Post ▼ Ask Question

Introduction To Multithreading In C#

Mike Gold Updated date Mar 25, 2019 1.1m 38 30

multithreading.ZIP

In working in New York and talking to programmers all over Wall Street, I've noticed a common thread of knowledge expected in most real time programming applications. That knowledge is known as multithreading. As I have migrated around the programming world, and performed interviews on potential programming candidates, it never ceases to amaze me how little is known about multithreading or why or how threading is applied. In a series of excellent articles written by Vance Morrison, MSDN has tried to address this problem: (See the August Issue of MSDN, What every Developer Must Know about Multithreaded Apps, and the October issue Understand the Impact of Low-Lock Techniques in Multithreaded Apps.

What is a thread?

Every application runs with at least one thread. So what is a thread? A thread is nothing more than a process. My guess is that the word thread comes from the Greek mythology of supernatural Muses weaving threads on a loom, where each thread represents a path in time of someone's life. If you mess with that thread, then you disturb the fabric of life or change the process of life. On the computer, a thread is a process moving through time. The process performs sets of sequential steps, each step executing a line of code. Because the steps are sequential, each step takes a given amount of time. The time it takes to complete a series of steps is the sum of the time it takes to perform each programming step.

What are multithreaded applications?

For a long time, most programming applications (except for embedded system programs) were single-threaded. That means there was only one thread in the entire application. You could never do computation A until completing computation B. A program starts at step 1 and continues sequentially (step 2, step 3, step 4) until it hits the final step (call it step 10). A multithreaded application allows you to run several threads, each thread running in its own process. So theoretically you can run step 1 in one thread and at the same time run step 2 in another thread. At the same time you could run step 3 in its own thread, and even step 4 in its own thread. Hence step 1, step 2, step 3, and step 4 would run concurrently. Theoretically, if all four steps took about the same time, you could finish your program in a quarter of the time it takes to run a single thread (assuming you had a 4 processor machine). So why isn't every program multithreaded? Because along with speed, you face complexity. Imagine if step 1 somehow depends on the information in step 2. The program might not run correctly if step 1 finishes calculating before step 2 or visa versa.

An Unusual Analogy

Another way to think of multiple threading is to consider the human body. Each one of the body's organs (heart, lungs, liver, brain) are all involved in processes. Each process is running simultaneously. Imagine if each organ ran as a step in a process: first the heart, then the brain, then the liver, then the lungs. We would probably drop dead. So the human body is like one big multithreaded application. All organs are processes running simultaneously, and all of these processes depend upon one another. All of these processes communi through nerve signals, blood flow and chemical triggers. As with all multithreaded applications, the human body is very complex. If s

When to Thread

Multiple threading is most often used in situations where you want programs to run more efficiently. For example, let's say your Window Form program contains a method (call it method_A) inside it that takes more than a second to run and needs to run repetitively. Well, if the entire program ran in a single thread, you would notice times when button presses didn't work correctly, or your typing was a bit sluggish. If method_A was computationally intensive enough, you might even notice certain parts of your Window Form not working at all. This unacceptable program behavior is a sure sign that you need multithreading in your program. Another common scenario where you would need threading is in a messaging system. If you have numerous messages being sent into your application, you need to capture them at the same time your main processing program is running and distribute them appropriately. You can't efficiently capture a series of messages at the same time you are doing any heavy processing, because otherwise you may miss messages. Multiple threading can also be used in an assembly line fashion where several processes run simultaneously. For example once process collects data in a thread, one process filters the data, and one process matches the data against a database. Each of these scenarios are common uses for multithreading and will significantly improve performance of similar applications running in a single thread.

When not to Thread

It is possible that when a beginning programmer first learns threading, they may be fascinated with the possibility of using threading in their program. They may actually become *thread-happy.* Let me elaborate,

Day 1) Programmer learns the that they can spawn a thread and begins creating a single new thread in their program, *Cool*!
Day 2) Programmer says, "I can make this even more efficient by spawning other threads in parts of my program!"
Day 3) P: "Wow, I can even fork threads within threads and REALLY improve efficiency!!"
Day 4) P: "I seem to be getting some odd results, but that's okay, we'll just ignore them for now."
Day 5) "Hmmmm, sometimes my widgetX variable has a value, but other times it never seems to get set at all, must be my computer isn't working, I'll just run the debugger".
Day 9) "This darn (stronger language) program is jumping all over the place!! I can't figure out what is going on!"
Week 2) Sometimes the program just sits there and does absolutely nothing! H-E-L-P!!!!!

