Multithreading in .NET

Multithreading allows you to increase the responsiveness of your application and, if your application runs on a multiprocessor or multi-core system, increase its throughput.

Processes and threads

A *process* is an executing program. An operating system uses processes to separate the applications that are being executed. A *thread* is the basic unit to which an operating system allocates processor time. Each thread has a scheduling priority and maintains a set of structures the system uses to save the thread context when the thread's execution is paused. The thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack. Multiple threads can run in the context of a process. All threads of a process share its virtual address space. A thread can execute any part of the program code, including parts currently being executed by another thread.

By default, a .NET program is started with a single thread, often called the *primary* thread. However, it can create additional threads to execute code in parallel or concurrently with the primary thread. These threads are often called *worker* threads.

When to use multiple threads?

You use multiple threads to increase the responsiveness of your application and to take advantage of a multiprocessor or multi-core system to increase the application's throughput.

Consider a desktop application, in which the primary thread is responsible for user interface elements and responds to user actions. Use worker threads to perform time-consuming operations that, otherwise, would occupy the primary thread and make the user interface non-responsive. You can also use a dedicated thread for network or device communication to be more responsive to incoming messages or events.

If your program performs operations that can be done in parallel, the total execution time can be decreased by performing those operations in separate threads and running the program on a multiprocessor or multi-core system. On such a system, use of multithreading might increase throughput along with the increased responsiveness.

How to use multithreading in .NET?

Starting with .NET Framework 4, the recommended way to utilize multithreading is to use Task Parallel Library (TPL) and Parallel LINQ (PLINQ). Both TPL and PLINQ rely on the ThreadPool threads. The System.Threading.ThreadPool class provides a .NET application with a pool of worker threads. You can also use thread pool threads.

At last, you can use the System.Threading.Thread class that represents a managed thread. Multiple threads might need to access a shared resource. To keep the resource in an uncorrupted state and avoid race conditions, you must synchronize the thread access to it. You also might want to coordinate the

interaction of multiple threads. .NET provides a range of types that you can use to synchronize access to a shared resource or coordinate thread interaction.

Thread Class

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.

Thread class is used to create and control a thread, to set its priority, and to get its status. You can retrieve a number of property values that provide information about a thread. In some cases, you can also set these property values to control the operation of the thread. These thread properties include: `Name`, `Priority`, `ThreadState`, `ManagedThreadId`, `IsThreadPoolThread`, `IsBackground`, `IsAlive`, `CurrentThread`, `ExecutionContext`, `CurrentCulture` etc.

Following constructors are provided by `Thread` class:

1. `Thread(ParameterizedThreadStart)`
- Initializes a new instance of the Thread class, specifying a delegate that allows an object to be passed to the thread when the thread is started.
2. `Thread(ParameterizedThreadStart, Int32)`
- Initializes a new instance of the Thread class, specifying a delegate that allows an object to be passed to the thread when the thread is started and specifying the maximum stack size for the thread.
3. `Thread(ThreadStart)`
- Initializes a new instance of the Thread class.
4. `Thread(ThreadStart, Int32)`
- Initializes a new instance of the Thread class, specifying the maximum stack size for the thread.

The `Thread` class constructor which takes one parameter is either of the type `ThreadStart` or `ParameterizedThreadStart`. These types are delegate types defined in System.Threading namespace.

```
public delegate void ThreadStart()
```

```
public delegate void ParameterizedThreadStart(object obj)
```

The `ThreadStart` delegate does not take any parameter as well as the return type is `void`. `ParameterizedThreadStart` delegate is taking one parameter of `object` type and it also does not return any value.

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

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.

Creating and Starting a thread

We can start a thread by supplying a delegate that represents the method the thread is to execute in its class constructor. You then call the Start method to begin execution.

