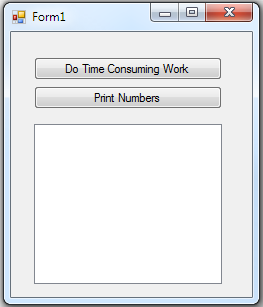
**Multithreading in C#**

**we will discuss**  
1. What is a process and a thread  
2. Simple multithreading example  
  
  
  
**Before we discuss multithreading, first let's understand the following terms**  
**1. Process**- Process is what the operating system uses to facilitate the execution of a program by providing the resources required. Each process has a unique process Id associated with it. You can view the process within which a program is being executed using windows task manager.  
**2. Thread**- Thread is a light weight process. A process has at least one thread which is commonly called as main thread which actually executes the application code. A single process can have multiple threads.  
  
**Please Note:**All the threading related classes are present in System.Threading namespace.  
  
**Multithreading Example:** Create a new windows forms application with 2 buttons and a listbox control as shown in the image below, and set the following properties.  
  
  
**For the first button control, set**  
Name = btnTimeConsumingWork  
Text = Do Time Consuming Work  
  
**For the second button control, set**  
Name = btnPrintNumbers  
Text = Print Numbers  
  
Double click on each of the buttons to generate their respective click event handlers.  
  
**For the listbox control, set**  
Name = listBoxNumbers  
  
**Copy and paste the following code in Form1.cs file**

usingSystem**;**

usingSystem.Threading**;**

usingSystem.Windows.Forms**;**

namespaceThreadingExample

**{**

publicpartialclassForm1**:**Form

**{**

publicForm1**()**

**{**

InitializeComponent**();**

**}**

privatevoidbtnTimeConsumingWork\_Click**(**objectsender**,**EventArgse**)**

**{**

btnTimeConsumingWork.Enabled=false**;**

btnPrintNumbers.Enabled=false**;**

DoTimeConsumingWork**();**

btnTimeConsumingWork.Enabled=true**;**

btnPrintNumbers.Enabled=true**;**

**}**

privatevoidDoTimeConsumingWork**()**

**{**

// Make the thread sleep, to introduce artifical latency

Thread.Sleep**(**5000**);**

**}**

privatevoidbtnPrintNumbers\_Click**(**objectsender**,**EventArgse**)**

**{**

for**(**inti=1**;**i<=10**;**i++**)**

**{**

listBoxNumbers.Items.Add**(**i**);**

**}**

**}**

**}**

**}**

**1.**At this point if we run the program, **one thread**is automatically created. This thread is called as the **Main thread**or **UI thread.** This is the thread that is responsible for doing all the work.   
**2.** Now when you click **"Do Time Consuming Work"**, the first 2 lines of code to disable the button is executed. As a result both the buttons are disabled.   
**3.** **DoTimeConsumingWork()**method is called next, and at this point the application is unresponsive as it is waiting for the method to complete. Note that the buttons are still disabled and you cannot click on any of them.  
**4.**Finally, once the **DoTimeConsumingWork()**method completes the buttons are enabled and the application is responsive.  
  
Now change the code in **btnTimeConsumingWork\_Click()**event handler method as shown below.

privatevoidbtnTimeConsumingWork\_Click**(**objectsender**,**EventArgse**)**

**{**

btnTimeConsumingWork.Enabled=false**;**

btnPrintNumbers.Enabled=false**;**

// Create another THREAD to offload the work of

// executing the time consuming method to it.

// As a result the UI thread, is free to execute the

// rest of the code and our application is more responsive.

ThreadbackGroundThread=newThread**(**DoTimeConsumingWork**);**

backGroundThread.Start**();**

//DoTimeConsumingWork();

btnTimeConsumingWork.Enabled=true**;**

btnPrintNumbers.Enabled=true**;**

**}**

So one of the benefits of **multithreaded programming**is that it makes your application more responsive.