thing, I'm just pointing out that in the process of creating threading efficiency in your programs, be *very, very careful.* Because unlike a single threaded program, you are handling many processes at the same time, and multiple processes, with multiple dependent variables, can be very tricky to follow. Think of multithreading like you would think of juggling. Juggling a single ball in your hand (although kind of boring) is fairly simple. However, if you are challenged to put two of those balls in the air, the task is a bit more difficult. 3, 4, and 5, balls are progressively more difficult. As the ball count increases, you have a better and better chance of really *dropping the ball.* Juggling a lot of balls at once requires knowledge, skill, and precise timing. So does multiple threading.

Multithreading

Figure 1 - Multithreading is like juggling processes

Problems with Threading

If every process in your program was mutually exclusive - that is, no process depended in any way upon another, then multiple threading would be very easy and very few problems would occur. Each process would run along in its own happy course and not bother the other processes. However, when more than one process needs to read or write the memory used by other processes, problems can occur. For example let's say there are two processes, process #1 and process #2. Both processes share variable X. If thread process #1 writes variable X with the value 5 first and thread process #2 writes variable X with value -3 next, the final value of X is -3. However if process #2 writes variable X with value -3 first and then process #1 writes variable X with value 5, the final value of X is 5. So you see, if the process that allows you to set X has no knowledge of process #1 or process #2, X can end up with different final values depending upon which thread got to X first. In a single threaded program, there is no way this could happen, because everything follows in sequence. In a single threaded program, since no processes are running in parallel, X is always set by method #1 first, (if it is called first) and then set by method #2. There are no surprises in a single threaded program, it's just step by step. With a mulithreaded program, two threads can enter a piece of code at the same time, and wreak havoc on the results. The problem with threads is that you need some way to control one thread accessing a shared piece of memory while another thread running at the same time is allowed to enter the same code and manipulate the shared data.

Thread Safety

program we force one thread to wait inside our code block while the other thread is finishing its business. This activity, known as thread blocking or synchronizing threads, allows us to control the timing of simultaneous threads running inside our program. In C# we lock on a particular part of memory (usually an instance of an object) and don't allow any thread to enter code of this object's memory until another thread is done using the object. By now you are probably thirsting for a code example, so here you go.

Let's take a look at a two-threaded scenario. In our example, we will create two threads in C#: Thread 1 and Thread 2, both running in their own while loop. The threads won't do anything useful, they will just print out a message saying which thread they are part of. We will utilize a shared memory class member called _threadOutput. _threadOutput will be assigned a message based upon the thread in which it is running. Listing #1 shows the two threads contained in DisplayThread1 and DisplayThread2 respectively.

Listing 1 - Creating two threads sharing a common variable in memory

```
// shared memory variable between the two threads
// used to indicate which thread we are in
private string _threadOutput = "";

/// <summary>
/// Thread 1: Loop continuously,
/// Thread 1: Displays that we are in thread 1
/// </summary>
void DisplayThread1()
{
      while (_stopThreads == false)
      {
            Console.WriteLine("Display Thread 1");

            // Assign the shared memory to a message about thread #1
            _threadOutput = "Hello Thread1";


            Thread.Sleep(1000);  // simulate a lot of processing

            // tell the user what thread we are in thread #1, and display shared memory
            Console.WriteLine("Thread 1 Output --> {0}", _threadOutput);
}

/// <summary>
/// Thread 2: Loop continuously,
/// Thread 2: Displays that we are in thread 2
/// </summary>
void DisplayThread2()
{
      while (_stopThreads == false)
      {
        Console.WriteLine("Display Thread 2");


       // Assign the shared memory to a message about thread #2
        _threadOutput = "Hello Thread2";


        Thread.Sleep(1000);  // simulate a lot of processing

       // tell the user we are in thread #2
        Console.WriteLine("Thread 2 Output --> {0}", _threadOutput);

      }
}
Class1()
{
      // construct two threads for our demonstration;
      Thread thread1 = new Thread(new ThreadStart(DisplayThread1));
      Thread thread2 = new Thread(new ThreadStart(DisplayThread2));

      // start them
      thread1.Start();
      thread2.Start();
}
```

The results of this code is shown in figure 2. Look carefully at the results. You will notice that the program gives some surprising outp we were looking at this from a single-threaded mindset). Although we clearly assigned _threadOutput to a string with a number

Threading in C#

Figure 2 - Unusual output from our two thread example.

