Asynchronous Methods

Introduction

- Some of the classes in the .NET Framework Base Class Library (BCL) provide both synchronous and asynchronous method signatures.
- Calling a synchronous method can create a delay in our program flow.
- Calling an asynchronous method might result in better performance in cases such as when a program needs to send out requests to multiple Web services.

Synchronous vs Asynchronous

- When calling a synchronous method the execution waits for the method to complete its execution before moving forward.
- When calling an asynchronous method the execution returns immediately so that the program can perform other operations while the called method completes its work.

Multiple Threads Problem

- When having a code that needs to be executed concurrently, assigning a dedicated thread for each one of the executions might consume a huge amount of memory.
- In most cases, each one of the threads isn't always busy. In most cases each thread consumes a fraction of the time it was allocated with.

The Asynchronous Method Pattern

- The asynchronous method pattern allows a handful of fully utilized threads to take on thousands of concurrent executions.
- If each and every thread is busy during its entire execution the asynchronous method pattern wouldn't have been relevant.

Optimizing Threads Resources

- The purpose of using asynchronous methods address isn't to provide a convenient mechanism for executing methods concurrently.
- We use asynchronous methods in order to optimize thread resources.
- The asynchronous method aims at getting a situation in which none of the threads is blocked. This way, all threads will be exploited to their maximum potential.

The Web Request Case Study

- When having a thread dedicated to processing a single web request we might find it spending 99 percent of its time blocked... waiting for the server to return its reply.
- The asynchronous method pattern exploits this potential allowing a handful of fully utilized threads to handle thousands of concurrent jobs.

Defining Asynchronous Method

- It is common to define an asynchronous method starting with “Begin” and define a pairing method starting with “End”.
- The signatures are as the following (convention):

```
IAsyncResult BeginXXX (in/ref_args,
                       AsyncCallback callback,
                       object state);

return_type EndXXX (out/ref_args,
                    IAsyncResult asyncResult);
```

- The `AsyncCallback` is a delegate that represents a callback method.

Defining Asynchronous Method

- The `BeginXXX` method returns a reference for an `IAsyncResult` object. This is the same reference that should be passed over to the `EndXXX` method.
- The last argument passed over to `BeginXXX` can be accessed within the call back method by referring the `AsyncState` property of the `IAsyncResult` object its reference is passed over to that callback method.

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Defining Asynchronous Method

- Similarly to asynchronous delegates, the `EndXXX` method allows the returned value to be retrieved as well as any `out`/`ref` arguments.

Defining Asynchronous Method

- Object of `AsyncCallback` type references a method to be called when the corresponding asynchronous operation completes.

```
IAsyncResult BeginXXX (in/ref_args,
                       AsyncCallback callback,
                       object state);
```

- It is common to call the `EndXXX` method from within the callback method.

Defining Asynchronous Method

```
public IAsyncResult BeginRead (  byte[] buffer,
                                 int offset,
                                 int size,
                                 AsyncCallback callback,
                                 object state);

public int EndRead (IAsyncResult asyncResult);
```

Asynchronous Delegates

- Asynchronous methods are very similar to asynchronous delegates.
- Unlike asynchronous delegates that might block for any length of time, calling an asynchronous method rarely ever blocks a thread.
- Calling the asynchronous method's Begin method might not return back immediately to the caller. Calling the asynchronous delegate's Begin method returns back immediately.

Asynchronous Delegates

- While the purpose of using asynchronous delegates is to execute a task in parallel with the caller thread, the purpose of asynchronous methods is to allow a big number of tasks to run on few threads.
- While the asynchronous delegate has a built-in support within the execution environment, the asynchronous method doesn't. It is just an agreed protocol.

Asynchronous Delegates

- Calling an asynchronous method from within an asynchronous delegate won't get us anywhere. We will still have multiple threads that aren't exploited efficiently.

Declaring Asynchronous Method

- Declaring an asynchronous method means avoiding the blocking I/O methods altogether and calling their asynchronous counterparts instead.

