**Asynchronous programming and Threading in C# (.Net 4.5)**

Asynchronous programming and threading is very important feature for concurrent or parallel programming. Asynchronous programming may or may not use threading. Still if we can see them together, I think we may have better understanding of both the features.

**Topics covered in this article**

1. Asynchronous programming
2. Is threading required
3. Task based Asynchronous Programming
4. **Parallel programming:**
5. Conclusion

**Asynchronous Programming**

Asynchronous operation means that the operation runs independent of main or other process flow. In general c# program starts executing from the Main method and ends when the Main method returns. In between all the operations runs sequentially one after another. One operation must wait until its previous operation finishes. Let’s see following code:

static void Main(string[] args)

{

DoTaskOne();

DoTaskTwo();

}

Method “DoTaskTwo” would not be started until “DoTaskOne” finishes. In other words method “DoTaskOne” blocks the execution as long it takes to finish.

In asynchronous programming a method is called that runs in the background and the calling thread is not blocked. After calling the method the execution flow immediately backs to calling thread and performs other tasks. Normally it uses Thread or Task (We will discuss Thread and Task in detail later).

In our case if we run the “DoTaskOne” asynchronously, after calling the “DoTaskOne” method, execution flow immediately backs to Main method and start “DoTaskTwo”.

We can create our own thread using Thread class or use asynchronous patterns provided by .Net to perform asynchronous programming. There are three different asynchronous patterns in .Net:

1. Asynchronous Programming Model (APM) pattern
2. Event-based Asynchronous Pattern (EAP)

Both the above models are not recommended by Microsoft so we will not discuss about them. If you are interested you can read more from following msdn link:

<https://msdn.microsoft.com/en-us/library/ms228963(v=vs.110).aspx> and

<https://msdn.microsoft.com/en-us/library/ms228969(v=vs.110).aspx>

1. Task-based Asynchronous Pattern (TAP): This model is recommended so we will discuss it in detail

**Threading is required or not**

If we use asynchronous programming pattern that .Net introduced in 4.5, in most of the cases we need not to create manual thread by us. The compiler does the difficult work that the developer used to do.

Creating a new tread is costly, it takes time. Unless we need to control a thread, then “Task-based Asynchronous Pattern (TAP)” and “Task Parallel Library (TPL)” is good enough for asynchronous and parallel programming. TAP and TPL uses Task (we will discuss what is Task latter). In general Task uses the thread from ThreadPool(A *thread pool* is a collection of threads already created and maintained by .Net framework. If we use Task, most of the cases we need not to use thread pool directly. Still if you want to know more about thread pool visit the link: ”https://msdn.microsoft.com/en-us/library/h4732ks0.aspx” )

But Task can be run:

1. In the current thread
2. In a new thread
3. In a thread from the thread pool
4. Or even without any thread

But if we use Task, as a developer we need not to worry about creation or uses of the thread, .Net framework handles the inner difficulties for us.

Anyway if we need some control over the thread like,

1. We want to set a name for the tread
2. We want to set priority for the thread
3. We want to make our thread foreground or background

Then we may have to create our own thread using thread class.

**Creating Thread using Thread class**

The constructor of Thread class accepts a delegate parameter of type

1. ThreadStart: This delegate defines a method with a void return type and no parameter.
2. And ParameterizedThreadStart: This delegate defines a method with a void return type and one object type parameter.

Following is the simple example how we can start a new thread with Start method:

static void Main(string[] args)

{

Thread thread = new Thread(DoTask);

thread.Start();// Start DoTask method in a new thread

//Do other tasks in main thread

}

static public void DoTask() {

//do something in a new thread

}

We can use lamda expression instead of named method:

static void Main(string[] args)

{

Thread thread = new Thread(() => {

//do something in a new thread

});

thread.Start();// Start a new thread

//Do other tasks in main thread

}

If we don’t require the variable reference we can even start the thread directly like:

static void Main(string[] args)

{

new Thread(() => {

//do something in a new thread

}).Start();// Start a new thread

//Do other tasks in main thread

}

But if we want to control the tread object after it is created we require the variable reference. We can assign different values to the different properties of the object like:

static void Main(string[] args)

{

Thread thread = new Thread(DoTask);

thread.Name = "My new thread";// Asigning name to the thread

thread.IsBackground = false;// Made the thread forground

thread.Priority = ThreadPriority.AboveNormal;// Setting thread priority

thread.Start();// Start DoTask method in a new thread

//Do other task in main thread

}

With the reference variable we can perform some function like abort the thread or wait for the tread to complete by calling the join method. If we call join to a thread the main tread blocks until the calling thread completes.

