**Multithreading in C#**

**Multithreading in C# with Examples**

In this article, I am going to discuss **Multithreading in C#** with examples. Multithreading is one of the most important concepts in C# that you need to understand as a developer. In this and few upcoming articles, I am going to cover all the concepts of C# Multithreading with examples. As part of this article, I am going to cover the following pointers.

1. **What is Multitasking?**
2. **How the operating system executes multiple applications at a time?**
3. **What is Thread?**
4. **Understanding the Thread class.**
5. **What are the drawbacks of Single-Threaded Applications?**
6. **How to overcome the drawbacks of the Single-Threaded Application using C# Multithreading with Examples?**

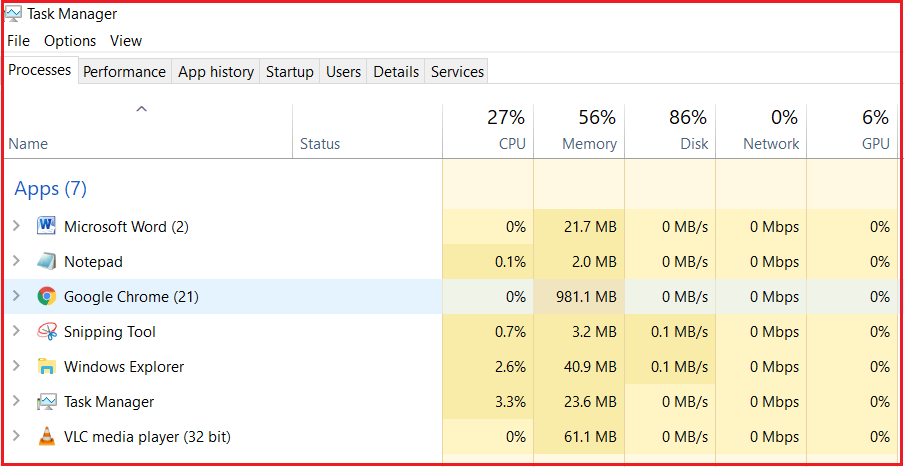
**What is Multitasking?**

Before understanding the concept of Multithreading in C#, let us first understand multitasking. Windows operating system is a multitasking operating system. It means it has the ability to run multiple applications at the same time. For example, in my machine, I can open the Google Chrome Browser, Microsoft word document, Notepad, VLC Media Player, Windows Explorer, etc. at the same time. This is possible because in my machine I have installed the Windows operating system and the Windows operating system is a multitasking operating system.

**How the operating system executes multiple applications at a time?**

To execute all the above applications, the operating system internally makes use of processes. A process is a part of the operating system (or a component under the operating system) which is responsible for executing the program or application. So, to execute each and every program or application, there will be a process.

You can see this using the Task Manager. Just right-click on the Taskbar and click on the Task Manager option which will open the Task Manager window. From that window, just click on the “Processes” button as shown below.



As you can see from the above image, each application is executing by one corresponding process. Along the same line, there are also multiple processes that are running in the background which are known as the background processes. These background processes are known as windows services and the Operating system runs a lot of windows services in the background.

So, we have an operating system and under the operating system, we have processes that running our applications. So under the process, an application runs. To run the code of an application the process will make use of a concept called Thread.

**What is Thread?**

Generally, a Thread is a lightweight process. In simple words, we can say that a Thread is a unit of a process that is responsible for executing the application code. So, every program or application has some logic or code and to execute that logic or code, Thread comes into the picture.

By default, every process has at least one thread which is responsible for executing the application code and that thread is called as Main Thread. So, every application by default is a single-threaded application.

**Note:** All the threading related classes in C# belong to the **System.Threading** namespace.

**Let us see an example to understand Threading.**

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"Welcome to Dotnet world!"**)**;

**}**

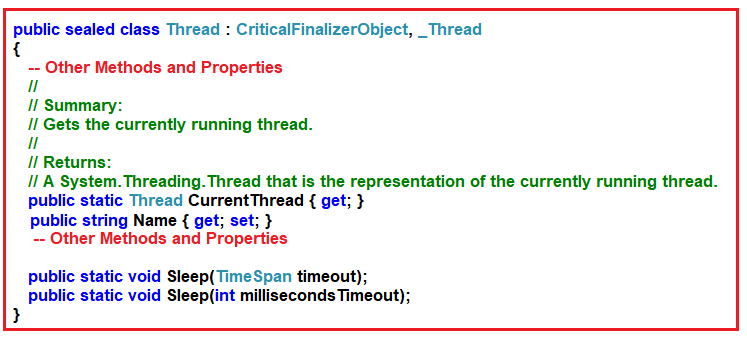
**}**

**}**

This is a very simple program. But internally there is a thread that is going to execute this code and that thread is called Main Thread. Now let us prove this.

**Understanding the Thread class in C#:**

The Thread class contains one static property i.e. **CurrentThread** which is going to return the instance of the currently executing thread. If you go to the definition of **Thread** class then you will find the following signature.



As you can see the **CurrentThread** static property return type is Thread i.e. it is going to return the instance of the currently executing thread. Along the same line, there is a non-static property called Name using which we can set and get the Name of the currently executing thread.

**Note:** By default, the thread does not have any name. If you want then you can provide any name to the thread by using the Name property of the Thread class. So, modify the program as shown below.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Thread t = Thread.CurrentThread;

//By Default the Thread does not have any name

//if you want then you can provide the name explicitly

t.Name = "Main Thread";

Console.WriteLine**(**"Current Executing Thread Name :" + t.Name**)**;

Console.WriteLine**(**"Current Executing Thread Name :" + Thread.CurrentThread.Name**)**;

Console.Read**()**;

**}**

**}**

**}**

**Output:**

Printing the Single Thread Name in C#

As you can see in order to run the application code one thread is created and i.e. the Main Thread. So, this proves that, by default, every application is a single-threaded application.

**What are the drawbacks of Single-Threaded Applications?**

In a single thread application, all the logic or code present in the program will be executed by a single thread only i.e. the Main thread. For example, if we have three methods in our application and if all these three methods are going to be called from the Main method. Then the main thread is responsible to execute all these three methods sequentially i.e. one by one. It will execute the first method and once it completes the execution of the first method then only it executes the second method and so on.

Let us understand this with an example. Modify the program as shown below.

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Method1**()**;

Method2**()**;

Method3**()**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

**}**

**}**

**static** **void** Method2**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method2 :" + i**)**;

**}**

**}**

**static** **void** Method3**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method3 :" + i**)**;

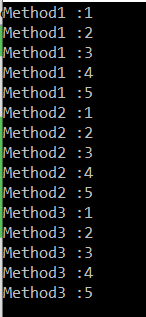
**}**

**}**

**}**

**}**

**Output:**



As you can see in the above output, the methods are called and execute one after the other. The Main thread first executes Method1 and once it completes the execution of Method1 then it calls Method2 and then Method3.

**What is the problem with the above program execution?**

In our example, we are just writing some simple code to print the values from 1 to 5. What will you do if one method is taking more than the expected time? Suppose Method2 is going to interact with a database or it is going to invoke any web service which will take more than 10 seconds to provide the response. In that case, the Method2 execution will be delayed for 10 seconds as it is waiting there to get a response back either from the database or from the Web Service. Until Method2 is not completed its execution, Method3 is not going to be executed because of the sequential execution of the program i.e. one by one.

**Let us understand this with an example.**

Note: Here we are not going to perform any database or Web Service call instead we can use the Thread class Sleep method to delay the execution of Method2 for 10 seconds. Following is the signature of Sleep Method:

**public static void Sleep(int millisecondsTimeout);**

The sleep method takes the time in milliseconds as input and then suspends the current thread execution for that specified number of milliseconds. So, please modify the Program as shown below.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Method1**()**;

Method2**()**;

Method3**()**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

**}**

**}**

**static** **void** Method2**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method2 :" + i**)**;

**if** **(**i == 3**)**

**{**

Console.WriteLine**(**"Performing the Database Operation Started"**)**;

//Sleep for 10 seconds

Thread.Sleep**(**10000**)**;

Console.WriteLine**(**"Performing the Database Operation Completed"**)**;

**}**

**}**

**}**

**static** **void** Method3**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method3 :" + i**)**;

**}**

**}**

**}**

**}**

Now run the application and notice that the Method2 execution is delayed for 10 seconds. Once Method2 completes its execution then only Method3 start its execution. This is because all these three methods are executed by a single thread and this is the drawback of the single-threaded application.

**How to solve the above problem?**

To solve the above problem, we are provided with a concept called **Multithreading in C#**. As we already discussed Operating System has Processes which is used to run our applications. The Process contains Thread which will actually run our application code.

A process can have multiple threads and each thread can perform a different task. In simple words, we can say that the three methods we define in our program can be executed by three different threads. The advantage is that the execution takes place simultaneously. So when multiple threads trying to execute the application code, then the operating system allocates some time period to each thread to execute.

Now, in our example, we want to execute the three methods using three different threads let say t1, t2, and t3. The thread t1 is going to execute Method1, thread t2 is going to execute the Method2. At the same time, the Method3 is going to be executed by thread t3. Let us modify the Program as shown below to execute the methods with different Threads.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

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

//Creating Threads

Thread t1 = new Thread**(**Method1**)**

**{**

Name = "Thread1"

**}**;

Thread t2 = new Thread**(**Method2**)**

**{**

Name = "Thread2"

**}**;

Thread t3 = new Thread**(**Method3**)**

**{**

Name = "Thread3"

**}**;

//Executing the methods

t1.Start**()**;

t2.Start**()**;

t3.Start**()**;

Console.WriteLine**(**"Main Thread Ended"**)**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

Console.WriteLine**(**"Method1 Started using " + Thread.CurrentThread.Name**)**;

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

**}**

Console.WriteLine**(**"Method1 Ended using " + Thread.CurrentThread.Name**)**;

**}**

**static** **void** Method2**()**

**{**

Console.WriteLine**(**"Method2 Started using " + Thread.CurrentThread.Name**)**;

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method2 :" + i**)**;

**if** **(**i == 3**)**

**{**

Console.WriteLine**(**"Performing the Database Operation Started"**)**;

//Sleep for 10 seconds

Thread.Sleep**(**10000**)**;

Console.WriteLine**(**"Performing the Database Operation Completed"**)**;

**}**

**}**

Console.WriteLine**(**"Method2 Ended using " + Thread.CurrentThread.Name**)**;

**}**

**static** **void** Method3**()**

**{**

Console.WriteLine**(**"Method3 Started using " + Thread.CurrentThread.Name**)**;

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method3 :" + i**)**;

**}**

Console.WriteLine**(**"Method3 Ended using " + Thread.CurrentThread.Name**)**;

**}**

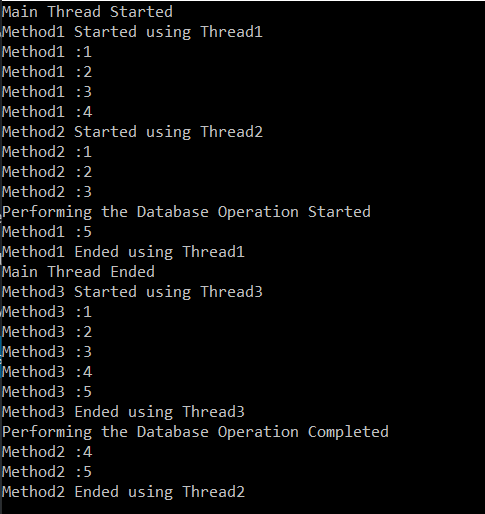
**}**

**}**

**Code Explanation:**

As you can see in the above code, we have created three different instances of Thread class. To the constructor of Thread class, we need to pass the method name which needs to be executed by that Thread. Then we call the **Start()** method on the Thread class which will start executing the method. Here the Main thread is going to create all other Threads.