Without Asynchronous Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;

namespace ConsoleApplication1
{
    class Program
    {
        public static void Main(string[] args)
        {
            ThreadPool.SetMinThreads(50, 50);
            IPAddress address = new IPAddress(new byte[] { 127, 0, 0, 1 });
            TcpListener listener = new TcpListener(address, 1400);
            listener.Start();
            while (true)
            {
                TcpClient client = listener.AcceptTcpClient();
                ThreadPool.QueueUserWorkItem(HandleRequest, client);
                //...
            }
        }
```

Without Asynchronous Method

```
        public static void HandleRequest(object ob)
        {
            using (TcpClient client = (TcpClient)ob)
            {
                using (NetworkStream ns = client.GetStream())
                {
                    byte[] vec =new byte[10000];
                    int temp = ns.Read(vec,0,10000); // BLOCK
                    //...
                }
            }
        }
    }
}
```

Using Asynchronous Method

- In order to scale to a big number of concurrent requests without increasing the number of threads we should employ the asynchronous method pattern.
- Implementing the asynchronous method pattern means avoiding the methods that block the I/O.

Using Asynchronous Method

```
class Program
{
    public static void Main(string[] args)
    {
        ThreadPool.SetMinThreads(50, 50);
        IPAddress address = new IPAddress(new byte[] { 127, 0, 0, 1 });
        TcpListener listener = new TcpListener(address, 1400);
        listener.Start();
        while (true)
        {
            TcpClient client = listener.AcceptTcpClient();
            ThreadPool.QueueUserWorkItem(HandleRequest, client);
            //...
        }
    }
    public static void HandleRequest(object ob)
    {
        new Handler().StartHandling((TcpClient)ob);
    }
}
```

Using Asynchronous Method

```
class Handler
{
    volatile TcpClient client;
    volatile NetworkStream stream;
    byte[] vec = new byte[10000];
    volatile int bytesRead = 0;

    internal void StartHandling(TcpClient c)
    {
        try
        {
            client = c;
            stream = c.GetStream();
            Read();
        }
        catch (Exception ex) {//..}
    }

    void Read()  // NON BLOCKING READ
    {
        stream.BeginRead(vec,bytesRead,vec.Length-
            bytesRead,CallBackFunction,null);
    }
```

Using Asynchronous Method

```
    void CallBackFunction(IAsyncResult r)
    {
        try
        {
            int chunkSize = stream.EndRead(r);
            //...
        }
        //...
        //cleaning up streams we used
        //...
        return;
    }
}
```

Using Asynchronous Method

- Each client request is processed without calling any blocking method.

  The only method that might block a little is the `AcceptTcpClient` method. That method might block for a short time when there aren't any clients reuqest. We cannot avoid this small block.

Using Asynchronous Method

- Developing asynchronous methods we are limited. We cannot use every streaming type.

  Not every class offers asynchronous versions for its methods. Most-likely we will find ourselves limited working with lower level classes than what we would have hoped to work with.

Sample

client

```
namespace client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("client start");
            Thread[] threads = new Thread[40];
            ConnectServer();
        }
        static void ConnectServer()
        {
            String temp = "abcdefghijklmnopqrstuvwxyz0123456789";
            String str = "";
            for (int i = 0; i < 10000; i++)
            {
                str += temp;
            }
```

explanation

code demo

Sample

```
            for (int i = 0; i < 4000; i++)
            {
                using (TcpClient client = new TcpClient("127.0.0.1", 1300))
                {
                    using (NetworkStream stream = client.GetStream())
                    {
                        BinaryWriter writer = new BinaryWriter(stream);
                        writer.Write(str.ToCharArray());
                    }
                }
            }
        }
    }
}
```

Sample

server

```
namespace ConsoleApplication20
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("server start");
            new Server().Start(new IPAddress(new byte[] { 127, 0, 0, 1 }), 1300);
        }
    }

    public class Server
    {
        private static int index = 1;
        public void Start(IPAddress address, int port)
        {
            ThreadPool.SetMinThreads(20, 20);
            ThreadPool.SetMaxThreads(20, 20);
            TcpListener listener = new TcpListener(address, port);
            listener.Start();
            long start = 0, end = 0;
```

Sample

```
    for(int i=0; i<4000; i++)
    {
        TcpClient c = listener.AcceptTcpClient();
        Thread.Sleep(1);
        if (i % 1000 == 0) Console.WriteLine("i=" + i);
        if (i == 0)
        {
            start = DateTime.Now.Ticks;
            Console.WriteLine("start=" + start);
        }
        ThreadPool.QueueUserWorkItem(AsynchronousHandle, c);
    }
    end = DateTime.Now.Ticks;
    Console.WriteLine("end=" + end);
    Console.WriteLine("time="+ (end - start));
}
```

Sample