We would expect to see the following from our code,

Thread 1 Output --> Hello Thread 1 and Thread 2 Output --> Hello Thread 2, but for the most part, the results are completely unpredictable.

Sometimes we see Thread 2 Output --> Hello Thread 1 and Thread 1 Output --> Hello Thread 2. The thread output does not match the code! Even though, we look at the code and follow it with our eyes, _threadOutput = "Hello Thread 2", Sleep, Write "Thread 2 --> Hello Thread 2", this sequence we expect does not necessarily produce the final result.

Explanation

The reason we see the results we do is because in a multithreaded program such as this one, the code theoretically is executing the two methods DisplayThread1 and DisplayThread2, simultaneously. Each method shares the variable, _threadOutput. So it is possible that although _threadOutput is assigned a value "Hello Thread1" in thread #1 and displays _threadOutput two lines later to the console, that somewhere in between the time thread #1 assigns it and displays it, thread #2 assigns _threadOutput the value "Hello Thread2". Not only are these strange results, possible, they are quite frequent as seen in the output shown in figure 2. This painful threading problem is an all too common bug in thread programming known as a *race condition.* This example is a very simple example of the well-known threading problem. It is possible for this problem to be hidden from the programmer much more indirectly such as through referenced variables or collections pointing to thread-unsafe variables. Although in figure 2 the symptoms are blatant, a race condition can appear much more rarely and be intermittent once a minute, once an hour, or appear three days later. The race is probably the programmer's worst nightmare because of its infrequency and because it can be *very very* hard to reproduce.

Winning the Race

The best way to avoid race conditions is to write thread-safe code. If your code is thread-safe, you can prevent some nasty threading issues from cropping up. There are several defenses for writing thread-safe code. One is to share memory as little as possible. If you their own fields, hence no shared memory. If you do have static variables in your class or the instance of your class is shared by several other threads, then you must find a way to make sure one thread cannot use the memory of that variable until the other class is done using it. The way we prevent one thread from affecting the memory of the other class while one is occupied with that memory is called *locking.* C# allows us to lock our code with either a Monitor class or a lock { } construct. (The lock construct actually internally implements the Monitor class through a try-finally block, but it hides these details from the programmer). In our example in listing 1, we can lock the sections of code from the point in which we populate the shared _threadOutput variable all the way to the actual output to the console. We lock our critical section of code in both threads so we don't have a race in one or the other. The quickest and dirtiest way to lock inside a method is to lock on this pointer. Locking on this pointer will lock on the entire class instance, so any thread trying to modify a field of the class while inside the lock will be *blocked.* Blocking means that the thread trying to change the variable will sit and wait until the lock is released on the locked thread. The thread is released from the lock upon reaching the last bracket in the lock { } construct.

Listing 2 - Synchronizing two Threads by locking them

```
/// <summary>
/// Thread 1, Displays that we are in thread 1 (locked)
 /// </summary>
 void DisplayThread1()
 {
       while (_stopThreads == false)
       {
          // lock on the current instance of the class for thread #1
             lock (this)
             {
                   Console.WriteLine("Display Thread 1");
                   _threadOutput = "Hello Thread1";
                   Thread.Sleep(1000);  // simulate a lot of processing
                   // tell the user what thread we are in thread #1
                   Console.WriteLine("Thread 1 Output --> {0}", _threadOutput);
             }// lock released  for thread #1 here
       }
 }

/// <summary>
```

```
void DisplayThread2()
{
      while (_stopThreads == false)
      {

          // lock on the current instance of the class for thread #2
            lock (this)
            {
                  Console.WriteLine("Display Thread 2");
                  _threadOutput = "Hello Thread2";
                  Thread.Sleep(1000);  // simulate a lot of processing
                  // tell the user what thread we are in thread #1
                  Console.WriteLine("Thread 2 Output --> {0}", _threadOutput);
            } // lock released  for thread #2 here
      }
}
```

The results of locking the two threads is shown in figure 3. Note that all thread output is nicely synchronized. You always get a result saying Thread 1 Output --> Hello Thread 1   and Thread 2 Output --> Hello Thread 2. Note, however, that thread locking does come at a price. When you lock a thread, you force the other thread to wait until the lock is released. In essence, you've slowed down the program, because while the other thread is waiting to use the shared memory, the first thread isn't doing anything in the program. Therefore you need to use locks sparingly; don't just go and lock every method you have in your code if they are not involved in shared memory. Also be careful when you use locks, because you don't want to get into the situation where thread #1 is waiting for a lock to be released by thread #2, and thread #2 is waiting for a lock to be released by thread #1. When this situation happens, both threads are blocked and the program appears frozen. This situation is known as *deadlock* and is almost as bad a situation as a race condition because it can also happen at unpredictable, intermittent periods in the program.