**Note:** You will not get the output in a sequential manner. Run the application and see the output as shown below. The output may vary in your machine.



**Note:** The main advantage of using Multithreading is the maximum utilization of CPU resources.

Here, in this article, I try to explain the concept of Multithreading in C# with Examples. In the next article, I am going to discuss the [**Constructors of the Thread class**](https://dotnettutorials.net/lesson/constructors-of-thread-class-csharp/). I hope you understood the basics for C# Multithreading with Examples and enjoy this article.

**Constructors of Thread class in C#**

**Constructors of Thread class in C#**

In this article, I am going to discuss the **Constructors of Thread class in C#** with some examples. Please read our previous article before proceeding to this article where we discussed the basics of [**Multithreading in C#**](https://dotnettutorials.net/lesson/multithreading-in-csharp/) with examples. As part of this article, we are going to discuss the following pointers in detail with examples.

1. **Understanding the Constructors of Thread Class in C#.**
2. **Why the constructor of Thread class taking a parameter of Delegate type?**
3. **Understanding ThreadStart delegate in C#.**
4. **Thread Function with Parameter in C#.**
5. **Understanding ParameterizedThreadStart Delegate in C#.**
6. **When to use ParameterizedThreadStart over ThreadStart delegate?**
7. **What are the Problems with the ParameterizedThreadStart delegate in C#?**
8. **How to Overcome the Problems of ParameterizedThreadStart delegate in C#?**

**Understanding the Constructors of Thread Class in C#.**

Let us understand this with an example. Please have a look at the following example.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Thread t1 = new Thread**(**DisplayNumbers**)**;

t1.Start**()**;

Console.Read**()**;

**}**

**static** **void** DisplayNumbers**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

**}**

**}**

**}**

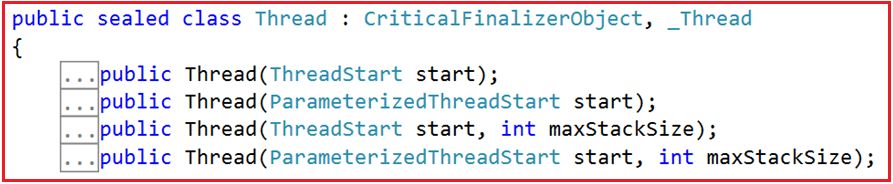
**}**

As you can see in the above example, we created an instance of the Thread class and to the constructor of the Thread class, we passed the method name that we want the thread to execute as shown below.

**Thread t1 = new Thread(DisplayNumbers);**

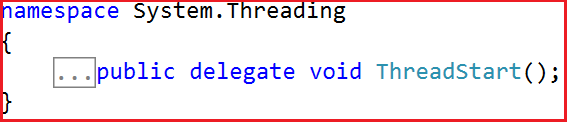
**Constructors of Thread Class in C#:**

In C#, the Thread class contains four constructors. If you go to the definition of Thread class then you can see the Constructors as shown below.



Now you may have one question, the Thread class constructor which takes one parameter is either of the type **ThreadStart** or **ParameterizedThreadStart**, but in our example, we are passing the method name as a parameter to the Thread class Constructor and it works, how?

To understand this, let’s go to the definition of **ThreadStart** and you can see **ThreadStart** is a delegate as shown below.



**Why the constructor of Thread class taking a parameter of Delegate type?**

As we already discussed, the main objective of creating a Thread in C# is to execute a function. A delegate is a type-safe function pointer. It means the delegate points to a function that the thread has to execute. In simple words, we can say that all the threads that we create require an entry point (i.e. a pointer to the function) from where it should execute. This is the reason why threads always require a delegate. If you want to learn Delegates in C# with examples, then I strongly recommended you to read the following article where we discussed Delegates in detail.

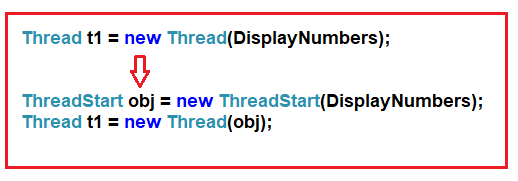
[**Delegates in C# with Examples**](https://dotnettutorials.net/lesson/delegates-csharp/)

**Note:** The signature of the delegate should be the same as the signature of the method it points to.

The **ThreadStart** delegate does not take any parameter as well as the return type is void. In our example, the **DisplayNumbers()** function signature is the same as the **ThreadStart** delegate signature as this function return type is void as well as it does not take any parameters.

**Thread t1 = new Thread(DisplayNumbers);**

The above statement implicitly converted to **ThreadStart** delegate instance. So, you can write the above statement as shown below.



It’s a two steps process. First, we need to create the **ThreadStart delegate instance** and while creating the instance to its constructor we need to **pass the method name** which we want to execute. In the second step to the**Constructor of Thread** class, we need to pass the **ThreadStart instance** as a parameter.

**The Complete Example is given below:**

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating the ThreadStart Delegate instance by passing the

//method name as a parameter to its constructor

ThreadStart obj = new ThreadStart**(**DisplayNumbers**)**;

//Passing the ThreadStart Delegate instance as a parameter

//its constructor

Thread t1 = new Thread**(**obj**)**;

t1.Start**()**;

Console.Read**()**;

**}**

**static** **void** DisplayNumbers**()**

**{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

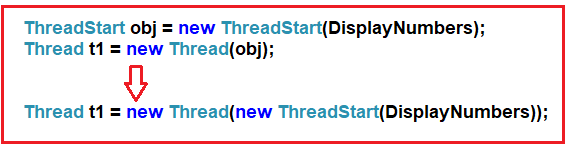
**}**

**}**

**}**

**}**

You can also combine the above two statements into a single statement as shown below.



It is also possible to create a Thread class instance using the delegate keyword as shown below.  
**Thread t1 = new Thread(delegate() { DisplayNumbers(); });**

We can also rewrite the same line using an anonymous method as shown below.  
**Thread t1 = new Thread(() => { DisplayNumbers(); });**  
**Thread t1 = new Thread(() => DisplayNumbers());**

You can also directly write the logic as part of the anonymous method instead of calling the DisplayNumbers method as shown below.

Thread t1 = new Thread**(()** =**>** **{**

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

**}**

**})**;

**Thread Function with Parameter in C#:**

Let change the **DisplayNumbers()** method implementation. Now, this method takes one input parameter of the object type. And then convert that object type to an integer value and then print the numbers up to that value starting from 1.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

Thread t1 = new Thread**(**obj.DisplayNumbers**)**;

t1.Start**(**5**)**;

Console.Read**()**;

**}**

**public** **void** DisplayNumbers**(object** Max**)**

**{**

**int** Number = Convert.ToInt32**(**Max**)**;

**for** **(int** i = 1; i **<**= Number; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

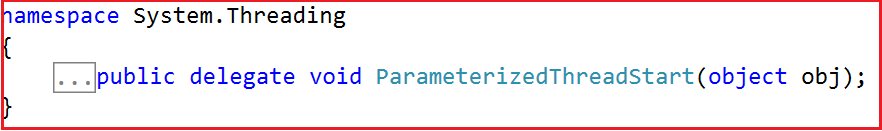
**}**

**}**

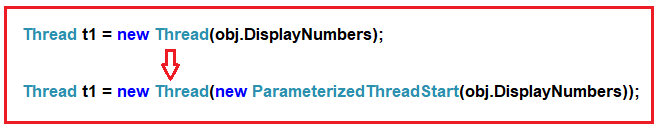
**}**

**}**

When the method taking one parameter, then the Thread class uses the **ParameterizedThreadStart** delegate. The definition of **ParameterizedThreadStart** is shown below.



As you can see the **ParameterizedThreadStart** delegate is taking one parameter of **object** type and like **ThreadStart** it also does not return any value. Now the **DisplayNumbers()** method signature is the same as the signature of this **ParameterizedThreadStart** delegate. So the Framework internally converts the statements as shown below.



**Creating the ParameterizedThreadStart instance manually:**

Let us see how to create the **ParameterizedThreadStart** delegate instance manually and passing that instance to the Thread class Constructor. So, modify the program as shown below.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

ParameterizedThreadStart PTSD = new ParameterizedThreadStart**(**obj.DisplayNumbers**)**;

Thread t1 = new Thread**(**PTSD**)**;

t1.Start**(**5**)**;

Console.Read**()**;

**}**

**public** **void** DisplayNumbers**(object** Max**)**

**{**

**int** Number = Convert.ToInt32**(**Max**)**;

**for** **(int** i = 1; i **<**= Number; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

**}**

**}**

**}**

**}**

Now run the application and it should display the output as expected.

**When to use ParameterizedThreadStart over ThreadStart delegate in C#?**

You need to use the **ParameterizedThreadStart** delegate if your method taking any values else you just need to use the ThreadStart delegate which does not take any parameter.

**What are the Problems with the ParameterizedThreadStart delegate in C#?**

As you can see, the parameter type of **ParameterizedThreadStart** delegate is **object** type. So, the parameter of the thread function is also going to be the object data type. And you cannot change the data type from object to any other type and if you try then it will give you a compile-time error. As the thread function operates on object data type now we can pass any type of values and it accepts. As a result, the function is not going to be type-safe as we can pass any type of values.

**Let us try to pass a string value and see what happens as shown below.**

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

ParameterizedThreadStart PTSD = new ParameterizedThreadStart**(**obj.DisplayNumbers**)**;

Thread t1 = new Thread**(**PTSD**)**;

t1.Start**(**"Hi"**)**;

Console.Read**()**;

**}**

**public** **void** DisplayNumbers**(object** Max**)**

**{**

**int** Number = Convert.ToInt32**(**Max**)**;

**for** **(int** i = 1; i **<**= Number; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

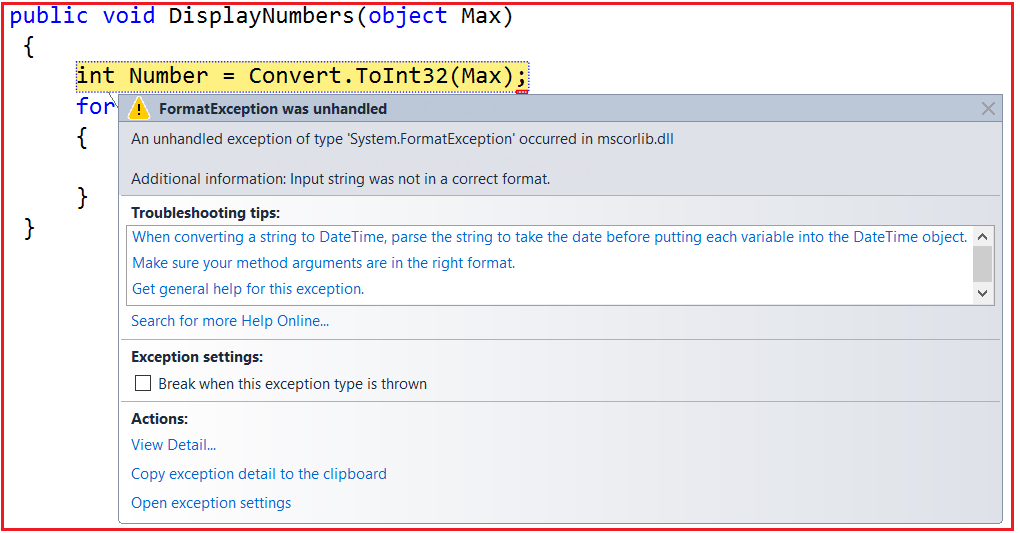
**}**

**}**

**}**

**}**

Now you will not get any compile-time error, but once you run the application, then you will get runtime error as shown below.