```
void SimpleHandle(object clientObject)
{
    using (TcpClient client = (TcpClient)clientObject)
    {
        using (NetworkStream stream = client.GetStream())
        {
            byte[] data = new byte[360000];
            stream.Read(data, 0, 360000); // blocking
            Array.Sort(data);
            FileStream fs = new FileStream("__simple_"+ (++index)
                + DateTime.Now.Ticks + ".txt", FileMode.OpenOrCreate);
            fs.Write(data,0,data.Length);
        }
    }
}
void AsynchronousHandle(object clientObject)
{
    using (TcpClient client = (TcpClient)clientObject)
    {
        using (NetworkStream stream = client.GetStream())
        {
            byte[] data = new byte[360000];
            stream.BeginRead(data, 0, 360000, ReadCallBack, data);
        }
    }
}
```

Sample

```
        void ReadCallBack(IAsyncResult result)
        {
            Array.Sort((byte[])result.AsyncState);
            FileStream fs = new FileStream("__asynch_" + (++index)
                + DateTime.Now.Ticks + ".txt", FileMode.OpenOrCreate);
            byte[] data = (byte[])result.AsyncState;
            fs.BeginWrite(data, 0, data.Length, null, null);
        }
    }
}
```

Asynchronous Methods

Introduction

- Some of the classes in the .NET Framework Base Class Library (BCL) provide both synchronous and asynchronous method signatures.
- Calling a synchronous method can create a delay in our program flow.
- Calling an asynchronous method might result in better performance in cases such as when a program needs to send out requests to multiple Web services.

Using the asynchronous pattern doesn't guarantee each one of the executions to execute parallel with the caller. If there is a need in a true parallel execution we better avoid using asynchronous method and prefer using alternatives, such as using asynchronous delegates or using the BackgroundWorker class.

Synchronous vs Asynchronous

- When calling a synchronous method the execution waits for the method to complete its execution before moving forward.
- When calling an asynchronous method the execution returns immediately so that the program can perform other operations while the called method completes its work.

Using the asynchronous pattern doesn't guarantee each one of the executions to execute parallel with the caller. If there is a need in a true parallel execution we better avoid using asynchronous method and prefer using alternatives, such as using asynchronous delegates or using the BackgroundWorker class.

Multiple Threads Problem

- When having a code that needs to be executed concurrently, assigning a dedicated thread for each one of the executions might consume a huge amount of memory.
- In most cases, each one of the threads isn't always busy. In most cases each thread consumes a fraction of the time it was allocated with.

Using the asynchronous pattern doesn't guarantee each one of the executions to execute parallel with the caller. If there is a need in a true parallel execution we better avoid using asynchronous method and prefer using alternatives, such as using asynchronous delegates or using the BackgroundWorker class.

The Asynchronous Method Pattern

- The asynchronous method pattern allows a handful of fully utilized threads to take on thousands of concurrent executions.
- If each and every thread is busy during its entire execution the asynchronous method pattern wouldn't have been relevant.

Using the asynchronous pattern doesn't guarantee each one of the executions to execute parallel with the caller. If there is a need in a true parallel execution we better avoid using asynchronous method and prefer using alternatives, such as using asynchronous delegates or using the BackgroundWorker class.

Optimizing Threads Resources

- The purpose of using asynchronous methods address isn't to provide a convenient mechanism for executing methods concurrently.
- We use asynchronous methods in order to optimize thread resources.
- The asynchronous method aims at getting a situation in which none of the threads is blocked. This way, all threads will be exploited to their maximum potential.

Declaring an asynchronous method we should abstain from calling any blocking method. An asynchronous method aims never to block any thread.

The Web Request Case Study

- When having a thread dedicated to processing a single web request we might find it spending 99 percent of its time blocked... waiting for the server to return its reply.
- The asynchronous method pattern exploits this potential allowing a handful of fully utilized threads to handle thousands of concurrent jobs.

Declaring an asynchronous method we should abstain from calling any blocking method. An asynchronous method aims never to block any thread.

Defining Asynchronous Method

- It is common to define an asynchronous method starting with “Begin” and define a pairing method starting with “End”.
- The signatures are as the following (convention):

