Software Development Technologies

Threads and Synchronization in C#

Muhammad Rizwan

rairizwanali@gmail.com

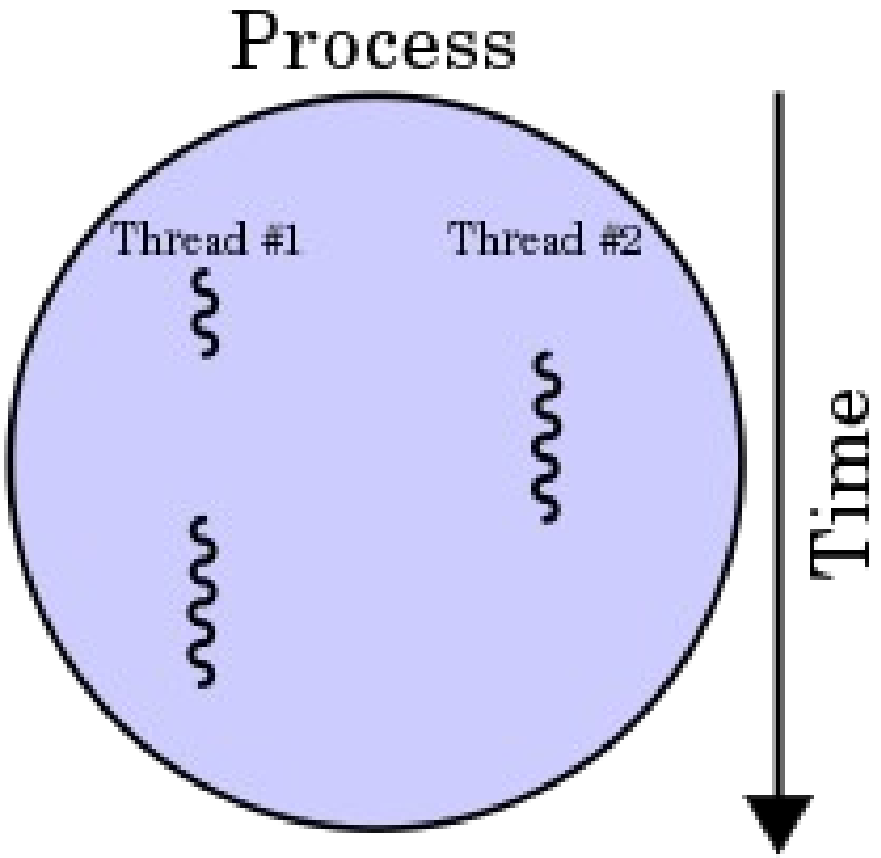
Contents

- What is a Thread?
- Threads in C#
- Types of Threads
- Threads Priority in C#
- Controlling Threads in C#

What is Thread ?

- In computer science, a **Thread** is the smallest unit of processing that can be scheduled by an operating system.
- It generally results from a fork of a computer program.
- The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is **contained** inside a **process**.
- Multiple threads can exist within the same process and share resources such as **memory**, while different processes do not share these resources.
- Consider the Example of a Word Processing Application i.e., Ms Word. **Ms Word** is a process and **spell-checker** within it is a Thread. And the memory they share is Word document.

What is Thread ?



Threads in C#

- A thread is an **independent** stream of **instructions** in a program.
- All your C# programs up to this point have one entry point — the **Main()** method.
- Execution starts with the first statement in the Main() method and continues until that method returns.
- This program structure is all very well for programs in which there is one **identifiable sequence** of **tasks** With the Thread class, you can create and control threads.

Threads in C#

- The code here is a very simple example of creating and starting a new thread.
- The constructor of the Thread class is overloaded to accept a delegate parameter of type **ThreadStart**
- OR, a simple method without any return value and input parameters.
- The **ThreadStart** delegate defines a method with a void return type and without arguments.
- After the Thread object is created, you can start the thread with the Start() method.

Threads in C#

```
using System;
using System.Threading;
namespace Wrox.ProCSharp.Threading
{
        class Program
        {
                static void Main()
                {
                        Thread  t1 = new Thread(ThreadMain);
                        t1.Start();
                        Console.WriteLine("This is the main thread.");
                }
                static void ThreadMain()
                {
                        Console.WriteLine("Running in a thread.");
                }
        }
}
```

Threads in C#

- When you run the application, you get the output of the two threads:

  This is the main thread.

  Running in a thread.

Types of Threads

- There are two types of Threads
  - Foreground Threads
  - Background Threads
- The **process** of the application keeps running as long as at least one **foreground** thread is running.
- **Even** if **Main()** method **ends**, the **process** of the application **remains active** until all foreground threads finish their work.
- A thread you create with the Thread class, **by default**, is a **foreground thread**.

Threads in C#

- When you create a thread with the **Thread class**, you can define whether it should be a foreground or background thread by setting the property ***IsBackground***.
- Background threads are very useful for background tasks.
- **For example**, when you close the Word application, it doesn't make sense for the spell checker to keep its process running.
- The spell-checker thread can be killed when the application is closed (Background Thread).
- However, the thread organizing the Outlook message store should remain active until it is finished, even if Outlook is closed (Foreground Thread).

Threads Priority in C#

- The operating system schedules threads based on a priority, and the thread with the highest priority is scheduled to run in the CPU.
- With the Thread class, you can influence the priority of the thread by setting the *Priority* property.
- The Priority property requires a value that is defined by the ***ThreadPriority*** **enumeration**.
- The levels defined are ***Highest*** , ***AboveNormal*** , ***Normal*** , ***BelowNormal*** , and ***Lowest*** .

Controlling Threads

- The thread is invoked by the *Start*() method of a Thread object.
- However, after invoking the *Start*() method, the new thread is still not in the Running state, but in the ***Unstarted*** state.
- The thread changes to the Running state as soon as the operating system thread scheduler selects the thread to run.
- You can read the current state of a thread by reading the property ***Thread.ThreadState***.
- With the ***Thread.Sleep*()** method, a thread goes into the *WaitSleepJoin* state.
- And waits until it is woken up again after the time span defined with the *Sleep*() method has elapsed.

Controlling Threads

- To stop another thread, you can invoke the method *Thread.Abort*().
- When this method is called, an exception of type ***ThreadAbortException*** is thrown in the thread that receives the abort.
- With a handler to catch this exception, the thread can do some clean-up before it ends.

Thread Synchronization

- A race condition can occur if two or more threads access the shared data in the absence of synchronization.
- It is best to avoid synchronization issues by not sharing data between threads. Of course, this is not always possible.
- If data sharing is necessary, you must use synchronization techniques so that only one thread at a time accesses and changes shared state.

Thread Synchronization

```
public  class myClass
{
    public  static int count;
    public void A()
    {
      for (int i = 0; i <100; i++)
      {
        count++;
      }
    }
  }
```

Thread Synchronization

```
public class myProgram
{
        Thread T1 = new Thread(A);
        Thread T2 = new Thread(A);
        T1.Start();
        T2.Start();

        Console.WriteLine(“ The count variable = {0}”,myClass.count);
}
```

Thread Synchronization

- These two threads run as two independent threads.
- If you run this program then you will discover that the final value of count isn't predictable because it all depends on when the two threads get access to count.
- Every time you run this program the last value of count variable would be different.

Thread Synchronization

- C# has its own keyword for the synchronization of multiple threads: the lock statement.
- The lock statement is an easy way to hold for a lock and release it.

```
public void A()
  {
       lock (this)
       {
         for (int i = 0; i <100; i++)
        {
          count++;
        }
       }
  }
```

The End

- Thanks for listening
- Questions would be appreciated.