**Part 93 - Protecting shared resources from concurrent access in multithreading.**

In this video we will discuss,   
**1. What happens if shared resources are not protected from concurrent access in multithreaded program**  
**2. How to protect shared resources from concurrent access**

**What happens if shared resources are not protected from concurrent access in multithreaded program**  
The output or behaviour of the program can become inconsistent. Let us understand this with an example.

usingSystem**;**

classProgram

**{**

staticintTotal=0**;**

publicstaticvoidMain**()**

**{**

AddOneMillion**();**

AddOneMillion**();**

AddOneMillion**();**

Console.WriteLine**(**"Total = "+Total**);**

**}**

publicstaticvoidAddOneMillion**()**

**{**

for**(**inti=1**;**i<=1000000**;**i++**)**

**{**

Total++**;**

**}**

**}**

**}**

The above program is a single-threaded program. In the **Main**() method, **AddOneMillion()**method is called 3 times, and it updates the Total field correctly as expected, and finally prints the correct total i.e 3000000.  
  
**Now, let's rewrite the program using multiple threads.**

usingSystem**;**

usingSystem.Threading**;**

classProgram

**{**

staticintTotal=0**;**

publicstaticvoidMain**()**

**{**

Threadthread1=newThread**(**Program.AddOneMillion**);**

Threadthread2=newThread**(**Program.AddOneMillion**);**

Threadthread3=newThread**(**Program.AddOneMillion**);**

thread1.Start**();**

thread2.Start**();**

thread3.Start**();**

thread1.Join**();**

thread2.Join**();**

thread3.Join**();**

Console.WriteLine**(**"Total = "+Total**);**

**}**

publicstaticvoidAddOneMillion**()**

**{**

for**(**inti=1**;**i<=1000000**;**i++**)**

**{**

Total++**;**

**}**

**}**

**}**

Every time we run the above program, we get a different output. The inconsistent output is because the Total field which is a shared resource is not protected from concurrent access by multiple threads. The operator ++ is not thread safe. There are several ways to fix this. Let's explore 2 of the options.  
  
Using **Interlocked.Increment**() method: Modify **AddOneMillion()**method as shown below. The Interlocked.Increment() Method, increments a specified variable and stores the result, as an atomic operation

publicstaticvoidAddOneMillion**()**

**{**

for**(**inti=1**;**i<=1000000**;**i++**)**

**{**

Interlocked.Increment**(**refTotal**);**

**}**

**}**

**The other option is to use a lock.**

staticobject\_lock=newobject**();**

publicstaticvoidAddOneMillion**()**

**{**

for**(**inti=1**;**i<=1000000**;**i++**)**

**{**

lock**(**\_lock**)**

**{**

Total++**;**

**}**

**}**

**}**

**Which option is better?**  
**From a performance perspective using Interlocked class is better over using locking.**Locking locks out all the other threads except a single thread to read and increment the Total variable. This will ensure that the Total variable is updated safely. The downside is that since all the other threads are locked out, there is a performance hit.  
  
The Interlocked class can be used with addition/subtraction (increment, decrement, add, etc.) on and int or long field. The Interlocked class has methods for incrementing, decrementing, adding, and reading variables atomically.

**The following code prints the time taken in ticks.**1 millisecond consists of 10000 ticks.

publicstaticvoidMain**()**

**{**

Stopwatchstopwatch=Stopwatch.StartNew**();**

Threadthread1=newThread**(**Program.AddOneMillion**);**

Threadthread2=newThread**(**Program.AddOneMillion**);**

Threadthread3=newThread**(**Program.AddOneMillion**);**

thread1.Start**();**

thread2.Start**();**

thread3.Start**();**

thread1.Join**();**

thread2.Join**();**

thread3.Join**();**

Console.WriteLine**(**"Total = "+Total**);**

stopwatch.Stop**();**

Console.WriteLine**(**"Time Taken in Ticks = "+stopwatch.ElapsedTicks**);**

**}**

**Please Note:** You can use the TimeSpan object to find ticks per second, ticks per millisecond etc. Stopwatch class is in System.Diagnostics namespace.