**How to Pass data to the Thread Function in a Type Safe Manner**

**How to Pass data to the Thread Function in a Type-Safe Manner**

In this article, I am going to discuss **how to pass data to the thread function in a type-safe manner**. Please read our previous article before proceeding to this article where we discussed the [**Thread class constructors in C#**](https://dotnettutorials.net/lesson/constructors-of-thread-class-csharp/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **How to Pass data to the Thread function in C#?**
2. **How to make the thread function type-safe in C#?**

**We have written the following program in our previous article.**

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Program obj = new Program**()**;

ParameterizedThreadStart PTSD = new ParameterizedThreadStart**(**obj.DisplayNumbers**)**;

Thread t1 = new Thread**(**PTSD**)**;

t1.Start**(**"Hi"**)**;

Console.Read**()**;

**}**

**public** **void** DisplayNumbers**(object** Max**)**

**{**

**int** Number = Convert.ToInt32**(**Max**)**;

**for** **(int** i = 1; i **<**= Number; i++**)**

**{**

Console.WriteLine**(**"Method1 :" + i**)**;

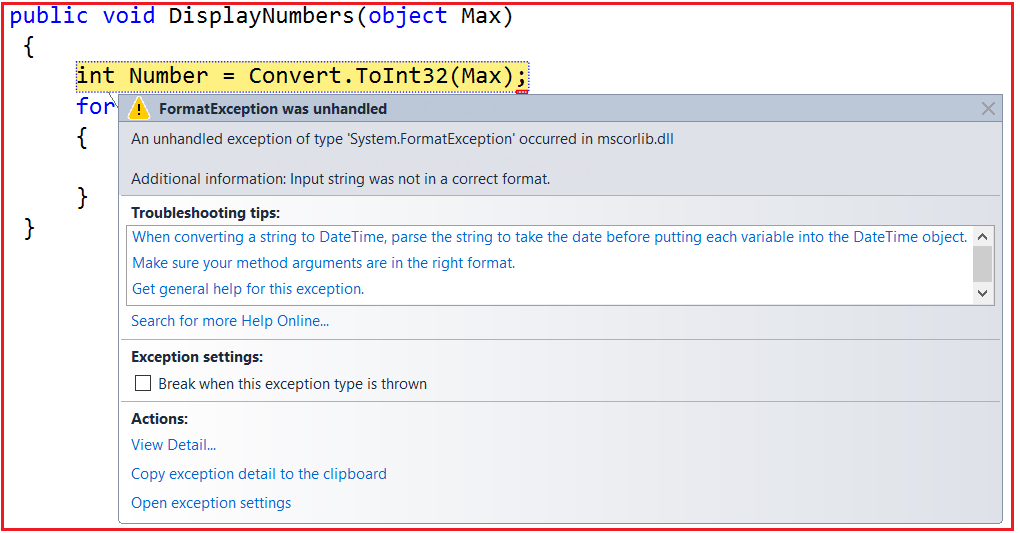
**}**

**}**

**}**

**}**

At the time of compilation, we will not get any compile-time error. But when we run the application, we will get the following runtime error.



This is because the thread function is not type-safe as it operates on the object data type. Let see how to make the thread function type so that we can pass the data in a type-safe manner.

**How to make the thread function type-safe in C#**

When we are saying type-safe means we should not have to use the object data type. Here in our example, we need to use the data type as an integer. So at the time compilation, we pass any data other than integer then it should givens you a compile-time error. Let us see how to achieve this step by step.

**Step1:**

In order to pass the data in a type-safe manner to the Thread function, first, you need to encapsulate the thread function and the data it requires in a helper class. So, create a class file with the **NumberHelper.cs** and then copy and paste the following code in it.

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**public** **class** NumberHelper

**{**

**int** \_Number;

**public** NumberHelper**(int** Number**)**

**{**

\_Number = Number;

**}**

**public** **void** DisplayNumbers**()**

**{**

**for** **(int** i = 1; i **<**= \_Number; i++**)**

**{**

Console.WriteLine**(**"value : " + i**)**;

**}**

**}**

**}**

**}**

As you can see we created the above Number Helper class with one private variable, one parameterized constructor and one method. The private variable \_**Number** is going to hold the target number. The constructor takes one input parameter of integer type and then it assigns that value to the private variable. So, while we are creating the instance of **NumberHelper** class we need to supply the Number value. Finally, the **DisplayNumbers** function is used to display the values starting from 1 to the value that is present in the **\_Number** variable.

**Step2:**

In the main method create an instance of **NumberHelper** class and to its constructor pass the integer value. Then create the **ThreadStart** delegate instance and pass the Display Number function. So, please modify the Main method as shown below.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** Max = 10;

NumberHelper obj = new NumberHelper**(**Max**)**;

Thread T1 = new Thread**(**new ThreadStart**(**obj.DisplayNumbers**))**;

T1.Start**()**;

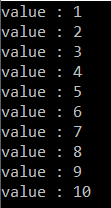
Console.Read**()**;

**}**

**}**

**}**

Now run the application and it should display the output as expected as shown below.



**How to retrieve data from a thread function**

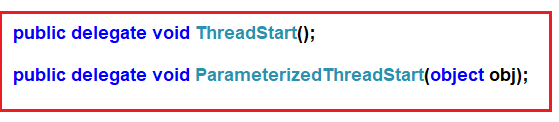
**How to retrieve data from a thread function using a callback method**

In this article, I am going to discuss **how to retrieve data from a thread function using a callback method** with an example. Please read our previous article before proceeding to this article where we discussed [**how to pass the data to a thread function in a type-safe**](https://dotnettutorials.net/lesson/how-to-pass-data-to-the-thread-function-in-a-type-safe-manner/) manner. As part of this article, we will discuss the following pointers.

1. **What is a Callback Method in C#?**
2. **How does a callback method work in C#?**
3. **How to retrieve data from a thread function using a callback method in C#?**

**How to retrieve the data from a thread function in C#?**

As of now, we have discussed the uses of **ThreadStart** and **ParameterizedThreadStart** delegates. If you notice that the return type of these two delegates is void as shown in the below image.



So, using the above two delegates we cannot return any data from a method as the return type is void. Then the question that comes to our mind is how to retrieve the data from a thread function?

The answer is by using a callback method.

**How to retrieve data from a thread Function using the callback method:**

Let us see an example with step by step procedure to show how we can use a callback method to retrieve the data from a thread function.

**Step1:**

In order to retrieve the data from a thread function, first, you need to encapsulate the thread function and the data it requires in a helper class. To the constructor of the Helper class, you need to pass the required data as well as a delegate representing the callback method.

From the thread function body, you need to invoke the callback delegate just before the thread method ends. And one more thing you need to take care that the callback delegate and the actual callback method signature should be the same.

So, create a class file with the **NumberHelper.cs** and then copy and paste the following code in it. The code is explained through comments, so please go through the comment lines.

**using** *System;*

**namespace** *ThreadingDemo*

**{**

// First Create the callback delegate with the same signature of the callback method.

**public** **delegate** **void** ResultCallbackDelegate**(int** Results**)**;

//Creating the Helper class

**public** **class** NumberHelper

**{**

//Creating two private variables to hold the Number and ResultCallback instance

**private** **int** \_Number;

**private** ResultCallbackDelegate \_resultCallbackDelegate;

//Initializing the private variables through constructor

//So while creating the instance you need to pass the value for Number and callback delegate

**public** NumberHelper**(int** Number, ResultCallbackDelegate resultCallbackDelagate**)**

**{**

\_Number = Number;

\_resultCallbackDelegate = resultCallbackDelagate;

**}**

//This is the Thread function which will calculate the sum of the numbers

**public** **void** CalculateSum**()**

**{**

**int** Result = 0;

**for** **(int** i = 1; i **<**= \_Number; i++**)**

**{**

Result = Result + i;

**}**

//Before the end of the thread function call the callback method

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

**{**

\_resultCallbackDelegate**(**Result**)**;

**}**

**}**

**}**

**}**

**Step2:**

Here, in the second step, first, we need to create the callback method whose signature should be the same as the signature of the CallBack Delegate. In our example, **ResultCallBackMethod** is the callback method and its signature is the same as the signature of the **ResultCallbackDelegate** delegate. Within the Main method, we need to create an instance of the ResultCallbackDelegate delegate and while creating the instance we need to pass the **ResultCallBackMethod** as the parameter to its constructor. So when we invoke the delegate it will call the **ResultCallBackMethod**method.

Please modify the Program class code as shown below. The example code is self-explained. So, please go through the comment lines for better understanding.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Create the ResultCallbackDelegate instance and to its constructor

//pass the callback method name

ResultCallbackDelegate resultCallbackDelegate = new ResultCallbackDelegate**(**ResultCallBackMethod**)**;

**int** Number = 10;

//Creating the instance of NumberHelper class by passing the Number

//the callback delegate instance

NumberHelper obj = new NumberHelper**(**Number, resultCallbackDelegate**)**;

//Creating the Thread using ThreadStart delegate

Thread T1 = new Thread**(**new ThreadStart**(**obj.CalculateSum**))**;

T1.Start**()**;

Console.Read**()**;

**}**

//Callback method and the signature should be the same as the callback delegate signature

**public** **static** **void** ResultCallBackMethod**(int** Result**)**

**{**

Console.WriteLine**(**"The Result is " + Result**)**;

**}**

**}**

**}**

Now run the application and you should see the output as expected.

**What is a Callback Method in C#?**

We can define a callback function as a function pointer that is being passed as an argument to another function. And then it is expected to call back that function at some point in time.

In our example, we call the thread function of NumberHelper class from the Main method of Program class. While creating the instance of **NumberHelper** class we pass the callback function as an argument to that class constructor. And then we expected that callback method to be called at some point in time.

**IsAlive and Join Method of Thread class in C#**

**IsAlive and Join Method of Thread class in C# with Examples**

In this article, I am going to discuss the **IsAlive and** **Join Method of Thread class in C#** with examples. Please read our previous article before proceeding to this article where we discussed [**how to return data from a thread function using the call back method**](https://dotnettutorials.net/lesson/how-to-retrieve-data-from-a-thread-function/) in C# with an example. As part of this article, we are going to discuss the following pointers.