### Advantages and disadvantages of multithreading

**Advantages of multithreading:**  
**1.** To maintain a responsive user interface  
**2.** To make efficient use of processor time while waiting for I/O operations to complete.  
**3.** To split large, CPU-bound tasks to be processed simultaneously on a machine that has multiple CPUs/cores.   
**Disadvantages of multithreading:**  
**1.** On a single-core/processor machine threading can affect performance negatively as there is overhead involved with context-switching.  
**2.** Have to write more lines of code to accomplish the same task.  
**3.** Multithreaded applications are difficult to write, understand, debug and maintain.  
  
**Please Note:** Only use multithreading when the advantages of doing so outweigh the disadvantages.

### ThreadStart delegate

**Let us understand the purpose of ThreadStart delegate with an example.**

usingSystem**;**

usingSystem.Threading**;**

namespaceThreadStartDelegateExample

**{**

classProgram

**{**

publicstaticvoidMain**()**

**{**

ThreadT1=newThread**(**Number.PrintNumbers**);**

T1.Start**();**

**}**

**}**

classNumber

**{**

publicstaticvoidPrintNumbers**()**

**{**

for**(**inti=1**;**i<=10**;**i++**)**

**{**

Console.WriteLine**(**i**);**

**}**

**}**

**}**

**}**

**In the example above to create a THREAD,** we created an instance of Thread class and to it's constructor we have passed the name of the function that we want the thread to execute.  
ThreadT1=newThread**(**Number.PrintNumbers**);**  
  
**We can rewrite the above line using ThreadStart delegate as shown below.**  
ThreadT1=newThread**(**newThreadStart**(**Number.PrintNumbers**));**  
  
**Why a delegate need to be passed as a parameter to the Thread class constructor?**  
The purpose of creating a Thread is to execute a function. A delegate is a type safe function pointer, meaning it points to a function that the thread has to execute. In short, all threads require an entry point to start execution. Any thread you create will need an explicitly defined entry point i.e a pointer to the function where they should begin execution. So threads always require a delegate.  
  
**In the code below, we are not explicitly creating the ThreadStart delegate, then how is it working here?**  
ThreadT1=newThread**(**Number.PrintNumbers**);**  
  
It's working in spite of not creating the ThreadStart delegate explicitly because the framework is doing it automatically for us.  
  
**We can also rewrite the same line using delagate() keyword as shown below.**

ThreadT1=newThread**(**delegate**() {**Number.PrintNumbers**(); });**

**We can also rewrite the same line using lambda expression as shown below.**

ThreadT1=newThread**(()**=>Number.PrintNumbers**());**

**Thread function need not be a static function always.** It can also be a non-static function. In case of non-static function we have to create an instance of the class. An example is shown below.

classProgram

**{**

publicstaticvoidMain**()**

**{**

Numbernumber=newNumber**();**

ThreadT1=newThread**(**number.PrintNumbers**);**

T1.Start**();**

**}**

**}**

classNumber

**{**

publicvoidPrintNumbers**()**

**{**

for**(**inti=1**;**i<=10**;**i++**)**

**{**

Console.WriteLine**(**i**);**

**}**

**}**

**}**

### ParameterizedThreadStart delegate

Use **ParameterizedThreadStart delegate**to pass data to the thread function. Here is an example that shows the usage of ParameterizedThreadStart delegate.

usingSystem**;**

usingSystem.Threading**;**

namespaceThreadStartDelegateExample

**{**

classProgram

**{**

publicstaticvoidMain**()**

**{**

Console.WriteLine**(**"Please enter the target number"**);**

objecttarget=Console.ReadLine**();**

// Create an instance ParameterizedThreadStart delegate

ParameterizedThreadStartparameterizedThreadStart=

newParameterizedThreadStart**(**Number.PrintNumbers**);**

ThreadT1=newThread**(**parameterizedThreadStart**);**

// Pass the traget number to the start function, which

// will then be passed automatically to PrintNumbers() function

T1.Start**(**target**);**

**}**

**}**

classNumber

**{**

publicstaticvoidPrintNumbers**(**objecttarget**)**

**{**

intnumber=0**;**

if**(**int.TryParse**(**target.ToString**(),**outnumber**))**

**{**

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

**{**

Console.WriteLine**(**i**);**

**}**

**}**

**}**

**}**

**}**

**The code in the Main() function can also be written as shown below.**

publicstaticvoidMain**()**

**{**

Console.WriteLine**(**"Please enter the target number"**);**

objecttarget=Console.ReadLine**();**

ThreadT1=newThread**(**Number.PrintNumbers**);**

T1.Start**(**target**);**

**}**

**Here we are not explicitly creating an instance of ParameterizedThreadStart delegate. Then how is it working?**  
It's working because, the compiler implicitly converts newThread**(**Number.PrintNumbers**)**  to newThread**(**newParameterizedThreadStart**(**Number.PrintNumbers**))**.  
  