If we want to pass some data to the method we can pass it as a parameter of Start method. As the method parameter is object type we need to cast it properly.

static void Main(string[] args)

{

Thread thread = new Thread(DoTaskWithParm);

thread.Start("Passing string");// Start DoTaskWithParm method in a new thread

//Do other task in main thread

}

static public void DoTaskWithParm(object data)

{

//we need to cast the data to appropriate object

}

**“async” and “await” keywords**

.Net framework introduced two new keywords to perform asynchronous programing: “async” and “await”. To use “await” keyword within a method we need to declare the method with “async” modifier. “await” keyword is used before calling an asynchronous method. “await” keyword suspends further execution of the method and control is return to the calling thread. See the example:

private **async** static void CallerWithAsync()// async modifier is used

{

string result = **await** GetSomethingAsync();// await is used before a method call. It suspends //execution of CallerWithAsync() method and control returs to the calling thread that can //perform other task.

Console.WriteLine(result);// this line would not be executed before GetSomethingAsync() //method completes

}

The “async” modifier can only be used with methods returning a Task or void. It cannot be used with the entry point of a program, the Main method.

We cannot use await keyword before all the methods. To use “await” the method must have to return “awaitable” type. Following are the types that are “awaitable”:

1. Task
2. Task<T>
3. Custom “awaitable” type. Using a custom type is a rare and advanced scenario; we will not discuss it here.

**Task-based Asynchronous Pattern**

First of all we need an asynchronous method that returns Task or Task<T>. We can create Task by following ways:

1. Task.Factory.StartNew method: Prior to .Net 4.5 (in .Net 4) this was the primary method to create and schedule a task.
2. Task.Run or Task.Run<T> Method: From .Net 4.5 this method should be used. This method is sufficient for most of the common cases.
3. Task.FromResult method:  If the result is already computed, we can use this method to create a task.

Task.Factory.StartNew has still some important uses for advance scenario. Please see the link for more information: “http://blogs.msdn.com/b/pfxteam/archive/2011/10/24/10229468.aspx”

Following links shows some ways to create Task: “http://dotnetcodr.com/2014/01/01/5-ways-to-start-a-task-in-net-c/”

**Creating and awaiting for a Task**

We will create our Task using Task.Run<T> method. This method Queues the specified work to run on the ThreadPool and returns a task handle for that work. Following steps are needed to create an asynchronous Task from a synchronous method:

1. Let’s assume that we have a method that is synchronous but take some time to complete:

static string Greeting(string name)

{

Thread.Sleep(3000);

return string.Format("Hello, {0}", name);

}

1. To access this method asynchronously we have to wrap it with an asynchronous method. Let’s assume the name is “GreetingAsync”. It is a convention to add “Async” suffix to the name of an asynchronous method.

static **Task<string>** Greeting**Async**(string name)

{

**return Task.Run<string>(() =>**

**{**

**return Greeting(name);**

**});**

}

1. Now we can call the asynchronous method GreetingAsync by using the await keyword

private **async** static void CallWithAsync()

{

//some other tasks

string result = **await** GreetingAsync("Bulbul");

//We can add multiple “await” in same “async” method like

//string result1 = **await** GreetingAsync(“Ahmed”);

//string result2 = **await** GreetingAsync(“Every Body”);

Console.WriteLine(result);

}

When “CallWithAsync” method is called it starts executing like regular synchronous method until it reaches “await” keyword. When it reaches to the “await” keywords it poses execution for the method and start waiting for “GreetingAsync("Bulbul")” method to be finished. In the meantime the control returns to the caller of “CallWithAsync” method and the caller can do its other task as usual.