1. **Understanding the significance of the Join Method of Thread class.**
2. **Examples using different overloaded versions of the Join Method.**
3. **Understanding the use of the IsAlive method of Thread class.**

**Understanding the significance of the Join Method of Thread class.**

Let us understand the use of the Join Method of Thread class with an example. Please have a look at the following example. In the below example we have created three methods and then execute the methods using three separate threads. The point that you need to remember is the threads thread1, thread2, and thread3 are called as the child threads of Main thread. This is because these three threads are created by the main thread only.

**Example:**

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

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

//Main Thread creating three child threads

Thread thread1 = new Thread**(**Method1**)**;

Thread thread2 = new Thread**(**Method2**)**;

Thread thread3 = new Thread**(**Method3**)**;

thread1.Start**()**;

thread2.Start**()**;

thread3.Start**()**;

Console.WriteLine**(**"Main Thread Ended"**)**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

Console.WriteLine**(**"Method1 - Thread1 Started"**)**;

Thread.Sleep**(**3000**)**;

Console.WriteLine**(**"Method1 - Thread 1 Ended"**)**;

**}**

**static** **void** Method2**()**

**{**

Console.WriteLine**(**"Method2 - Thread2 Started"**)**;

Thread.Sleep**(**2000**)**;

Console.WriteLine**(**"Method2 - Thread2 Ended"**)**;

**}**

**static** **void** Method3**()**

**{**

Console.WriteLine**(**"Method3 - Thread3 Started"**)**;

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

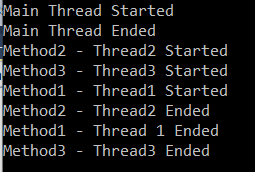
Console.WriteLine**(**"Method3 - Thread3 Ended"**)**;

**}**

**}**

**}**

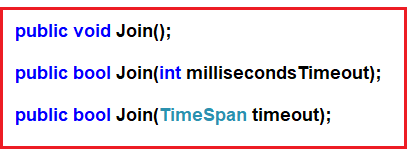
**Output:** The output may vary when you run the application.



As you can see from the above output, the Main thread is not waiting for all the child threads to complete their execution or task. If you want that the Main thread should not be existed until and unless all the child thread completes their task then you need to use the Join method which is available in Thread class.

**Join Method of Thread class in C#:**

The Join method of Thread class in C# blocks the current thread and makes it wait until the child thread on which the Join method invoked completes its execution. There are three overloaded versions available for the Join Method in Thread class as shown below.



The first version of the Join method which does not take any parameter will block the calling thread (i.e. the Parent thread) until the thread (child thread) completes its execution. In this case, the calling thread is going to wait for indefinitely time until the thread on which the Join Method invoked is completed.

The second version of the Join Method allows us to specify the time out. It means it will block the calling thread until the child thread terminates or the specified time elapses. This overloaded takes the time in milliseconds. This method returns true if the thread has terminated and returns false if the thread has not terminated after the amount of time specified by the millisecondsTimeout parameter has elapsed.

The third overloaded version of this method is the same as the second overloaded version. The only difference is that here we need to use the TimeSpan to set the amount of time to wait for the thread to terminate.

**Example: Using the Join Method of Thread class in C#**

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

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

//Main Thread creating three child threads

Thread thread1 = new Thread**(**Method1**)**;

Thread thread2 = new Thread**(**Method2**)**;

Thread thread3 = new Thread**(**Method3**)**;

thread1.Start**()**;

thread2.Start**()**;

thread3.Start**()**;

thread1.Join**()**;

thread2.Join**()**;

thread3.Join**()**;

Console.WriteLine**(**"Main Thread Ended"**)**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

Console.WriteLine**(**"Method1 - Thread1 Started"**)**;

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

Console.WriteLine**(**"Method1 - Thread 1 Ended"**)**;

**}**

**static** **void** Method2**()**

**{**

Console.WriteLine**(**"Method2 - Thread2 Started"**)**;

Thread.Sleep**(**2000**)**;

Console.WriteLine**(**"Method2 - Thread2 Ended"**)**;

**}**

**static** **void** Method3**()**

**{**

Console.WriteLine**(**"Method3 - Thread3 Started"**)**;

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

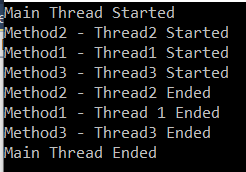
Console.WriteLine**(**"Method3 - Thread3 Ended"**)**;

**}**

**}**

**}**

**Output:**



Now, for example, if you don’t want the main thread to wait until the thread3 completes its execution. Then you just need to call the Join method on thread1 and thread2.

**Second Overloaded version of Join Method:**

You need to use the second overloaded version when you want the main thread to wait for a specified amount of time. For example, you want the main thread to wait for 3 seconds for the thread3 to complete its task. Then you need to the Join method as shown below.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

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

//Main Thread creating three child threads

Thread thread1 = new Thread**(**Method1**)**;

Thread thread2 = new Thread**(**Method2**)**;

Thread thread3 = new Thread**(**Method3**)**;

thread1.Start**()**;

thread2.Start**()**;

thread3.Start**()**;

**if(**thread3.Join**(**3000**))**

**{**

Console.WriteLine**(**"Thread 3 Execution Completed in 3 second"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Thread 3 Execution Not Completed in 3 second"**)**;

**}**

Console.WriteLine**(**"Main Thread Ended"**)**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

Console.WriteLine**(**"Method1 - Thread1 Started"**)**;

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

Console.WriteLine**(**"Method1 - Thread 1 Ended"**)**;

**}**

**static** **void** Method2**()**

**{**

Console.WriteLine**(**"Method2 - Thread2 Started"**)**;

Thread.Sleep**(**2000**)**;

Console.WriteLine**(**"Method2 - Thread2 Ended"**)**;

**}**

**static** **void** Method3**()**

**{**

Console.WriteLine**(**"Method3 - Thread3 Started"**)**;

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

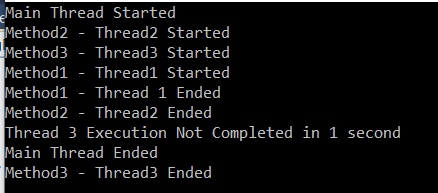
Console.WriteLine**(**"Method3 - Thread3 Ended"**)**;

**}**

**}**

**}**

**Output:**



**IsAlive Method of Thread Class:**

The IsAlive method of Thread class returns true if the thread is still executing else returns false. Let us understand this with an example.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

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

Thread thread1 = new Thread**(**Method1**)**;

thread1.Start**()**;

**if** **(**thread1.IsAlive**)**

**{**

Console.WriteLine**(**"Thread1 Method1 is still doing its work"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Thread1 Method1 Completed its work"**)**;

**}**

thread1.Join**()**;

Console.WriteLine**(**"Main Thread Ended"**)**;

Console.Read**()**;

**}**

**static** **void** Method1**()**

**{**

Console.WriteLine**(**"Method1 - Thread1 Started"**)**;

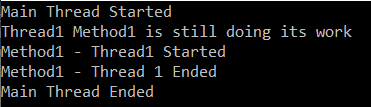
Console.WriteLine**(**"Method1 - Thread 1 Ended"**)**;

**}**

**}**

**}**

**Output:**



**Protecting Shared Resource in Multithreading Using Locking**

**Protecting Shared Resource in Multithreading Using Locking**

In this article, I am going to discuss **How to Protect the Shared Resources in Multithreading using Locking from Concurrent Access** with an example. Please read our previous article where we discussed the significance of the [**Alive and Join Method of Thread Class**](https://dotnettutorials.net/lesson/join-method-of-thread-class/) using some examples.

1. **What happened when accessing shared resources in a single-thread application?**
2. **What happened when accessing shared resources in a multi-thread application?**
3. **Understanding the Locking Mechanism.**
4. **How to protect the shared resources in a multithread environment from concurrent access?**

In a multithreading application, it is very important for us to handle multiple threads for executing critical section code. For example, if we have a shared resource and if multiple threads want to access the shared resource then we need to protect the shared resource from concurrent access otherwise we will get some inconsistency output. In C#, we can use lock and Monitor to provide thread safety in a multithreaded application. Both lock and monitor provides a mechanism which ensures that only one thread is executing the critical section code at any given point of time to avoid any functional breaking of code. In this article, I am going to discuss how to protect the shared resource using the lock and in the next article, I am going to discuss how to do the same thing using the monitor.

**Accessing shared resource in a single-threaded environment:**

Before understanding lock to protect the resource in a multithread environment, let us first understand the problem if we will not protect the shared resource. In the below example, we have a shared resource i.e. **DisplayMessage()** method and we call that method three times from the Main method as shown below.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

DisplayMessage**()**;

DisplayMessage**()**;

DisplayMessage**()**;

Console.Read**()**;

**}**

**static** **void** DisplayMessage**()**

**{**

Console.Write**(**"[Welcome to the "**)**;

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

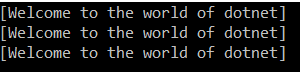
Console.WriteLine**(**"world of dotnet!]"**)**;

**}**

**}**

**}**

**Output:**



As the above program is a single-threaded program, so we got the output as expected. Let us see what happens if we access the shared resources in a multithreaded environment.

**Accessing shared resource in a multithreaded environment:**

In the following example, we created three different threads and then invoke the same DisplayMessage() method.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Thread t1 = new Thread**(**DisplayMessage**)**;

Thread t2 = new Thread**(**DisplayMessage**)**;

Thread t3 = new Thread**(**DisplayMessage**)**;

t1.Start**()**;

t2.Start**()**;

t3.Start**()**;

Console.Read**()**;

**}**

**static** **void** DisplayMessage**()**

**{**

Console.Write**(**"[Welcome to the "**)**;

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

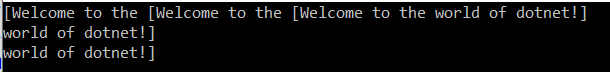
Console.WriteLine**(**"world of dotnet!]"**)**;

**}**

**}**

**}**

**Output:**



As you can see we are not getting the output as expected. So, the point that you need to keep in mind is, if the shared resources are not protected in the multithreaded environment from concurrent access then the output or the behavior of the application becomes inconsistent that’s what we see in our previous example.

**How to protect the shared resources in a multithread environment from concurrent access in C#?**

We can protect the shared resources in a multithread environment from concurrent access by using the concept Monitor and locking. Let us see how to protect the shared resource using locking and see the output.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Thread t1 = new Thread**(**DisplayMessage**)**;

Thread t2 = new Thread**(**DisplayMessage**)**;

Thread t3 = new Thread**(**DisplayMessage**)**;

t1.Start**()**;

t2.Start**()**;

t3.Start**()**;

Console.Read**()**;

**}**

**private** **static** **object** \_lockObject = new **object()**;

**static** **void** DisplayMessage**()**

**{**

**lock(**\_lockObject**)**

**{**

Console.Write**(**"[Welcome to the "**)**;

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

Console.WriteLine**(**"world of dotnet!]"**)**;

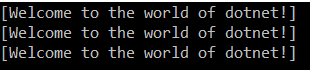
**}**

**}**

**}**

**}**

Now run the application and see the output as expected as shown below.



