Chapter 14

Contax T

Waiting for the Orange Line

Collections And Threads

Learning Objectives

- *Understand the requirements for thread synchronization when manipulating collections*
- *Understand the difference between the ICollection and ICollection<T> interfaces*
- *Explain the purpose of the SyncRoot and IsSynchronized properties*
- *Explain how to create a synchronized collection and why it's not thread safe*
- *Understand how to synchronize access via the Monitor.Enter() and Monitor.Exit() methods*
- *State the relationship between the Monitor class and the C# lock keyword*
- *Employ the C# lock keyword to lock an object for thread synchronization*
- *State the names of the three synchronized collections in the System.Collections.Generic namespace*

Introduction

If you intend to use collection classes in a multithreaded environment you'll need to know how to ensure that only one thread has access to a collection at any time. This holds especially true if the items within a collection might be modified and enumerated by multiple threads. Fortunately, coordinating or synchronizing multiple thread access to a collection is easy to do; unfortunately, with the evolution of the .NET framework, several different thread synchronization strategies exist and are still supported in the framework, which makes it confusing for developers, both novice and experienced, as to which thread synchronization strategies work and which ones don't.

In this chapter I will show you how to synchronize multiple thread access to a collection. I will show you how to use the ICollection's SyncRoot and IsSynchronized properties as well as the Synchronized() method provided by some collections that is used to create Synchronized collection instances. I'll also explain why some collections implement the ICollection interface, which publishes the SyncRoot and IsSynchronized properties, while other collection's don't and how to program around this idiosyncrasy of the .NET collections framework. I will also explain why the Synchronized() method doesn't guarantee thread safety when enumerating through the elements of a collection.

Next I'll demonstrate the use of the Monitor.Enter() and Monitor.Exit() methods. I'll show you how to use the Monitor class in conjunction with a try/catch/finally block to ensure you exit the monitor. Following this I'll show you how to use the C# `lock` keyword to lock access to a collection using a separate lock object.

Some of the material I discuss in this chapter is deprecated in favor of more robust means of thread synchronization. I'm referring specifically to the reliance upon the SyncRoot and IsSynchronized properties of the ICollection interface and the use of synchronized collections created with the Synchronized() method found in some old-school, non-generic collection types. I present this material so that you better understand what you see when you dive into the .NET framework documentation and to increase your awareness of what has come before.

Also, I make no attempt to cover all aspects of thread synchronization. Specifically, I will omit coverage of WaitHandles, Mutexes, and the lightweight synchronization types introduced in .NET 4.0.

When you've finished this chapter you will have a clear understanding of how to apply a simple, effective thread synchronization strategy you can use to ensure thread-safe access to your collection objects. You'll also have a short list of simple rules to follow when implementing thread synchronization.

The Need For Thread Synchronization

If all you ever wanted to do was to read from a collection in a single-threaded environment then you could very well skip this chapter, and so could I, but that's not why you bought this book, so I'll keep typing.

Generally speaking, if your code is going to execute in a multi-threaded environment and multiple threads may execute *shared code segments* or access *shared resources or objects*, you'll want to control and coordinate access to these *critical code sections* by employing *thread synchronization mechanisms* provided by both the .NET framework and the C# language. However, not all thread synchronization mechanisms work as expected and in fact some are downright misleading. And, to make matters worse, the .NET framework has evolved and what was once provided for synchronization for the classes in the Collections namespace has been inconsistently carried forward and applied to the System.Collections.Generic classes. I'll talk more about this particular issue in another section titled: *SyncRoot, IsSynchroinzed, and Synchronized()*. Right now, I want to show you why thread synchronization is important, especially when multiple threads are trying to access and perhaps modify a collection's elements.

When might multiple threads need access to the same collection? The obvious scenario is when one thread is inserting objects into a collection and another thread is enumerating the collection at the same time. Example 14.1 offers a short program that demonstrates this scenario.

14.1 UnSynchronizedDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections.Generic;
4
5   public class UnSynchronizedDemo {
6
7      private List<int> _list = new List<int>();
```