Threading in C#

Figure 3 - Synchronizing the dual thread program using locks

Alternative Solution

can be used together for controlling the blocking of a thread. When an AutoResetEvent is initialized with false, the program will stop at the line of code that calls WaitOne until the Set method is called on the AutoResetEvent. After the Set method is executed on the AutoResetEvent, the thread becomes unblocked and is allowed to proceed past WaitOne. The next time WaitOne is called, it has automatically been reset, so the program will again wait (block) at the line of code in which the WaitOne method is executing. You can use this "stop and trigger" mechanism to block on one thread until another thread is ready to free the blocked thread by calling Set. Listing 3 shows our same two threads using the AutoResetEvent to block each other while the blocked thread waits and the unblocked thread executes to display _threadOutput to the Console. Initially, _blockThread1 is initialized to signal false, while _blockThread2 is initialized to signal true. This means that _blockThread2 will be allowed to proceed through the WaitOne call the first time through the loop in DisplayThread_2, while _blockThread1 will block on its WaitOne call in DisplayThread_1. When the _blockThread2 reaches the end of the loop in thread 2, it signals _blockThread1 by calling Set in order to release thread 1 from its block. Thread 2 then waits in its WaitOne call until Thread 1 reaches the end of its loop and calls Set on _blockThread2. The Set called in Thread 1 releases the block on thread 2 and the process starts again. Note that if we had set both AutoResetEvents (_blockThread1 and _blockThread2) initially to signal false, then both threads would be waiting to proceed through the loop without any chance to trigger each other, and we would experience a *deadlock.*

Listing 3 - Alternatively Blocking threads with the AutoResetEvent

```
AutoResetEvent _blockThread1 = new AutoResetEvent(false);
AutoResetEvent _blockThread2 = new AutoResetEvent(true);

/// <summary>
/// Thread 1, Displays that we are in thread 1
/// </summary>
void DisplayThread_1()
{
      while (_stopThreads == false)
      {
               // block thread 1  while the thread 2 is executing
                _blockThread1.WaitOne();

                // Set was called to free the block on thread 1, continue executing the code
                  Console.WriteLine("Display Thread 1");

```

```
                   // tell the user what thread we are in thread #1
                  Console.WriteLine("Thread 1 Output --> {0}", _threadOutput);

                // finished executing the code in thread 1, so unblock thread 2
                  _blockThread2.Set();
      }
}

/// <summary>
/// Thread 2, Displays that we are in thread 2
/// </summary>
void DisplayThread_2()
{
      while (_stopThreads == false)
      {
            // block thread 2  while thread 1 is executing
                  _blockThread2.WaitOne();

            // Set was called to free the block on thread 2, continue executing the code
                  Console.WriteLine("Display Thread 2");

                  _threadOutput = "Hello Thread 2";
                  Thread.Sleep(1000);  // simulate a lot of processing

                   // tell the user we are in thread #2
                  Console.WriteLine("Thread 2 Output --> {0}", _threadOutput);

            // finished executing the code in thread 2, so unblock thread 1
                _blockThread1.Set();
      }
}
```

The output produced by listing 3 is the same output as our locking code, shown in figure 3, but the AutoResetEvent gives us some more dynamic control over how one thread can notify another thread when the current thread is done the processing.

As we are pushing the theoretical limits of microprocessor speed, technology needs to find new ways to be able to optimize speed and performance in computer technology. With the invention of multiple processor chips and the inroads into parallel programming, understanding multithreading can prepare you for the paradigm needed to handle these more recent technologies that will bring us the advantage we need to continue to challenge Moore's Law. C# and .NET give us the ability to support multithreading and parallel processing. If we understand how to use to utilize these tools skillfully, we can prepare for these hardware promises of the future in our own day-to-day programming activities. In the meantime, *sharp*en your knowledge of threading, so you can *.net* the possibilities.