**Note:** The section or block or particular resource that you want to protect should be placed inside the lock block.

Let us understand this with an example. In the following example, we are only protecting the shared Count variable from concurrent access.

**using** *System.Threading;*

**using** *System;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **int** Count = 0;

**static** **void** Main**(**string**[]** args**)**

**{**

Thread t1 = new Thread**(**IncrementCount**)**;

Thread t2 = new Thread**(**IncrementCount**)**;

Thread t3 = new Thread**(**IncrementCount**)**;

t1.Start**()**;

t2.Start**()**;

t3.Start**()**;

//Wait for all three threads to complete their execution

t1.Join**()**;

t2.Join**()**;

t3.Join**()**;

Console.WriteLine**(**Count**)**;

Console.Read**()**;

**}**

**private** **static** **object** \_lockObject = new **object()**;

**static** **void** IncrementCount**()**

**{**

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

**{**

//Only protecting the shared Count variable

**lock** **(**\_lockObject**)**

**{**

Count++;

**}**

**}**

**}**

**}**

**}**

When you run the above program, it will give you the output as expected as 3000000.

# Protecting Shared Resource in Multithreading Using Monitor

## ****Protecting Shared Resource in Multithreading Using Monitor****

In this article, I am going to discuss **How to Protect the Shared Resources in Multithreading using Monitor from Concurrent Access** with some examples. Please read our previous article before proceeding to this article where we discussed why we need to protect the shared resource as well as [**how to protect the shared resource using the lock**](https://dotnettutorials.net/lesson/locking-in-multithreading/) with some examples. As part of this article, we are going to discuss the following pointers.

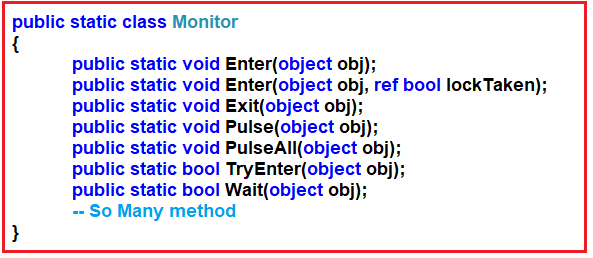
1. **Understanding the Monitor class in C#.**
2. **How to Protect the Shared Resources in Multithreading using Monitor class in C#?**
3. **Understanding Different Methods of Monitor class with examples.**
4. **Difference between Monitor and lock in C#.**

As we already discussed both Monitor and lock are used to provide thread safety to a shared resource in a multithreaded application in C#. So, let us understand the Monitor class and its methods in detail as well as how to protect the shared resource using the monitor class.

##### ****Understanding the Monitor Class in C#:****

The Monitor class in C# provides a mechanism that synchronizes access to objects. Let us simplify the above definition. In simple words, we can say that, like the lock, we can also use this class to protect shared resources in a multi-threaded environment. This can be done by acquiring an exclusive lock on the object so that only one thread can enter into the critical section at any given point of time.

The Monitor is a static class and belongs to the System.Threading namespace. As a static class, it provides a collection of static methods as shown in the below image. Using these static methods you can provide access to the monitor associated with a particular object.



Let us understand the methods of the Monitor class.

###### **Enter, TryEnter:**

These two methods are used to acquire an exclusive lock for an object. This action marks the beginning of a critical section. No other thread can enter into the critical section unless it is executing the instructions in the critical section using a different locked object.

###### **Wait:**

The Wait method is used to release the lock on an object and permit other threads to lock and access the object by blocking the current thread until it reacquires the lock. The calling thread waits while another thread accesses the object. Pulse signals are used to notify waiting threads about changes to an object’s state.

###### **Pulse (signal), PulseAll:**

The above two methods are used to send a signal to one or more waiting threads. The signal notifies a waiting thread that the state of the locked object has changed, and the owner of the lock is ready to release the lock.

###### **Exit():**

The Exit method is used to release the exclusive lock from the specified object. This action marks the end of a critical section protected by the locked object.

##### ****Example:****

**Let us see an example to understand how to use Monitor in C#.**

**using** *System;*

**using** *System.Threading;*

**namespace** *MonitorDemo*

**{**

**class** Program

**{**

**static** **readonly** **object** lockObject = new **object()**;

**public** **static** **void** PrintNumbers**()**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name + " Trying to enter into the critical section"**)**;

Monitor.Enter**(**lockObject**)**;

**try**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name + " Entered into the critical section"**)**;

**for** **(int** i = 0; i **<** 5; i++**)**

**{**

Thread.Sleep**(**100**)**;

Console.Write**(**i + ","**)**;

**}**

Console.WriteLine**()**;

**}**

**finally**

**{**

Monitor.Exit**(**lockObject**)**;

Console.WriteLine**(**Thread.CurrentThread.Name + " Exit from critical section"**)**;

**}**

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Thread**[]** Threads = new Thread**[**3**]**;

**for** **(int** i = 0; i **<** 3; i++**)**

**{**

Threads**[**i**]** = new Thread**(**PrintNumbers**)**;

Threads**[**i**]**.Name = "Child Thread " + i;

**}**

**foreach** **(**Thread t in Threads**)**

**{**

t.Start**()**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**

##### How to Protect the Shared Resources in Multithreading using Monitor from Concurrent Access

##### ****Monitor.EnterlockObject, ref IslockTaken):****

Let us understand the overloaded version of the Enter method. The **Monitor.Enter(lockObject, ref IslockTaken)** acquires an exclusive lock on the specified object. It then automatically sets a value that indicates whether the lock was taken or not. The second parameter which is a Boolean parameter returns true if the lock is acquired else it returns false.

**Example:**

**using** *System;*

**using** *System.Threading;*

**namespace** *MonitorDemo*

**{**

**class** Program

**{**

**static** **readonly** **object** lockObject = new **object()**;

**public** **static** **void** PrintNumbers**()**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name + " Trying to enter into the critical section"**)**;

Boolean IsLockTaken = **false**;

Monitor.Enter**(**lockObject, **ref** IsLockTaken**)**;

**try**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name + " Entered into the critical section"**)**;

**for** **(int** i = 0; i **<** 5; i++**)**

**{**

Thread.Sleep**(**100**)**;

Console.Write**(**i + ","**)**;

**}**

Console.WriteLine**()**;

**}**

**finally**

**{**

Monitor.Exit**(**lockObject**)**;

Console.WriteLine**(**Thread.CurrentThread.Name + " Exit from critical section"**)**;

**}**

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Thread**[]** Threads = new Thread**[**3**]**;

**for** **(int** i = 0; i **<** 3; i++**)**

**{**

Threads**[**i**]** = new Thread**(**PrintNumbers**)**;

Threads**[**i**]**.Name = "Child Thread " + i;

**}**

**foreach** **(**Thread t in Threads**)**

**{**

t.Start**()**;

**}**

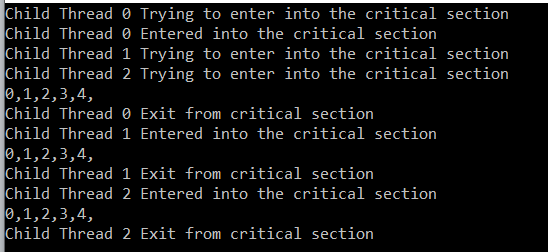
Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**



##### ****Difference between Monitor and lock in C#****

The **lock** is the shortcut for **Monitor.Enter with try and finally**. So, the lock provides the basic functionality to acquire an exclusive lock on a synchronized object. But, If you want more control to implement advanced multithreading solutions using TryEnter() Wait(), Pulse(), and PulseAll() methods, then the Monitor class is your option.

**Mutex in C#**

**Mutex in C# with Example**

In this article, I am going to discuss **how to use Mutex in C# in a multithread application** for thread synchronization with some examples. Please read our previous article where we discussed how to use the [**Monitor class to protect the shared resources**](https://dotnettutorials.net/lesson/multithreading-using-monitor/)from concurrent access in the multithread application. As part of this article, we are going to discuss the following pointers.

1. **What is Mutex in C#?**
2. **How to use Mutex in a multithread application to protect the shared resources?**
3. **Example to understand Mutex in C#.**

**What is Mutex in C#?**

Mutex also works likes a lock i.e. acquired an exclusive lock on a shared resource from concurrent access, but it works across multiple processes. As we already discussed exclusive locking is basically used to ensure that at any given point of time, only one thread can enter into the critical section.

The **Mutex** class provides the **WaitOne()** method which we need to call to lock the resource and similarly it provides **ReleaseMutex()** which is used to unlock the resource. Note that a **Mutex** can only be released from the same thread which obtained it.

**Example:**

The following example shows the use of the C# Mutex class. The code is self-explained. So, please go through the comment lines.

**using** *System;*

**using** *System.Threading;*

**namespace** *MutexDemo*

**{**

**class** Program

**{**

**private** **static** Mutex mutex = new Mutex**()**;

**static** **void** Main**(**string**[]** args**)**

**{**

//Create multiple threads to understand Mutex

**for** **(int** i = 1; i **<**= 5; i++**)**

**{**

Thread threadObject = new Thread**(**MutexDemo**)**;

threadObject.Name = "Thread " + i;

threadObject.Start**()**;

**}**

Console.ReadKey**()**;

**}**

//Method to implement syncronization using Mutex

**static** **void** MutexDemo**()**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name + " Wants to Enter Critical Section for processing"**)**;

**try**

**{**

//Blocks the current thread until the current WaitOne method receives a signal.

//Wait until it is safe to enter.

mutex.WaitOne**()**;

Console.WriteLine**(**"Success: " + Thread.CurrentThread.Name + " is Processing now"**)**;

Thread.Sleep**(**2000**)**;

Console.WriteLine**(**"Exit: " + Thread.CurrentThread.Name + " is Completed its task"**)**;

**}**

**finally**

**{**

//Call the ReleaseMutex method to unblock so that other threads

//that are trying to gain ownership of the mutex.

mutex.ReleaseMutex**()**;

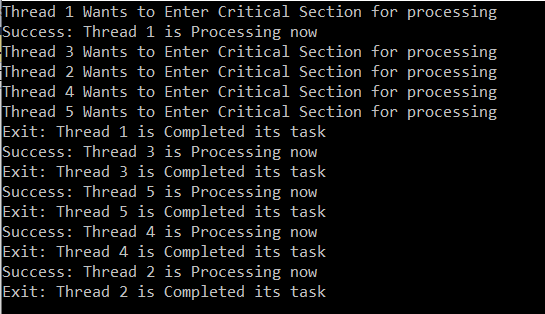
**}**

**}**

**}**

**}**

**Output:**



**Semaphore in C#**

**Semaphore in C# with Example**

In this article, I am going to discuss how to use **Semaphore in C#** with example. Please read our previous article where we discussed how to use [**Monitor in C#**](https://dotnettutorials.net/lesson/mutex-in-c/) to protect the shared resources from concurrent access in a multithread application. As part of this article, we are going to discuss the following pointers which are related to C# semaphore.