**When to use ParameterizedThreadStart over ThreadStart delegate?**  
Use **ParameterizedThreadStart**delegate if you have some data to pass to the Thread function, otherwise just use **ThreadStart**delegate.  
  
**Please note:**Using **ParameterizedThreadStart**delegate and **Thread.Start(Object)**method to pass data to the Thread function is not type safe as they operate on object datatype and any type of data can be passed. If you try to change the data type of the target parameter of PrintNumbers() function from object to int, a compiler error will be raised as the signature of PrintNumbers() function does not match with the signature of ParameterizedThreadStart delegate.

### Passing data to the Thread function in a type safe manner

**To pass data to the Thread function in a type safe manner**, encapsulate the thread function and the data it needs in a helper class and use the ThreadStart delegate to execute the thread function. An example is shown below.

usingSystem**;**

usingSystem.Threading**;**

namespaceThreadingExample

**{**

classProgram

**{**

publicstaticvoidMain**()**

**{**

// Prompt the user for the target number

Console.WriteLine**(**"Please enter the target number"**);**

// Read from the console and store it in target variable

inttarget=Convert.ToInt32**(**Console.ReadLine**());**

// Create an instance of the Number class, passing it

// the target number that was read from the console

Numbernumber=newNumber**(**target**);**

// Specify the Thread function

ThreadT1=newThread**(**newThreadStart**(**number.PrintNumbers**));**

// Alternatively we can just use Thread class constructor as shown below

// Thread T1 = new Thread(number.PrintNumbers);

T1.Start**();**

**}**

**}**

// Number class also contains the data it needs to print the numbers

classNumber

**{**

int\_target**;**

// When an instance is created, the target number needs to be specified

publicNumber**(**inttarget**)**

**{**

// The targer number is then stored in the class private variable \_target

this.\_target=target**;**

**}**

// Function prints the numbers from 1 to the traget number that the user provided

publicvoidPrintNumbers**()**

**{**

for**(**inti=1**;**i<=\_target**;**i++**)**

**{**

Console.WriteLine**(**i**);**

**}**

**}**

**}**

**}**

### Retrieving data from Thread function using callback method

usingSystem**;**

usingSystem.Threading**;**

namespaceThreadStartDelegateExample

**{**

// Step 1: Create a callback delegate. The actual callback method

// signature should match with the signature of this delegate.

publicdelegatevoidSumOfNumbersCallback**(**intsumOfNumbers**);**

// Step 2: Create Number class to compute the sum of numbers and

// to call the callback method

classNumber

**{**

// The traget number this class needs to compute the sum of numbers

int\_target**;**

// Delegate to call when the the Thread function completes   
// computing the sum of numbers

SumOfNumbersCallback\_callbackMethod**;**

// Constructor to initialize the target number and the callback delegateinitialize

publicNumber**(**inttarget**,**SumOfNumbersCallbackcallbackMethod**)**

**{**

this.\_target=target**;**

this.\_callbackMethod=callbackMethod**;**

**}**

// This thread function computes the sum of numbers and then invokes

// the callback method passing it the sum of numbers

publicvoidComputeSumOfNumbers**()**

**{**

intsum=0**;**

for**(**inti=1**;**i<=\_target**;**i++**)**

**{**

sum=sum+i**;**

**}**

if**(**\_callbackMethod!=null**)**

**{**

\_callbackMethod**(**sum**);**

**}**

**}**

**}**

// Step 3: This class consumes the Number class created in Step 2

classProgram

**{**

// Callback method that will receive the sum of numbers

publicstaticvoidPrintSumOfNumbers**(**intsum**)**

**{**

Console.WriteLine**(**"Sum of numbers is "+sum**);**

**}**

publicstaticvoidMain**()**

**{**

// Prompt the user for the target number

Console.WriteLine**(**"Please enter the target number"**);**

// Read from the console and store it in target variable

inttarget=Convert.ToInt32**(**Console.ReadLine**());**

// Create an instance of callback delegate and to it's constructor

// pass the name of the callback function (PrintSumOfNumbers)

