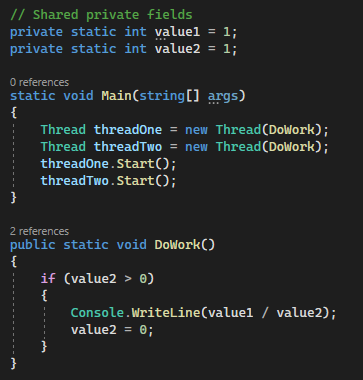
Let’s assume the below example:

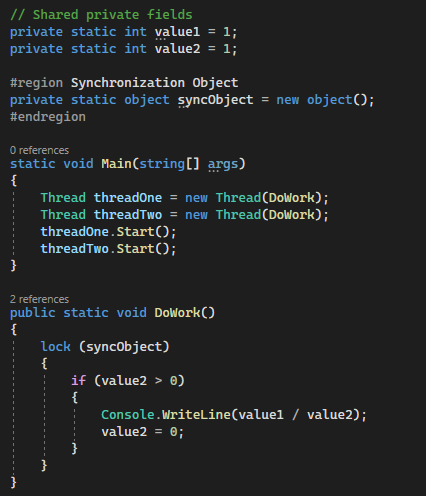


I have two shared variables, both initialized to one. Then we have DoWork method that checks value2 and then divide value1/value2 and then sets value2 to 0. By now, you should be alert to the possibility of a race condition in this code, if two or more threads call DoWork, then it’s perfectly possible for one thread to set value2 to 0, just as another thread is busy executing the Console.WriteLine method.

The result, division by zero exception. To fix this code, I need to make the check and the division one atomic operation by adding a lock statement. The lock statement needs a synchronization object to work with. This could be any reference type and there are no restrictions. After adding as shown in the image below, the code is now guarded against a race condition.

We can say that the code is thread safe, meaning it can safely be called from multiple threads simultaneously without crashing. Locking a section of code is a very fast operation. On a typical modern CPU, the operation takes around 20 nanoseconds to complete. So, there’s no need to worry about the performance overhead of locking a section of the code.

The lock statement is what we called syntactic sugar. This means that the C# compiler will actually expand the statement to a larger block of code and lock is simply a convenience provided by the compiler. So, we don’t have to type all of that code every time. The code produced by the compiler is very simple, and you can easily type it by hand if you want.



Below is the DoWork method, with the code of the previous example, with the possible division by zero, but now with the expanded lock statement. As you can see, a lock is nothing more than a call to Monitor.Enter, the Monitor class in C# provides for critical sections. The Enter method enters the critical section and the Exit method exits.

There’s an extra Boolean variable here called lockTaken, this variable acts as a signal to the finally block. If the Monitor was entered successfully, the Boolean will be set to true and the final block will exit the critical section, but if for whatever reason, the critical section could not be entered successfully, then the Boolean will be false and the finally block will do nothing.

This setup with the Boolean field prevents a lock leak where a critical section is entered but never exited because the corresponding Monitor.Exit is not called. The advantage of typing Monitor.Enter and Monitor.Exit directly is that I can now use other features of the Monitor class too, for example there’s Monitor.TryEnter method that I can use instead.

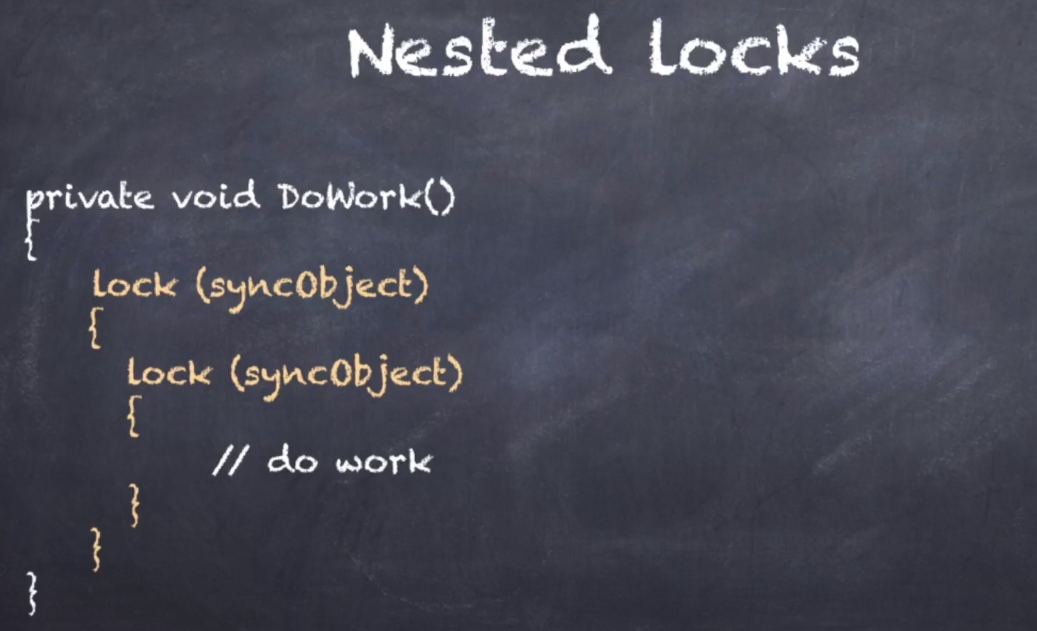
TryEnter expects a timeout as the second parameter, either in milliseconds or as a timespan value and it returns a Boolean. True, if the answer was successful and false if it was not and the operation timed out. Providing a timeout is very important because you never know if the critical section will be released.

If another thread is stuck in an infinite loop while holding the critical section, then your thread will wait indefinitely. A timeout will help you break out of this deadlock situation. You’ve seen that the lock statement requires a synchronization object. I created a special private object variable for that purpose, but you can also lock on any reference type you like, including the “this” value or the type object.

So, you might be wondering if there are any best practices in choosing the synchronization object. And in fact, there are. You’re advised to always use the private field as a synchronization object. The reason for this is simple. Consider for moment that you lock on a public field because the field is public, another thread could also lock on that same field.

Now, you have an unexpected dependency between two threads that might lead to both threads blocking and waiting for each other. This can easily happen when you always lock on this value. But when you lock on a unique private field that you create specifically for the occasion, then you prevent any other thread to lock on that same object.

So, now the unexpected dependency between threads can never happen. It’s just another safety net to protect your code from unexpected results. Finally, let me show you another capability of the lock statement (Nested Locks). You can arbitrarily nest locks inside each other like shown below.



The thread gains access to the critical section when the outermost lock succeeds and each subsequent lock is simply stacked on top of the first one. The critical section is released when all stacked locks have exited. In terms of the Monitor class, this means you can have any number of Monitor.Enter statements and the critical section is only unlocked when a matching number of Monitor.Exit statements have executed.

Nesting locks are useful when you are calling a method from within a critical section. To demonstrate, lets consider divide by zero code again, but this time I have put the actual division in another method. I can lock the original code in the DoWork method, but then call into another method from inside the critical section and have that method also set up its own critical section.

Both critical sections are stacked, so I can safely return from the innermost method while still retaining the lock. Only when I exit the outer critical section in the DoWork method the locked is released. So, in summary, don’t worry about nesting locks, but just make sure you use the same synchronization object for each critical section.

If you do, you can neatly stack the critical sections and the lock will not get released until you return from the first method.

**What have you learned?**