1. **What is Semaphore in C#?**
2. **How does Semaphore work in C#?**
3. **How to use the Semaphore class?**
4. **Understanding the different methods of Semaphore class with examples.**

**What is Semaphore in C#?**

The Semaphore in C# is used to limit the number of threads that can have access to a shared resource concurrently. In other words, we can say that Semaphore allows one or more threads to enter into the critical section and execute the task concurrently with thread safety. So, in real-time, we need to use Semaphore when we have a limited number of resources and we want to limit the number of threads that can use it.

**How does Semaphore work in C#?**

The Semaphores are Int32 variables that are stored in operating system resources. When we initialize the semaphore object we initialize it with a number. This number basically used to limits the threads that can enter into the critical section.

So, when a thread enters into the critical section, it decreases the value of the Int32 variable with 1 and when a thread exits from the critical section, it then increases the value of the Int32 variable with 1. The most important point that you need to remember is when the value of the Int32 variable is 0, then no thread can enter into the critical section.

**The syntax for semaphore initialization in C#:**

You can use the following statement to create the Semaphore instance in C#.

Semaphore in C# with an Example

As you can see in the above statement, we are passing two values to the constructor of the Semaphore class while initializing. These two values represent InitialCount and MaximumCount.

The InitialCount parameter sets the value for the Int32 variable. that is it defines the initial number of requests for the semaphore that can be granted concurrently. MaximumCount parameter defines the maximum number of requests for the semaphore that can be granted concurrently.

For example, if we set the maximum count value as 3 and initial count value 0, it means 3 threads are already in the critical section. If we set the maximum count value as 3 and the initial count value 2. It means a maximum of 3 threads can enter into the critical section and there is one thread that is currently in the critical section.

**Note:**The second parameter always must be equal or greater than the first parameter otherwise we will get an exception.

**Methods of Semaphore class:**

**WaitOne Method:**Threads can enter into the critical section by using the WaitOne method. We need to call the WaitOne method on the semaphore object. If the Int32 variable which is maintained by semaphore is greater than 0 then it allows the thread to enter into the critical section.

**Release Method:**We need to call the Release method when the thread wants to exits from the critical section. When this method is called, it increments the Int32 variable which is maintained by the semaphore object.

Let us see an example for a better understanding.

**using** *System;*

**using** *System.Threading;*

**namespace** *MutexDemo*

**{**

**class** Program

**{**

**public** **static** Semaphore semaphore = new Semaphore**(**2, 3**)**;

**static** **void** Main**(**string**[]** args**)**

**{**

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

**{**

Thread threadObject = new Thread**(**DoSomeTask**)**

**{**

Name = "Thread " + i

**}**;

threadObject.Start**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**static** **void** DoSomeTask**(object** id**)**

**{**

Console.WriteLine**(**Thread.CurrentThread.Name + " Wants to Enter into Critical Section for processing"**)**;

**try**

**{**

//Blocks the current thread until the current WaitHandle receives a signal.

semaphore.WaitOne**()**;

Console.WriteLine**(**"Success: " + Thread.CurrentThread.Name + " is Doing its work"**)**;

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

Console.WriteLine**(**Thread.CurrentThread.Name + "Exit."**)**;

**}**

**finally**

**{**

//Release() method to releage semaphore

semaphore.Release**()**;

**}**

**}**

**}**

**}**

**Deadlock in C#**

**Deadlock in C# with Example**

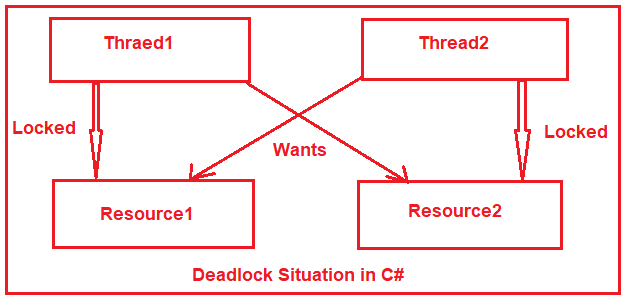
In this article, I am going to discuss **Deadlock in C#** with example. Please read our previous article where we discussed [**Semaphore in C#**](https://dotnettutorials.net/lesson/semaphore-in-multithreading/) with some examples. Deadlock is one of the most important aspects to understand as a developer. As part of this article, we are going to discuss the following pointers.

1. **What is deadlock?**
2. **Why Deadlock Occurred?**
3. **How a deadlock can occur in a multithreaded application?**
4. **How to avoid Deadlock by using Monitor.TryEnter method?**
5. **How to avoid Deadlock by acquiring locks in a specific order?**

**What is a Deadlock in C#?**

In simple words, we can define a deadlock in C# is a situation where two or more threads are **unmoving or frozen** in their execution because they are waiting for each other to finish.

For example, let’s say we have two threads **Thread1** and **Thread2** and at the same time let say we have two resources **Resource1** and **Resource2**. The **Thread1** locked the **Resource1** and trying to acquire a lock on **Respurce2**. At the same time, **Thread2** acquired a lock on **Resource2** and trying to acquire a lock on **Resource1**.



As you can see in the above image, **Thread1** is waiting to acquire a**lock on Resource2**which is held by **Thread2**. **Thread2** also can’t finish his work and release the lock on**Resource2** because it is waiting to acquire a lock on **Resource1** which is locked by **Thread1**, and hence a Deadlock situation occurred.

**Example to understand Deadlock in C#:**

Let us understand Deadlock in C# with an example. Create a class file with the name **Account.cs** and then copy and paste the following code in it.

**namespace** *DeadLockDemo*

**{**

**public** **class** Account

**{**

**public** **int** ID **{** **get**; **}**

**private** **double** Balance;

**public** Account**(int** id, **double** balance**)**

**{**

ID = id;

Balance = balance;

**}**

**public** **void** WithdrawMoney**(double** amount**)**

**{**

Balance -= amount;

**}**

**public** **void** DepositMoney**(double** amount**)**

**{**

Balance += amount;

**}**

**}**

**}**

The above Account class is very straight forward. We created the class with properties i.e. ID and Balance. Through the constructor of this class, we are initializing these properties. So, at the time of Account class instance creation, we need to pass the ID and Balance value. Here we have also created two methods. The WithdrawMoney method is used for withdrawing the amount while the DepositMoney method is used for adding the amount.

**AccountManager.cs:**

Create a class file with the name **AccountManager.cs** and then copy and paste the following code in it.

**using** *System;*

**using** *System.Threading;*

**namespace** *DeadLockDemo*

**{**

**public** **class** AccountManager

**{**

**private** Account FromAccount;

**private** Account ToAccount;

**private** **double** TransferAmount;

**public** AccountManager**(**Account AccountFrom, Account AccountTo, **double** AmountTransfer**)**

**{**

FromAccount = AccountFrom;

ToAccount = AccountTo;

TransferAmount = AmountTransfer;

**}**

**public** **void** FundTransfer**()**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} trying to acquire lock on {FromAccount.ID}"**)**;

**lock** **(**FromAccount**)**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} acquired lock on {FromAccount.ID}"**)**;

Console.WriteLine**(**$"{Thread.CurrentThread.Name} Doing Some work"**)**;

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

Console.WriteLine**(**$"{Thread.CurrentThread.Name} trying to acquire lock on {ToAccount.ID}"**)**;

**lock** **(**ToAccount**)**

**{**

FromAccount.WithdrawMoney**(**TransferAmount**)**;

ToAccount.DepositMoney**(**TransferAmount**)**;

**}**

**}**

**}**

**}**

**}**

In the above code, we created two Account type variables to hold the FromAccount and ToAccount details i.e. the Account from where the amount is going to deducted and the account to whom the amount is created. We also created another double type variable i.e. TransferAmount to hold the amount which is going to be deducted from the FromAccount and credited to the ToAccount. Through the constructor of this class, we are initializing the class variables.

We also created the FundTransfer method which is going to perform the required task. As you can see, it first acquires a lock on From Account and then doing some work. After 1 second it backs and trying to acquire a lock on To Account.

**Modifying the Main Method:**

Now modify the Main method of Program class as shown below. Here, for accountManager1, Account1001 is the FromAccount and Account1002 is the ToAccount. Similarly, for accountManager2, Account1002 is the FromAccount and Account1001 is the ToAccount

**using** *System;*

**using** *System.Threading;*

**namespace** *DeadLockDemo*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

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

Account Account1001 = new Account**(**1001, 5000**)**;

Account Account1002 = new Account**(**1002, 3000**)**;

AccountManager accountManager1 = new AccountManager**(**Account1001, Account1002, 5000**)**;

Thread thread1 = new Thread**(**accountManager1.FundTransfer**)**

**{**

Name = "Thread1"

**}**;

AccountManager accountManager2 = new AccountManager**(**Account1002, Account1001, 6000**)**;

Thread thread2 = new Thread**(**accountManager2.FundTransfer**)**

**{**

Name = "Thread2"

**}**;

thread1.Start**()**;

thread2.Start**()**;

thread1.Join**()**;

thread2.Join**()**;

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

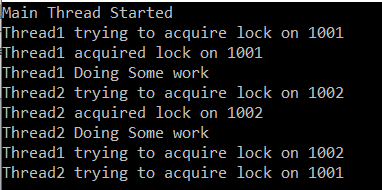
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Note:** For thread1, Account1001 is resource1 and Account1002 is resource2. On the other hand, for thread2, Account1002 is resource1 and Account1001 is resource2. With this keep in mind run the application and see deadlock occurred.

The reason is thread1 acquired an exclusive lock on Account1001 and then do some processing. Meantime thread2 started and it acquired an exclusive lock on Account1002 and then does some processing. Then thread1 back and wants to acquire a lock on Account1001 which is already locked by thread2. Similarly thread2 back and wants to acquire a lock on Account1002 which is already locked by thread1 and hence deadlock.

**Avoiding Deadlock by using Monitor.TryEnter method?**

The second parameter of **Monitor.TryEnter** method takes time out in milliseconds. Using that parameter we can specify a timeout for the thread to release the lock. If a thread is holding a resource for a long time while the other thread is waiting, then Monitor will provide a time limit and force the lock to release it. So that the other thread will enter into the critical section.

Modifying the **AccountManager** class as shown below:

**using** *System;*

**using** *System.Threading;*

**namespace** *DeadLockDemo*

**{**

**public** **class** AccountManager

**{**

**private** Account FromAccount;

**private** Account ToAccount;

**private** **double** TransferAmount;

**public** AccountManager**(**Account AccountFrom, Account AccountTo, **double** AmountTransfer**)**

**{**

this.FromAccount = AccountFrom;

this.ToAccount = AccountTo;

this.TransferAmount = AmountTransfer;

**}**

**public** **void** FundTransfer**()**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} trying to acquire lock on {FromAccount.ID}"**)**;

**lock** **(**FromAccount**)**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} acquired lock on {FromAccount.ID}"**)**;

Console.WriteLine**(**$"{Thread.CurrentThread.Name} Doing Some work"**)**;

Thread.Sleep**(**3000**)**;

Console.WriteLine**(**$"{Thread.CurrentThread.Name} trying to acquire lock on {ToAccount.ID}"**)**;

**if** **(**Monitor.TryEnter**(**ToAccount, 3000**))**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} acquired lock on {ToAccount.ID}"**)**;

**try**

**{**

FromAccount.WithdrawMoney**(**TransferAmount**)**;

ToAccount.DepositMoney**(**TransferAmount**)**;

**}**

**finally**

**{**

Monitor.Exit**(**ToAccount**)**;

**}**

**}**

**else**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} Unable to acquire lock on {ToAccount.ID}, So existing."**)**;

**}**

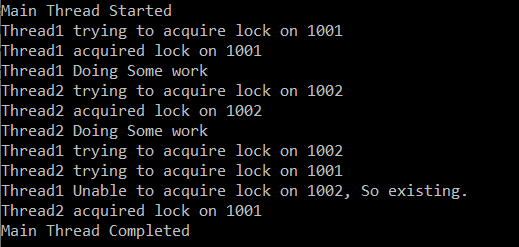
**}**

**}**

**}**

**}**

**Output:**



As you can see in the output thread1 release the lock and exists from the critical section which allows thread2 to enter the critical section.