SumOfNumbersCallbackcallbackMethod=newSumOfNumbersCallback**(**PrintSumOfNumbers**);**

// Create an instance of Number class and to it's constrcutor pass the target

// number and the instance of callback delegate

Numbernumber=newNumber**(**target**,**callbackMethod**);**

// Create an instance of Thread class and specify the Thread function to invoke

ThreadT1=newThread**(**newThreadStart**(**number.ComputeSumOfNumbers**));**

T1.Start**();**

**}**

**}**

**}**

### Significance of Thread.Join and Thread.IsAlive functions

**Join**blocks the current thread and makes it wait until the thread on which Join method is invoked completes. Join method also has a overload where we can specify the timeout. If we don't specify the timeout the calling thread waits indefinitely, until the thread on which Join() is invoked completes. This overloaded Join(int millisecondsTimeout) method returns boolean. True if the thread has terminated otherwise false. Join is particularly useful when we need to wait and collect result from a thread execution or if we need to do some cleanup after the thread has completed.  
  
**IsAlive returns boolean.**True if the thread is still executing otherwise false.  
  
  
  
**Program code used in the demo:**

usingSystem**;**

usingSystem.Threading**;**

namespaceThreadingExample

**{**

classProgram

**{**

publicstaticvoidMain**()**

**{**

Console.WriteLine**(**"Main Started"**);**

ThreadT1=newThread**(**Program.Thread1Function**);**

T1.Start**();**

ThreadT2=newThread**(**Program.Thread2Function**);**

T2.Start**();**

if**(**T1.Join**(**1000**))**

**{**

Console.WriteLine**(**"Thread1Function completed"**);**

**}**

else

**{**

Console.WriteLine**(**"Thread1Function hot not completed in 1 second"**);**

**}**

T2.Join**();**

Console.WriteLine**(**"Thread2Function completed"**);**

for**(**inti=1**;**i<=10**;**i++**)**

**{**

if**(**T1.IsAlive**)**

**{**

Console.WriteLine**(**"Thread1Function is still doing it's work"**);**

Thread.Sleep**(**500**);**

**}**

else

**{**

Console.WriteLine**(**"Thread1Function Completed"**);**

break**;**

**}**

**}**

Console.WriteLine**(**"Main Completed"**);**

**}**

publicstaticvoidThread1Function**()**

**{**

Console.WriteLine**(**"Thread1Function started"**);**

Thread.Sleep**(**5000**);**

Console.WriteLine**(**"Thread1Function is about to return"**);**

**}**

publicstaticvoidThread2Function**()**

**{**

Console.WriteLine**(**"Thread2Function started"**);**

**}**

**}**

**}**

### Protecting shared resources from concurrent access in multithreading

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.

### Difference between Monitor and lock in C#

Both **Monitor class**and **lock**provides a mechanism that synchronizes access to objects. lock is the shortcut for Monitor.Enter with try and finally.  
  