```
IAsyncResult BeginXXX (in/ref_args,
                       AsyncCallback callback,
                       object state);

return_type EndXXX (out/ref_args,
                    IAsyncResult asyncResult);
```

- The `AsyncCallback` is a delegate that represents a callback method.

Defining Asynchronous Method

- The `BeginXXX` method returns a reference for an `IAsyncResult` object. This is the same reference that should be passed over to the `EndXXX` method.
- The last argument passed over to `BeginXXX` can be accessed within the call back method by referring the `AsyncState` property of the `IAsyncResult` object its reference is passed over to that callback method.

```
public delegate void AsyncCallback (IAsyncResult ar);
```


Defining Asynchronous Method

- Similarly to asynchronous delegates, the `EndXXX` method allows the returned value to be retrieved as well as any `out`/`ref` arguments.

Defining Asynchronous Method

- Object of `AsyncCallback` type references a method to be called when the corresponding asynchronous operation completes.

```
IAsyncResult BeginXXX (in/ref_args,
                       AsyncCallback callback,
                       object state);
```

- It is common to call the `EndXXX` method from within the callback method.

Defining Asynchronous Method

```
public IAsyncResult BeginRead (   byte[] buffer,
                                  int offset,
                                  int size,
                                  AsyncCallback callback,
                                  object state);

public int EndRead (IAsyncResult asyncResult);
```


Asynchronous Delegates

- Asynchronous methods are very similar to asynchronous delegates.
- Unlike asynchronous delegates that might block for any length of time, calling an asynchronous method rarely ever blocks a thread.
- Calling the asynchronous method's Begin method might not return back immediately to the caller. Calling the asynchronous delegate's Begin method returns back immediately.

Asynchronous Delegates

- While the purpose of using asynchronous delegates is to execute a task in parallel with the caller thread, the purpose of asynchronous methods is to allow a big number of tasks to run on few threads.
- While the asynchronous delegate has a built-in support within the execution environment, the asynchronous method doesn't. It is just an agreed protocol.

Asynchronous Delegates

- Calling an asynchronous method from within an asynchronous delegate won't get us anywhere. We will still have multiple threads that aren't exploited efficiently.

Declaring Asynchronous Method

- Declaring an asynchronous method means avoiding the blocking I/O methods altogether and calling their asynchronous counterparts instead.

Without Asynchronous Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;

namespace ConsoleApplication1
{
    class Program
    {
        public static void Main(string[] args)
        {
            ThreadPool.SetMinThreads(50, 50);
            IPAddress address = new IPAddress(new byte[] { 127, 0, 0, 1 });
            TcpListener listener = new TcpListener(address, 1400);
            listener.Start();
            while (true)
            {
                TcpClient client = listener.AcceptTcpClient();
                ThreadPool.QueueUserWorkItem(HandleRequest, client);
                //...
            }
        }
```


Without Asynchronous Method

```
        public static void HandleRequest(object ob)
        {
            using (TcpClient client = (TcpClient)ob)
            {
                using (NetworkStream ns = client.GetStream())
                {
                    byte[] vec =new byte[10000];
                    int temp = ns.Read(vec,0,10000); // BLOCK
                    //...
                }
            }
        }
    }
}
```


Using Asynchronous Method

- In order to scale to a big number of concurrent requests without increasing the number of threads we should employ the asynchronous method pattern.
- Implementing the asynchronous method pattern means avoiding the methods that block the I/O.

Using Asynchronous Method

```
class Program
{
    public static void Main(string[] args)
    {
        ThreadPool.SetMinThreads(50, 50);
        IPAddress address = new IPAddress(new byte[] { 127, 0, 0, 1 });
        TcpListener listener = new TcpListener(address, 1400);
        listener.Start();
        while (true)
        {
            TcpClient client = listener.AcceptTcpClient();
            ThreadPool.QueueUserWorkItem(HandleRequest, client);
            //...
        }
    }
    public static void HandleRequest(object ob)
    {
        new Handler().StartHandling((TcpClient)ob);
    }
}
```