**How to avoid Deadlock in C# by acquiring locks in a specific order?**

Please modify the AccountManager class as shown below.

**using** *System;*

**using** *System.Threading;*

**namespace** *DeadLockDemo*

**{**

**public** **class** AccountManager

**{**

**private** Account FromAccount;

**private** Account ToAccount;

**private** **readonly** **double** TransferAmount;

**private** **static** **readonly** Mutex mutex = new Mutex**()**;

**public** AccountManager**(**Account AccountFrom, Account AccountTo, **double** AmountTransfer**)**

**{**

this.FromAccount = AccountFrom;

this.ToAccount = AccountTo;

this.TransferAmount = AmountTransfer;

**}**

**public** **void** FundTransfer**()**

**{**

**object** \_lock1, \_lock2;

**if** **(**FromAccount.ID **<** ToAccount.ID**)**

**{**

\_lock1 = FromAccount;

\_lock2 = ToAccount;

**}**

**else**

**{**

\_lock1 = ToAccount;

\_lock2 = FromAccount;

**}**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} trying to acquire lock on {((Account)\_lock1).ID}"**)**;

**lock** **(**\_lock1**)**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} acquired lock on {((Account)\_lock1).ID}"**)**;

Console.WriteLine**(**$"{Thread.CurrentThread.Name} Doing Some work"**)**;

Thread.Sleep**(**3000**)**;

Console.WriteLine**(**$"{Thread.CurrentThread.Name} trying to acquire lock on {((Account)\_lock2).ID}"**)**;

**lock(**\_lock2**)**

**{**

Console.WriteLine**(**$"{Thread.CurrentThread.Name} acquired lock on {((Account)\_lock2).ID}"**)**;

FromAccount.WithdrawMoney**(**TransferAmount**)**;

ToAccount.DepositMoney**(**TransferAmount**)**;

**}**

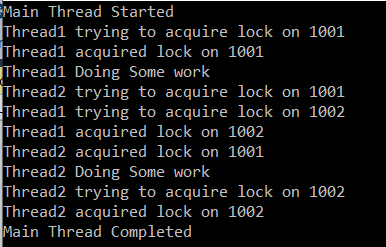
**}**

**}**

**}**

**}**

**Output:**



# Performance Testing of a Multithreaded Application

## ****Performance Testing of a Multithreaded Application in C#****

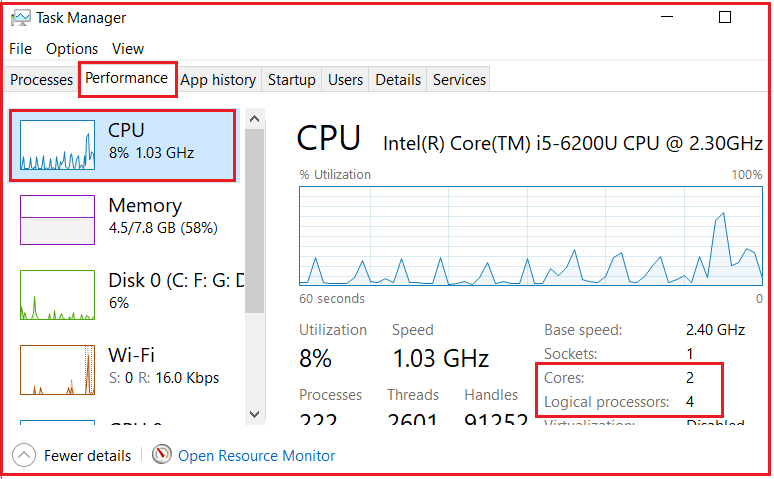
In this article, I am going to discuss the **Performance Testing of a multithreaded application in C#** with an example. Please read our previous article where we discussed [**Deadlock in C#**](https://dotnettutorials.net/lesson/deadlock-in-csharp/). As part of this article, I will show you the performance implications of a multithreaded program when we run the application on a machine having a single core/processor versus a machine multi-core/processor.

#### ****How to find out how many processors you have on your machine?****

You can find out how many processors you have on your machine in many different ways. Some of them are as follows:

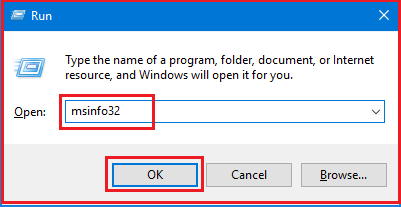
##### ****Way1:**** ****Using Task Manager****

Right-click on the Taskbar and select the “**Task Manager**” option from the context menu. Then click on the “**Performance”** tab and select the “**CPU**” from the left side panel and then you will see the cores and Logical processors on the right side as shown in the below image.

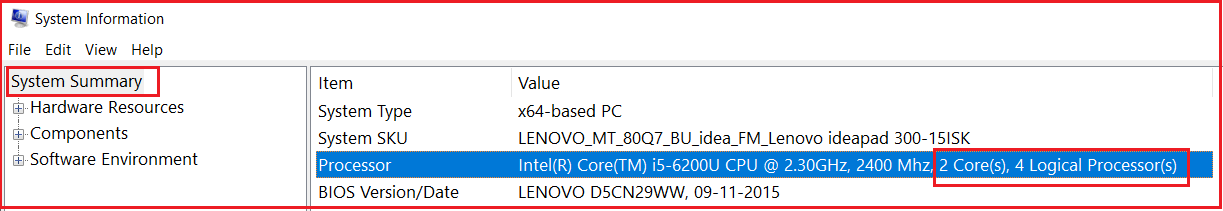


##### ****Way2:**** ****Using msinfo32 Command****

Press the **Windows key + R** to open the **Run** command, then type **msinfo32** and click on the **OK** button as shown in the below image.



Once you click on the OK button, it will open up the System Information app. From that select the **Summary** option and scroll down until you find Processor. The details will tell you both how many cores and logical processors your CPU has as shown in the below image.



##### ****Way3: Using dot net code.****

You can use the following code in any type of .net application to find out the total processors on the machine.  
**Environment.ProcessorCount**

##### ****Way4: Using the command prompt****

On the Windows command prompt write the following code and press enter

**echo %NUMBER\_OF\_PROCESSORS%**

##### ****Performance Testing With Multiple Processors:****

If you have a machine with multiple processors, then multiple threads can execute your application code in parallel on different cores. For example, if your machine has two cores and there are two threads to execute your application code, then each thread is going to be run on an individual core. As a result, we will get better performance.

If you have two threads and if each thread takes 10 milliseconds to complete the execution, then on a machine with 2 processors, the total time taken is 10 milliseconds.

##### ****Performance Testing With Single Processor:****

If you have a machine with a single processor, then multiple threads are going to execute one after the other. It is not possible for a single-core processor machine to execute multiple threads in parallel. The operating system switches between the threads so fast, it just gives us the illusion that the threads are running in parallel. On a single core or processor machine, multiple threads can negatively affect the performance as there is overhead involved with context-switching.

If you have two threads and if each thread takes 10 milliseconds to complete the execution, then on a machine with a single processor, the total time taken is 20 milliseconds plus thread context switching time if any.

##### ****Example: Using Single Thread****

**using** *System;*

**using** *System.Diagnostics;*

**using** *System.Threading;*

**namespace** *MultithreadingPerformanceTesting*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

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

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

EvenNumbersSum**()**;

OddNumbersSum**()**;

stopwatch.Stop**()**;

Console.WriteLine**(**$"Total time in milliseconds : {stopwatch.ElapsedMilliseconds}"**)**;

Console.ReadKey**()**;

**}**

**public** **static** **void** EvenNumbersSum**()**

**{**

**double** Evensum = 0;

**for** **(int** count = 0; count **<**= 50000000; count++**)**

**{**

**if** **(**count % 2 == 0**)**

**{**

Evensum = Evensum + count;

**}**

**}**

Console.WriteLine**(**$"Sum of even numbers = {Evensum}"**)**;

**}**

**public** **static** **void** OddNumbersSum**()**

**{**

**double** Oddsum = 0;

**for** **(int** count = 0; count **<**= 50000000; count++**)**

**{**

**if** **(**count % 2 == 1**)**

**{**

Oddsum = Oddsum + count;

**}**

**}**

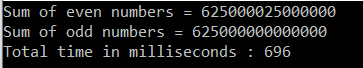
Console.WriteLine**(**$"Sum of odd numbers = {Oddsum}"**)**;

**}**

**}**

**}**

**Output:**



As you can see it takes approximately 696 milliseconds to complete the execution.

##### ****Example: Using Multiple Threads****

Let’s rewrite the previous example using multiple threads and compare the output.

**using** *System;*

**using** *System.Diagnostics;*

**using** *System.Threading;*

**namespace** *MultithreadingPerformanceTesting*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

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

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

Thread thread1 = new Thread**(**EvenNumbersSum**)**;

Thread thread2 = new Thread**(**OddNumbersSum**)**;

thread1.Start**()**;

thread2.Start**()**;

thread1.Join**()**;

thread2.Join**()**;

stopwatch.Stop**()**;

Console.WriteLine**(**$"Total time in milliseconds : {stopwatch.ElapsedMilliseconds}"**)**;

Console.ReadKey**()**;

**}**

**public** **static** **void** EvenNumbersSum**()**

**{**

**double** Evensum = 0;

**for** **(int** count = 0; count **<**= 50000000; count++**)**

**{**

**if** **(**count % 2 == 0**)**

**{**

Evensum = Evensum + count;

**}**

**}**

Console.WriteLine**(**$"Sum of even numbers = {Evensum}"**)**;

**}**

**public** **static** **void** OddNumbersSum**()**

**{**

**double** Oddsum = 0;

**for** **(int** count = 0; count **<**= 50000000; count++**)**

**{**

**if** **(**count % 2 == 1**)**

**{**

Oddsum = Oddsum + count;

**}**

**}**

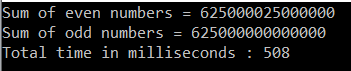
Console.WriteLine**(**$"Sum of odd numbers = {Oddsum}"**)**;

**}**

**}**

**}**

**Output:**



# Thread Pooling in C#

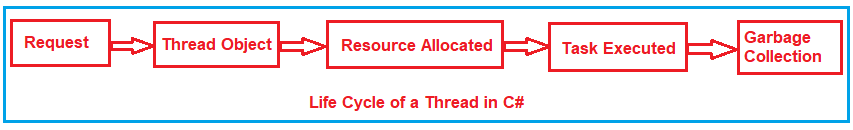
## ****Thread Pool in C#****

In this article, I am going to discuss **Thread Pool in C#** with examples. Please read our previous article where we discussed the [**Performance Testing of a multithreaded application**](https://dotnettutorials.net/lesson/performance-testing-of-a-multithreaded-application/)in C#**.**As part of this article, we are going to discuss the following pointers.