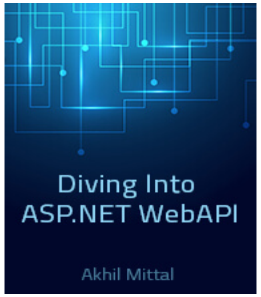
C# · C# Multiple Threading · Multithreading · Threading

Next Recommended Reading

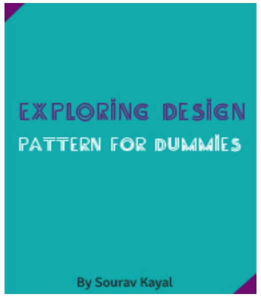
Introduction to Multithreading: Part 1

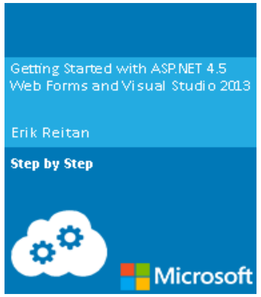
OUR BOOKS

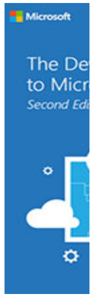
SOLUTIONS MANUAL TO OBJECTS FIRST WITH JAVA – A PRACTICAL INTRODUCTION USING BLUEJ
BY-AFZAAL AHMAD ZEESHAN
C#Corner
Total Page 34

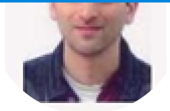


Microsoft
Introducing Microsoft SQL Server 2014 Technical Overview
ROSS MISTRY • STACIA MISNER




Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013
Erik Reitan
Step by Step
Microsoft





Michael Gold is President of Microgold Software Inc., makers of the WithClass UML Tool. His company is a Microsoft VBA Partner and Borland Partner. Mike is a Microsoft MVP and founding member of C# Corner. He has a BSEE ... Read more

https://www.c-sharpcorner.com/members/mike-gold2

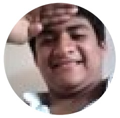
30 24.9m 1

View Previous Comments

30 38

Type your comment here and press Enter Key (Minimum 10 characters)

☑ Follow Comments



This article is really useful, it's easy and didactic for a newbie to this topic to understand it, and the analogy was really nice.

Jesús Hagiwara

•2049 •33 •0

Aug 11, 2019

1 0 Reply

Great and excellent example

Jorge Uribe

•2002 •80 •0

May 25, 2018

0 0 Reply

Great Article. Thanks

Sunil Kumar

•2016 •66 •0

Dec 07, 2017

1 0 Reply

Its awesome example..

shobhit bhalla

•1837 •245 •0

Sep 07, 2017

1 0 Reply

Easy to learn for everyone. Thanks for the post!

Guest User

•Tech Writer •71 •3.8k

Jan 13

2 0

● 2034 ● 48 ● 0

👍 2 ↩ 0 Reply

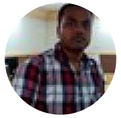
Nice...

kalu singh rao

● 356 ● 6.3k ● 94.9k

Jul 09, 2016

👍 2 ↩ 0 Reply



Its very power full article, thanks for sharing

Kailash Chandra Behera

● 170 ● 12.8k ● 7.7m

Nov 17, 2015

👍 2 ↩ 0 Reply

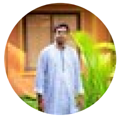
Thank you, nice article

Cesar Bravo

● 2081 ● 1 ● 0

Jul 27, 2015

👍 2 ↩ 0 Reply



good one...

Sharad

● 1277 ● 848 ● 103.8k

Jul 17, 2015

👍 2 ↩ 0 Reply

FEATURED ARTICLES

How To Upgrade to Windows 11

Exploring Subject <T> In Reactive Extensions For .Net

Micro Frontends With Webpack

What’s New In iPhone 13

Understanding Synchronization Context Task.ConfigureAwait In Action

View All ➔

TRENDING UP

01 The Best VS Code Extensions For Remote Working

02 Getting Started With .NET 6.0

03 Understanding Matplotlib With Examples

04 How To Post Data In ASP.NET Core Using Ajax

05 Top 15 Git Commands With Examples For Every Developers💪

06 Understanding Pandas With Examples

07 Visual Studio 2022 Installation Step By Step

08 Debugging The Hottest Release Of Visual Studio With Code Demos

09 The Best VS Code Extensions To Supercharge Git

10 Implementing Smart Contract in ASP.NET Application

About Us Contact Us Privacy Policy Terms Media Kit Sitemap Report a Bug FAQ Partners

C# Tutorials Common Interview Questions Stories Consultants Ideas Certifications