When “GreetingAsync("Bulbul")” method finishes, “CallWithAsync” method resumes its other task after “await” keywords. In this case it executes the code “Console.WriteLine(result)”

1. Continuation with Task: “ContinueWith” method of Task class defines the code that should be invoked as soon as the task is completed.

private static void CallWithContinuationTask()

{

Task<string> t1 = GreetingAsync("Bulbul");

**t1.ContinueWith**(t =>

{

string result = t.Result;

Console.WriteLine(result);

});

}

We need not to use “await” keyword if we use “ContinueWith” method, compiler will put the “await” keyword into appropriate place.

**Awaiting for multiple asynchronous methods:**

Let us look at the following code:

private **async** static void CallWithAsync()

{

string result = **await** GreetingAsync("Bulbul");

string result1 = **await** GreetingAsync(“Ahmed”);

Console.WriteLine(result);

Console.WriteLine(result1);

}

Here we are awaiting for the two calling sequentially. Second call of the “GreetingAsync(“Ahmed”)” will be started after finishing the first call “GreetingAsync("Bulbul")”. If “result” and “result1” of the above code are not dependent, then the sequential “awiting” is not a good practice.

In that case we can simply call the methods without “await” keywords and “awaits” for them in a single place by combinators. In that case both the method call can be executed in parallel.

private async static void MultipleAsyncMethodsWithCombinators()

{

**Task<string> t1** = GreetingAsync("Bulbul");

**Task<string> t2** = GreetingAsync("Ahmed");

**await Task.WhenAll(t1, t2);**

Console.WriteLine("Finished both methods.\n " +

"Result 1: {0}\n Result 2: {1}", t1.Result, t2.Result);

}

Here we use Task.WhenAll combinator. Task.WhenAll creates a task that will complete when all of the supplied tasks have completed. Task class has another combinator. Task.WhenAny, that will complete when any of the supplied tasks have completed.

**Handling Exceptions:**

We have to put “await” code blocks inside a try block to handle the exception of the method.

private **async** static void CallWithAsync()

{

**try**

**{**

string result = **await** GreetingAsync("Bulbul");

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(“handled {0}”, ex.Message);**

**}**

}

If we have multiple “await” inside the try block only the first exception will be handled and the next “await” would not be reached. If we want all the methods to be called even some one throw exception we have to call them without “await” keyword and wait for all the tasks using Task.WhenAll method.

private **async** static void CallWithAsync()

{

**try**

**{**

**Task<string> t1** = GreetingAsync("Bulbul");

**Task<string> t2** = GreetingAsync("Ahmed");

**await Task.WhenAll(t1, t2);**

**}**

**catch (Exception ex)**

**{**

**Console.WriteLine(“handled {0}”, ex.Message);**

**}**

**}**

Although all tasks will be completed, we can see exception only from first task. It’s not the task that threw the exception first, but the first task in the list.

One way to get the error from all the tasks is to declare them outside the try block so that the can be accessed from exception block and then check the “IsFaulted” property of the task. If it has an exception then the “IsFaulted” property will be true. And then we can get the exception by the inner exception of the task instances.

But there is another better way like this:

static async void ShowAggregatedException()

{

**Task taskResult = null;**

try

{

**Task<string> t1** = GreetingAsync("Bulbul");

**Task<string> t2** = GreetingAsync("Ahmed");

**await (taskResult = Task.WhenAll(t1, t2));**

}

catch (Exception ex)

{

Console.WriteLine("handled {0}", ex.Message);

foreach (var innerEx in **taskResult.Exception.InnerExceptions)**

{

Console.WriteLine("inner exception {0}", nnerEx.Message);

}

}

}

**Canceling the Task:**

Previously if we used thread from ThreadPool, it was not possible to cancel the thread. Now Task class provides a way to cancel the started task based on the CancellationTokenSource class, Steps to cancel a task:

1. The asynchronous method should except a parameter of type “CancellationToken”
2. Create an instance of CancellationTokenSource class like:

var **cts = new CancellationTokenSource();**

1. Pass the CancellationToken from the instace to the asynchronous method, like:

**Task<string> t1** = GreetingAsync("Bulbul", **cts.Token**);

1. From the long running method, we have to call **ThrowIfCancellationRequested() method of** CancellationToken.

static string Greeting(string name, CancellationToken token)

{

Thread.Sleep(3000);

token. **ThrowIfCancellationRequested();**

return string.Format("Hello, {0}", name);

}

1. Catch the **OperationCanceledException from where we are awiting for the Task.**
2. Now if we cancel the operation by calling Cancel method of instance of CancellationTokenSource, OperationCanceledException will be thrown from the long running operation. We can set time to cancel the operation to the instanc also. For more detail about CancellationTokenSource class please see following link: “https://msdn.microsoft.com/en-us/library/system.threading.cancellationtokensource%28v=vs.110%29.aspx**”**

Let us see the whole things in an example code, in this example we are canceling the operation after one second:

static void Main(string[] args)

{

CallWithAsync();

Console.ReadKey();

}

async static void CallWithAsync()

{

try

{

CancellationTokenSource source = new CancellationTokenSource();

source.CancelAfter(TimeSpan.FromSeconds(1));

var t1 = await GreetingAsync("Bulbul", source.Token);

}

catch (OperationCanceledException ex)

{

Console.WriteLine(ex.Message);

}

}

static Task<string> GreetingAsync(string name, CancellationToken token)

{

return Task.Run<string>(() =>

{

return Greeting(name, token);

});

}

static string Greeting(string name, CancellationToken token)

{

Thread.Sleep(3000);

token.ThrowIfCancellationRequested();

return string.Format("Hello, {0}", name);

}

**Parallel programming:**

**.Net 4.5 and above has introduced a class called “Parallel”, an abstraction over the thread class. Using the “Parallel” class we can implement parallelism. Parallelism differs from the Threading in a way that it uses all the available CPU or core. Two type of parallelism is possible,**

1. **Data Parallelism: If we have a big collection of data and we want some operation on each of the data to perform parallely then we can use data parallelism. Parallel class has static For or ForEach method to perform data parallelism, like**

ParallelLoopResult result =

Parallel.For(0, 100, async (int i) =>

{

Console.WriteLine("{0}, task: {1}, thread: {2}", i,

Task.CurrentId, Thread.CurrentThread.ManagedThreadId);

await Task.Delay(10);

});

Parallel For or ForEach may use several threads and the index (in the code i) is not sequential.

If we want to stop parallel For o ForEach method earlier we may pass ParallelLoopState as a parameter and based on the state we can break the loop.

ParallelLoopResult result =

Parallel.For(0, 100, async (int i, ParallelLoopState pls) =>

{

Console.WriteLine("{0}, task: {1}, thread: {2}", i,

Task.CurrentId, Thread.CurrentThread.ManagedThreadId);

await Task.Delay(10);

if (i > 5) pls.Break();

});

Be careful while breaking the loop, as it is running in several threads may run larger than the break point. We should not take any decision based on the break point.

1. **Task Parallelism: If we want to run multiple task in parallel we can use task parallelism by calling the invoke method of Parallel class. Parallel.Invoke method accepts an array of Action delegate. For example:**

static void ParallelInvoke()

{

**Parallel.Invoke(MethodOne, MethodTwo);**

}

**Conclusion:**

I have tried to introduce asynchronous programming technique that .Net framework 4.5 provided. I have tried to keep the things simple and did not go into advanced detail. Many of the examples and references are taken from the [**Professional C# 2012 and .NET 4.5 of Wrox**](https://www.google.com.bd/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CBwQFjAA&url=http%3A%2F%2Fwww.wrox.com%2FWileyCDA%2FWroxTitle%2FProfessional-C-2012-and-NET-4-5.productCd-1118314425.html&ei=gJdpVdv5E5CJuASMnIHACw&usg=AFQjCNFJf7f-u9cLJoh3TvGzFSknQxBWmA&sig2=VN21AO_MOXe9wxecFdAMiA&bvm=bv.94455598,d.c2E)**.**