1. **The Request Life cycle of a Thread.**
2. **What is Thread Pooling in C#?**
3. **Why do we need C# Thread Pool?**
4. **Performance testing between normal thread and thread pooling**

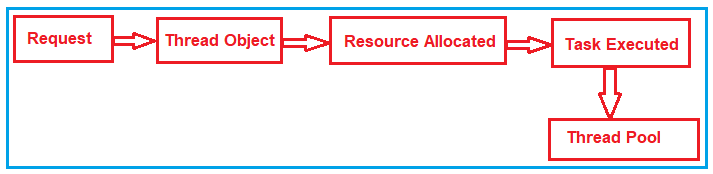
##### ****The Request Life cycle of a Thread in C# with Example.****

Let us understand the life cycle of a thread in C#. In order to understand this, please have a look at the following image. When the .NET framework receives a request (the request can be a method call or function call from any kind of application). To that handle request, a thread object is created. When the thread object is created some resources are allocated to that thread object such as memory. After then the task is executed and once the task is completed then the garbage collector removes that thread object for free-up memory allocation. This is the life cycle of a thread in C#.



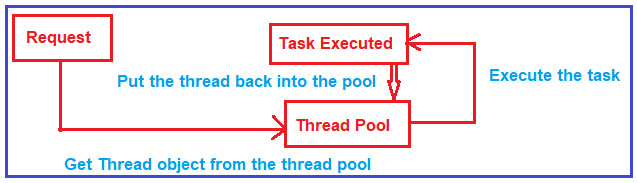
These steps are going to be repeated again and again for each request that comes in a multithread application. That means every time a new thread object created and get allocated in the memory. If there are many requests then there will be many thread objects and if there are many thread objects then there will be load on the memory which slows down your application.

There is a great room for performance improvements. The Thread object is created, resources are allocated, the task is executed, and then it should not go for garbage collection, instead of how about taking the thread object and put it into a pool as shown in the below image. This is where thread pooling comes into the picture.



##### ****Thread Pool in C#:****

Thread pool in C# is nothing but a collection of threads that can be reused to perform no of tasks in the background. Now when a request comes, then it directly goes to the thread pool and checks whether there are any free threads available or not. If available, then it takes the thread object from the thread pool and executes the task as shown in the below image.



Once the thread completes its task then it again sent back to the thread pool so that it can reuse. This reusability avoids an application to create the number of threads and this enables less memory consumption.

##### ****How to use C# Thread Pool?****

Let us see a simple example to understand how to use Thread Pooling. Once you understand how to use thread pooling then we will see the performance benchmark between the normal thread object and thread pool.

###### **Step1:**

In order to implement thread pooling in C#, first, we need to import the Threading namespace as ThreadPool class belongs to this namespace as shown below.

**using System.Threading;**

###### **Step2:**

Once you import the Threading namespace, then you need to use the **ThreadPool** class and using this class you need to call the **QueueUserWorkItem** static method. If you go to the definition of the **QueueUserWorkItem** method, then you will see that this method takes one parameter of type **WaitCallback** object. While creating the object of the **WaitCallback** class, you need to pass the method name that you want to execute.

**ThreadPool.QueueUserWorkItem(new WaitCallback(MyMethod));**

Here, the **QueueUserWorkItem** method Queues the function for execution and that function executes when a thread becomes available from the thread pool. If no thread is available then it will wait until one thread gets freed. Here MyMethod is the method that we want to execute by a thread pool thread.

##### ****The complete code is given below.****

As you can see in the below code, here, we create one method that is MyMethod and as part of that method, we simply printing the thread id, whether the thread is a background thread or not and whether it is from thread pool or not. And we want to execute this method 10 times using the thread pool threads. So, here we use a simple for each loop and use the ThreadPool class and call that method.

**using** *System;*

**using** *System.Threading;*

**namespace** *ThreadPoolApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

ThreadPool.QueueUserWorkItem**(**new WaitCallback**(**MyMethod**))**;

**}**

Console.Read**()**;

**}**

**public** **static** **void** MyMethod**(object** obj**)**

**{**

Thread thread = Thread.CurrentThread;

string message = $"Background: {thread.IsBackground}, Thread Pool: {thread.IsThreadPoolThread}, Thread ID: {thread.ManagedThreadId}";

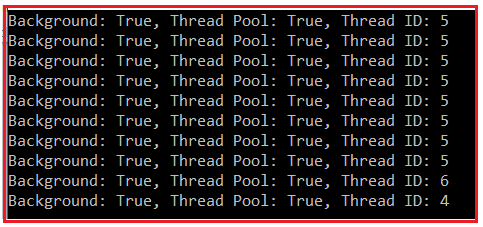
Console.WriteLine**(**message**)**;

**}**

**}**

**}**

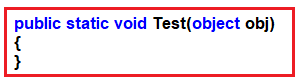
Once you execute the above code, it will give you the following output. As you can see, it shows that it is a background thread and this thread is from the thread pool and the thread Ids may vary in your output. Here, you can see three threads handle all the 10 method calls.



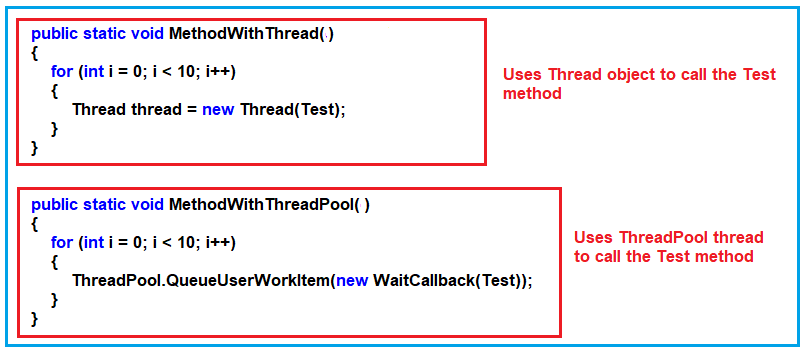
##### ****Performance testing using and without using Thread Pool in C# with Example:****

Let us see an example to understand the performance benchmark. Here, we will compare how much time does the thread object takes and how much time does the thread pool thread takes to do the same task i.e. to execute the same methods.

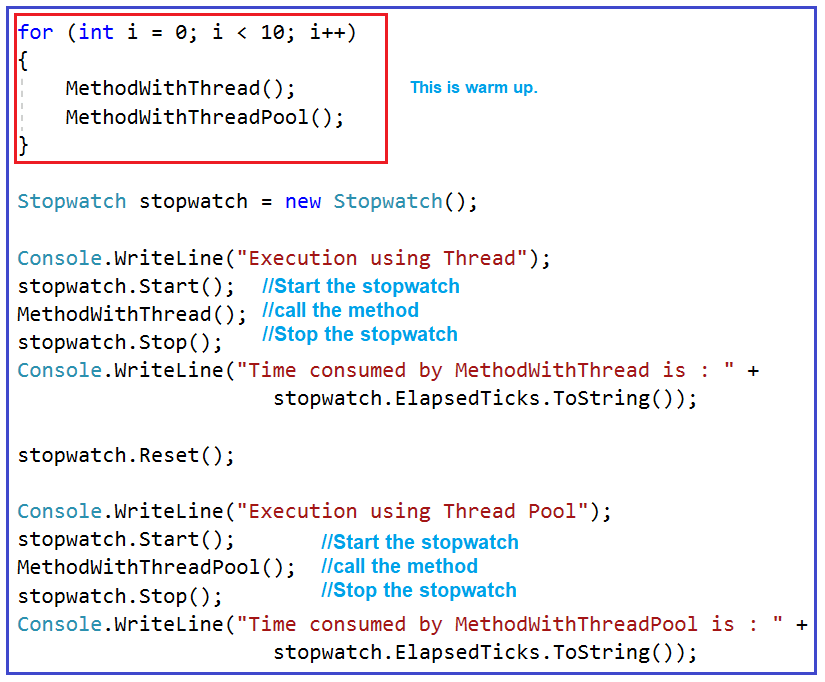
In order to do this, what we are going to do is, we will create a method called **Test** as shown below. This method takes an input parameter of type object and as part of that Test method we are doing nothing means an empty method.



Then we will create two methods such as **MethodWithThread** and **MethodWithThreadPool** and inside these two methods, we will create one for loop which will execute 10 times. Within for loop, we are going to call the Test as shown below. As you can see, the **MethodWithThread** method uses the Thread object to call the Test method while the **MethodWithThreadPool** method uses the ThreadPool object to call the Test method.



Now we need to call the above two methods (**MethodWithThread** and **MethodWithThreadPool**) from the main method. As we are going to test the performance benchmark, so we are going to call these two methods between the stopwatch start and end as shown below. The Stopwatch class is available in **System.Diagnostics** namespace. The for loop within the Main method is for warm-up. This is because when we run the code for the first time, compilation happens and compilation takes some time and we don’t want to measure that.



##### ****The complete code is given below.****

**using** *System;*

**using** *System.Diagnostics;*

**using** *System.Threading;*

**namespace** *ThreadPoolApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

MethodWithThread**()**;

MethodWithThreadPool**()**;

**}**

Stopwatch stopwatch = new Stopwatch**()**;

Console.WriteLine**(**"Execution using Thread"**)**;

stopwatch.Start**()**;

MethodWithThread**()**;

stopwatch.Stop**()**;

Console.WriteLine**(**"Time consumed by MethodWithThread is : " +

stopwatch.ElapsedTicks.ToString**())**;

stopwatch.Reset**()**;

Console.WriteLine**(**"Execution using Thread Pool"**)**;

stopwatch.Start**()**;

MethodWithThreadPool**()**;

stopwatch.Stop**()**;

Console.WriteLine**(**"Time consumed by MethodWithThreadPool is : " +

stopwatch.ElapsedTicks.ToString**())**;

Console.Read**()**;

**}**

**public** **static** **void** MethodWithThread**()**

**{**

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

Thread thread = new Thread**(**Test**)**;

**}**

**}**

**public** **static** **void** MethodWithThreadPool**()**

**{**

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

ThreadPool.QueueUserWorkItem**(**new WaitCallback**(**Test**))**;

**}**

**}**

**public** **static** **void** Test**(object** obj**)**

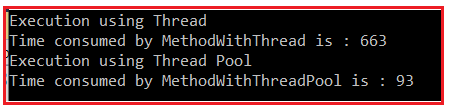
**{**

**}**

**}**

**}**

**Output:**



As you can see in the above output, the Time consumed by MethodWithThread is 663 and the Time consumed by MethodWithThreadPool is 93. If you observe there is a vast time difference between these two.

So it proofs that the thread pool gives better performance as compared to the thread class object. If there are needs to create one or two threads then you need to use Thread class object while if there is a need to create more than 5 threads then you need to go for thread pool class in a multithreaded environment.