Using Asynchronous Method

```
class Handler
{
    volatile TcpClient client;
    volatile NetworkStream stream;
    byte[] vec = new byte[10000];
    volatile int bytesRead = 0;

    internal void StartHandling(TcpClient c)
    {
        try
        {
            client = c;
            stream = c.GetStream();
            Read();
        }
        catch (Exception ex) {//..}
    }

    void Read()  // NON BLOCKING READ
    {
        stream.BeginRead(vec,bytesRead,vec.Length-
            bytesRead,CallBackFunction,null);
    }
```


Using Asynchronous Method

```
    void CallBackFunction(IAsyncResult r)
    {
        try
        {
            int chunkSize = stream.EndRead(r);
            //...
        }
        //...
        //cleaning up streams we used
        //...
        return;
    }
}
```


Using Asynchronous Method

- Each client request is processed without calling any blocking method.

  The only method that might block a little is the `AcceptTcpClient` method. That method might block for a short time when there aren't any clients reuqest. We cannot avoid this small block.

Using Asynchronous Method

- Developing asynchronous methods we are limited. We cannot use every streaming type.

  Not every class offers asynchronous versions for its methods. Most-likely we will find ourselves limited working with lower level classes than what we would have hoped to work with.

Sample

client

```
namespace client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("client start");
            Thread[] threads = new Thread[40];
            ConnectServer();
        }
        static void ConnectServer()
        {
            String temp = "abcdefghijklmnopqrstuvwxyz0123456789";
            String str = "";
            for (int i = 0; i < 10000; i++)
            {
                str += temp;
            }
```

explanation

code demo

Sample

```
            for (int i = 0; i < 4000; i++)
            {
                using (TcpClient client = new TcpClient("127.0.0.1", 1300))
                {
                    using (NetworkStream stream = client.GetStream())
                    {
                        BinaryWriter writer = new BinaryWriter(stream);
                        writer.Write(str.ToCharArray());
                    }
                }
            }
        }
    }
}
```


Sample

server

```
namespace ConsoleApplication20
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("server start");
            new Server().Start(new IPAddress(new byte[] { 127, 0, 0, 1 }), 1300);
        }
    }

    public class Server
    {
        private static int index = 1;
        public void Start(IPAddress address, int port)
        {
            ThreadPool.SetMinThreads(20, 20);
            ThreadPool.SetMaxThreads(20, 20);
            TcpListener listener = new TcpListener(address, port);
            listener.Start();
            long start = 0, end = 0;
```


Sample

```
        for(int i=0; i<4000; i++)
        {
            TcpClient c = listener.AcceptTcpClient();
            Thread.Sleep(1);
            if (i % 1000 == 0) Console.WriteLine("i=" + i);
            if (i == 0)
            {
                start = DateTime.Now.Ticks;
                Console.WriteLine("start=" + start);
            }
            ThreadPool.QueueUserWorkItem(AsynchronousHandle, c);
        }
        end = DateTime.Now.Ticks;
        Console.WriteLine("end=" + end);
        Console.WriteLine("time="+ (end - start));
    }
```


Sample

```
void SimpleHandle(object clientObject)
{
    using (TcpClient client = (TcpClient)clientObject)
    {
        using (NetworkStream stream = client.GetStream())
        {
            byte[] data = new byte[360000];
            stream.Read(data, 0, 360000); // blocking
            Array.Sort(data);
            FileStream fs = new FileStream("__simple_"+ (++index)
                + DateTime.Now.Ticks + ".txt", FileMode.OpenOrCreate);
            fs.Write(data,0,data.Length);
        }
    }
}
void AsynchronousHandle(object clientObject)
{
    using (TcpClient client = (TcpClient)clientObject)
    {
        using (NetworkStream stream = client.GetStream())
        {
            byte[] data = new byte[360000];
            stream.BeginRead(data, 0, 360000, ReadCallBack, data);
        }
    }
}
```


Sample

```
        void ReadCallBack(IAsyncResult result)
        {
            Array.Sort((byte[])result.AsyncState);
            FileStream fs = new FileStream("__asynch_" + (++index)
                + DateTime.Now.Ticks + ".txt", FileMode.OpenOrCreate);
            byte[] data = (byte[])result.AsyncState;
            fs.BeginWrite(data, 0, data.Length, null, null);
        }
    }
}
```