**This means that, the following code**

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

publicstaticvoidAddOneMillion**()**

**{**

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

**{**

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

**{**

Total++**;**

**}**

**}**

**}**

**can be rewritten as shown below:**

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

publicstaticvoidAddOneMillion**()**

**{**

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

**{**

// Acquires the exclusive lock

Monitor.Enter**(**\_lock**);**

try

**{**

Total++**;**

**}**

finally

**{**

// Releases the exclusive lock

Monitor.Exit**(**\_lock**);**

**}**

**}**

**}**

**In C# 4, it is implement slightly differently as shown below**

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

publicstaticvoidAddOneMillion**()**

**{**

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

**{**

boollockTaken=false**;**

// Acquires the exclusive lock

Monitor.Enter**(**\_lock**,**reflockTaken**);**

try

**{**

Total++**;**

**}**

finally

**{**

// Releases the exclusive lock

if**(**lockTaken**)**

Monitor.Exit**(**\_lock**);**

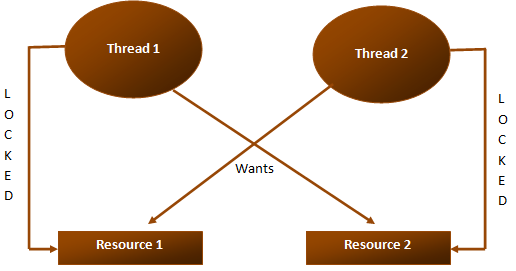
**}**

**}**

**}**

**So, in short, lock is a shortcut and it's the option for the basic usage.** If you need more control to implement advanced multithreading solutions using TryEnter() Wait(), Pulse(), & PulseAll() methods, then the Monitor class is your option.

### Deadlock in a multithreaded program

we will discuss, why and how a deadlock can occur in multithreading with an example.  
  
  
  
**Scenario when a deadlock can occur**  
**Let's say we have 2 threads**  
**a)** Thread 1  
**b)** Thread 2  
  
**and 2 resources**  
**a)** Resource 1  
**b)** Resource 2  
  
**Thread 1 has already acquired a lock on Resource 1 and wants to acquire a lock on Resource 2.** At the same time Thread 2 has already acquired a lock on Resource 2 and wants to acquire a lock on Resource 1. Two threads never give up their locks, hence a deadlock.  
  
  
  
**Example code used in the demo**

usingSystem**;**

usingSystem.Threading**;**

publicclassProgram

**{**

publicstaticvoidMain**()**

**{**

Console.WriteLine**(**"Main Started"**);**

AccountaccountA=newAccount**(**101**,**5000**);**

AccountaccountB=newAccount**(**102**,**3000**);**

AccountManageraccountManagerA=new

AccountManager**(**accountA**,**accountB**,**1000**);**

ThreadT1=newThread**(**accountManagerA.Transfer**);**

T1.Name="T1"**;**

AccountManageraccountManagerB=new

AccountManager**(**accountB**,**accountA**,**2000**);**

ThreadT2=newThread**(**accountManagerB.Transfer**);**

T2.Name="T2"**;**

T1.Start**();**

T2.Start**();**

T1.Join**();**

T2.Join**();**

Console.WriteLine**(**"Main Completed"**);**

**}**

**}**

publicclassAccount

**{**

double\_balance**;**

int\_id**;**

publicAccount**(**intid**,**doublebalance**)**

**{**

this.\_id=id**;**

this.\_balance=balance**;**

**}**

publicintID

**{**

get

**{**

return\_id**;**

**}**

**}**

publicvoidWithdraw**(**doubleamount**)**

**{**

\_balance-=amount**;**

**}**

publicvoidDeposit**(**doubleamount**)**

**{**

\_balance+=amount**;**

**}**

**}**

publicclassAccountManager

**{**

Account\_fromAccount**;**

Account\_toAccount**;**

double\_amountToTransfer**;**

publicAccountManager**(**AccountfromAccount**,**

AccounttoAccount**,**doubleamountToTransfer**)**

**{**

this.\_fromAccount=fromAccount**;**

this.\_toAccount=toAccount**;**

this.\_amountToTransfer=amountToTransfer**;**

**}**

publicvoidTransfer**()**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name

+" trying to acquire lock on "

+\_fromAccount.ID.ToString**());**

lock**(**\_fromAccount**)**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name

+" acquired lock on "

+\_fromAccount.ID.ToString**());**

Console.WriteLine**(**Thread.CurrentThread.Name

+" suspended for 1 second"**);**

Thread.Sleep**(**1000**);**

Console.WriteLine**(**Thread.CurrentThread.Name+

" back in action and trying to acquire lock on "

+\_toAccount.ID.ToString**());**

lock**(**\_toAccount**)**

**{**

\_fromAccount.Withdraw**(**\_amountToTransfer**);**

\_toAccount.Deposit**(**\_amountToTransfer**);**

**}**

**}**

**}**

**}**