C# Interlocked

Interlocked helps with threaded programs. It safely changes the value of a shared variable from multiple threads in a C# program. This is possible with the lock statement. But you can instead use the Interlocked type for simpler and faster code.

Lock Statement

Interlocked.Add

When using Interlocked, forget all you know about addition, subtraction and assignment operators. Instead, you will use the Add, Increment, Decrement, Exchange and CompareExchange methods. These change the operations to be atomic. This means no operations can be performed on the value during the call.

Next:

This example familiarizes us with the syntax of Interlocked.

Program that uses Interlocked.Add: C#

```
using System;
using System.Threading;

class Program
{
    static int _value;

    static void Main()
    {
        Thread thread1 = new Thread(new ThreadStart(A));
        Thread thread2 = new Thread(new ThreadStart(A));
        thread1.Start();
        thread2.Start();
        thread1.Join();
        thread2.Join();
        Console.WriteLine(Program._value);
    }
```

```
    static void A()
    {
        // Add one.
        Interlocked.Add(ref Program._value, 1);
    }
}
```

Output

```
2
```

When running this program, there is no chance that thread 1 or thread 2 will read the value of the field before the other thread has written to it. What could happen without Interlocked is this: both threads read the value, then both change it afterwards. The result would be 1 not 2 and the programmer would be confused.

Thread Join Method

ThreadStart

ParameterizedThreadStart

Increment, Decrement

Continuing on, the Interlocked type offers the Increment and Decrement methods as well. These methods are much the same as Add, but they use the value of 1 as an implicit argument.

Program that uses Interlocked.Increment and Decrement: C#

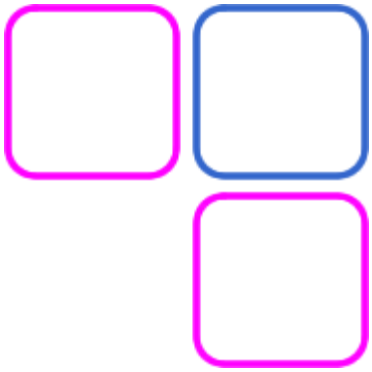
```
using System;
using System.Threading;

class Program
{
    static int _value;

    static void Main()
    {
        Thread thread1 = new Thread(new ThreadStart(A));
        Thread thread2 = new Thread(new ThreadStart(A));
        thread1.Start();
        thread2.Start();
        thread1.Join();
        thread2.Join();
        Console.WriteLine(Program._value);
    }

    static void A()
    {
        // Add one then subtract two.
        Interlocked.Increment(ref Program._value);
```

```
        Interlocked.Decrement(ref Program._value);
        Interlocked.Decrement(ref Program._value);
    }
}
```

Output

```
-2
```

Exchange



This example is more interesting: we see first how the Interlocked.Exchange method works. Exchange is essentially an assignment: the value is changed to the argument.

Next:

CompareExchange is used—this encodes both a conditional and an assignment in the same statement. If the value equals the second argument, it is changed to the third argument. The original value is then returned.

Program that uses Interlocked: C#

```
using System;
using System.Threading;

class Program
{
    static long _value1;

    static void Main()
    {
        Thread thread1 = new Thread(new ThreadStart(A));
        thread1.Start();
        thread1.Join();

        // Written [2]
        Console.WriteLine(Interlocked.Read(ref Program._value1));
    }

    static void A()
    {
        // Replace value with 10.
        Interlocked.Exchange(ref Program._value1, 10);

        // CompareExchange: if 10, change to 20.
        long result = Interlocked.CompareExchange(ref Program._value1, 20, 10);

        // Returns original value from CompareExchange [1]
```

```
        Console.WriteLine(result);
    }
}
```

Output

```
10
20
```

Performance

While calling Interlocked methods seems simpler in programs, does it actually perform faster than using the lock construct? In this investigation, we test a lock before an integer increment in the first loop. Then we test a call to Interlocked.Increment in the second loop.

Program that tests Interlocked performance: C#

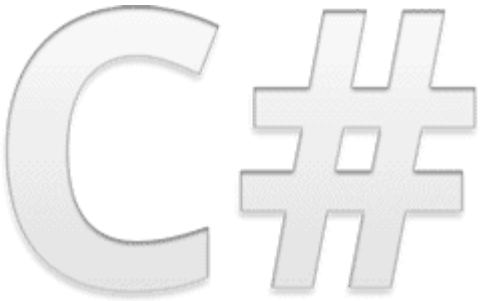
```
using System;
using System.Diagnostics;
using System.Threading;

class Program
{
    static object _locker = new object();
    static int _test;
    const int _max = 10000000;
    static void Main()
    {
        var s1 = Stopwatch.StartNew();
        for (int i = 0; i < _max; i++)
        {
            lock (_locker)
            {
                _test++;
            }
        }
        s1.Stop();
        var s2 = Stopwatch.StartNew();
        for (int i = 0; i < _max; i++)
        {
            Interlocked.Increment(ref _test);
        }
        s2.Stop();
        Console.WriteLine(_test);
        Console.WriteLine(((double)(s1.Elapsed.TotalMilliseconds * 1000000) /
            _max).ToString("0.00 ns"));
        Console.WriteLine(((double)(s2.Elapsed.TotalMilliseconds * 1000000) /
            _max).ToString("0.00 ns"));
        Console.Read();
    }
}
```

Result

```
20000000
40.02 ns
 6.40 ns [Interlocked.Increment]
```

We can see that the result is correct because the value printed is equal to the total number of increment operations. More interesting, though, are the timings. Interlocked.Increment was several times faster, requiring only 6 nanoseconds versus 40 nanoseconds for the lock construct.

Summary



Once you get past the awkward syntax of the Interlocked methods, they can make writing multithreaded C# programs easier. You can avoid locking on certain values.

And:

This results in fewer lines of code. It even provides a performance boost when only a value type is being accessed.