Example:

```
using System;
using System.Diagnostics;
using System.Threading;
public class Example
{
   public static void Main()
   {
      // To create a thread, create an instance of Thread class and pass the
      // method name to its constructor.
      // If the method needs an argument then it will be considered as
      // ParameterizedThreadStart delegate type and its argument is passed in
      // Start() method.
      var th = new Thread(ExecuteInForeground);
      th.Start(2500);
      Thread.Sleep(1000);
      Console.WriteLine("Main thread ({0}) exiting...",
                        Thread.CurrentThread.ManagedThreadId);
   }
   private static void ExecuteInForeground(Object obj)
   {
      int interval;
      try {
         interval = (int) obj;
      }
      catch (InvalidCastException) {
         interval = 5000;
      }
      var sw = Stopwatch.StartNew();
      Console.WriteLine("Thread {0}: {1}, Priority {2}",
                        Thread.CurrentThread.ManagedThreadId,
                        Thread.CurrentThread.ThreadState,
                        Thread.CurrentThread.Priority);
      do {
         Console.WriteLine("Thread {0}: Elapsed {1:N2} seconds",
                           Thread.CurrentThread.ManagedThreadId,
                           sw.ElapsedMilliseconds / 1000.0);
         Thread.Sleep(500);
      } while (sw.ElapsedMilliseconds <= interval);
      sw.Stop();
   }
}
// The example displays output like the following:
```

```
//       Thread 3: Running, Priority Normal
//       Thread 3: Elapsed 0.00 seconds
//       Thread 3: Elapsed 0.52 seconds
//       Main thread (1) exiting...
//       Thread 3: Elapsed 1.03 seconds
//       Thread 3: Elapsed 1.55 seconds
//       Thread 3: Elapsed 2.06 seconds
```

Retrieving Thread objects

You can use the static `CurrentThread` property to retrieve a reference to the currently executing thread from the code that the thread is executing. The following example uses the `CurrentThread` property to display information about the main application thread, another foreground thread, a background thread, and a thread pool thread.

Example:

```
using System;
using System.Threading;
public class Example
{
   static Object obj = new Object();
   public static void Main()
   {
      // Thread pool in C# is a collection of threads that can be
      // reused to perform number of tasks in the background.
      ThreadPool.QueueUserWorkItem(ShowThreadInformation);
      var th1 = new Thread(ShowThreadInformation);
      th1.Start();
      var th2 = new Thread(ShowThreadInformation);
      th2.IsBackground = true;
      th2.Start();
      Thread.Sleep(500);
      ShowThreadInformation(null);
   }
   private static void ShowThreadInformation(Object state)
   {
      // as multiple threads share same method, shared variables need to be
      // locked before thread starts its execution
      lock (obj) {
         var th  = Thread.CurrentThread;
         Console.WriteLine("Managed thread #{0}: ", th.ManagedThreadId);
         Console.WriteLine("   Background thread: {0}", th.IsBackground);
         Console.WriteLine("   Thread pool thread: {0}", th.IsThreadPoolThread);
         Console.WriteLine("   Priority: {0}", th.Priority);
         Console.Write("   Culture: {0}, ", th.CurrentCulture.Name);
         Console.WriteLine("   UI culture: {0}", th.CurrentUICulture.Name);
         Console.WriteLine();
      }
   }
}
// The example displays output like the following:
```

```
//       Managed thread #6:
//          Background thread: True
//          Thread pool thread: False
//          Priority: Normal
//          Culture: en-US, UI culture: en-US
//
//       Managed thread #3:
//          Background thread: True
//          Thread pool thread: True
//          Priority: Normal
//          Culture: en-US, UI culture: en-US
//
//       Managed thread #4:
//          Background thread: False
//          Thread pool thread: False
//          Priority: Normal
//          Culture: en-US, UI culture: en-US
//
//       Managed thread #1:
//          Background thread: False
//          Thread pool thread: False
//          Priority: Normal
//          Culture: en-US, UI culture: en-US
```

Foreground and background threads

Instances of the Thread class represent either foreground threads or background threads. Background threads are identical to foreground threads with one exception: a background thread does not keep a process running if all foreground threads have terminated. Once all foreground threads have been stopped, the runtime stops all background threads and shuts down.

By default, the following threads execute in the **foreground**:

- The **main** application thread.
- All threads created by calling a **Thread** class constructor.

The following threads execute in the **background** by default:

- Thread pool threads, which are a pool of worker threads maintained by the runtime. You can configure the thread pool and schedule work on thread pool threads by using the **ThreadPool** class.
- All threads that enter the managed execution environment from unmanaged code.

You can change a thread to execute in the background by setting the **`IsBackground`** property at any time. Background threads are useful for any operation that should continue as long as an application is running but should not prevent the application from terminating, such as monitoring file system changes or incoming socket connections.