```
8       private Random _random = new Random();
9       private const int ITEM_COUNT = 50;
10
11      public void InserterMethod(){
12        Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
13        try {
14          for(int i=0; i<ITEM_COUNT; i++){
15            _list.Add(_random.Next(500));
16          }
17
18          Thread.Sleep(10);
19
20          for(int i=0; i<ITEM_COUNT; i++){
21            _list.Add(_random.Next(500));
22          }
23        }catch(Exception e){
24          Console.WriteLine(e);
25        }
26        Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
27      }
28
29
30      public void ReaderMethod(){
31        Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
32
33        try{
34          foreach(int i in _list){
35            Console.Write(i + " ");
36            Thread.Sleep(10);
37          }
38        }catch(Exception e){
39          Console.WriteLine(e);
40        }
41
42        Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
43      }
44
45      public static void Main(){
46        UnSynchronizedDemo usd = new UnSynchronizedDemo();
47        Thread t1 = new Thread(usd.InserterMethod);
48        Thread t2 = new Thread(usd.ReaderMethod);
49        t1.Name = "Inserter Thread";
50        t2.Name = "Reader Thread";
51        t1.Start();
52        t2.Start();
53        t1.Join();
54        t2.Join();
55      }
56  }
```

Referring to example 14.1 — the UnSynchronzedDemo class declares and initializes a generic List<int> field named _list, a Random field named _random, and an integer constant named ITEM_COUNT. It defines two methods: the first on line 11 named InserterMethod() and the second on line 30 named ReaderMethod(). The InserterMethod() steps through the _list with a `for` statement inserting random values between 0 and 500. It then calls the Thread.Sleep() method on line 18 to pause for a moment before again inserting values into the _list with a second `for` loop.

The ReaderMethod() uses the `foreach` statement to iterate over the _list elements. As you know by now the `foreach` statement accesses a collection's enumerator.

The Main() method on line 45 creates an instance of the UnSynchronizedDemo class named usd and then creates two separate threads named t1 and t2. Thread t1 runs the InserterMethod and thread t2 runs the ReaderMethod. On lines 49 and 50 I name each thread appropriately and then start each thread. The calls to t1.Join() and t2.Join() signal the Main thread to pause until threads t1 and t2 have finished executing before exiting.

What will happen in this program depends on timing and the amount of items being inserted into the collection by the Inserter thread t1. It may execute normally or it may throw an exception. If run enough times you'll get either result, but mostly you'll get an exception because the Inserter thread is trying to modify the _list during the enumeration performed by the Reader thread. Figure 14-1 shows the usual result of running this program.

Referring to figure 14-1 — as the console output shows, the Inserter thread starts execution first followed by the Reader thread, which managed to print two numbers to the console before the Inserter thread again started to insert numbers into the _list, which caused the exception. To prevent the exception you'll need to coordinate access to the collection by using thread synchronization so that only one thread has access to the collection at any time. The following section shows how to use the C# lock keyword to synchronize thread access to a collection.

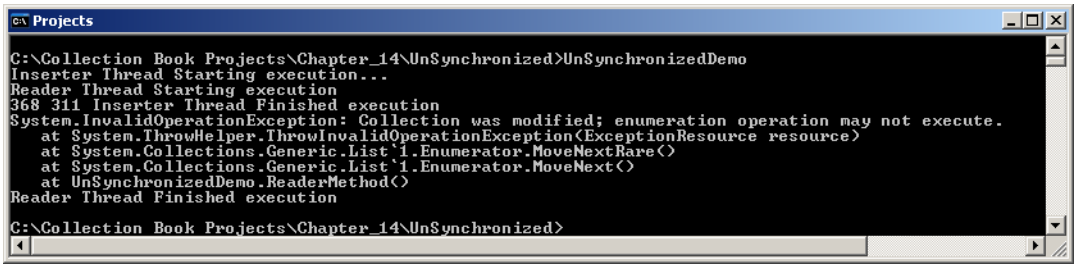

Figure 14-1: Results of Running Example 14.1

Quick Review

The need for thread synchronization arises when multiple threads of execution may access shared resources or shared code segments, which, if unsynchronized, would destabilize the code or leave the code in an invalid state. The .NET framework and the C# language provide various thread synchronization primitives and strategies that enable you to synchronize thread access to critical code segments.

Using The C# lock Keyword

The easiest way to implement thread synchronization is to use the C# lock keyword to obtain what is referred to as a "lock" on a particular object before entering a critical code section. Example 14.2 demonstrates the use of the lock keyword.

14.2 SynchronizedWithLockDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections.Generic;
4
5   public class SynchronizedWithLockDemo {
6      private List<int> _list = new List<int>();
7      private Random _random = new Random();
8      private const int ITEM_COUNT = 50;
9
10     public void InserterMethod(){
11       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
12       Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
13       lock(_list){
14         Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
15         for(int i=0; i<ITEM_COUNT; i++){
16           _list.Add(_random.Next(500));
17         }
18
19         Thread.Sleep(10);
20
21         for(int i=0; i<ITEM_COUNT; i++){
22           _list.Add(_random.Next(500));
23         }
24       }
25       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
26     }
27
28     public void ReaderMethod(){
29       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
30        Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
31       lock(_list){
32         Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
33         foreach(int i in _list){
34           Console.Write(i + " ");
35           Thread.Sleep(10);
36         }
37      }
38       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
39     }
40
41     public static void Main(){
```



```
42        SynchronizedWithLockDemo swld = new SynchronizedWithLockDemo();
43        Thread t1 = new Thread(swld.InserterMethod);
44        Thread t2 = new Thread(swld.ReaderMethod);
45        t1.Name = "Inserter Thread";
46        t2.Name = "Reader Thread";
47        t1.Start();
48        t2.Start();
49        t1.Join();
50        t2.Join();
51      }
52  }
```

Referring to example 14.2 — this program is the same as example 14.1 except that in the InserterMethod() and the ReaderMethod() access to the _list collection is synchronized with the use of the `lock` keyword. I've also added several more diagnostic console output statements to help trace the program's execution.

Note how the `lock` keyword is used. The `lock` keyword takes a reference to an object as an argument. The critical section is denoted by the opening and closing braces. In this example I'm using the _list itself as the lock object, which is perfectly fine.

The important thing to note is that **all threads you wish to synchronize must lock the same object**. I put this last phrase in bold because it's important. It does no good to try to synchronize access using different lock objects, as you'll see later when I show you how thread synchronization works under the covers.

Figure 14-2 shows the results of running this program.

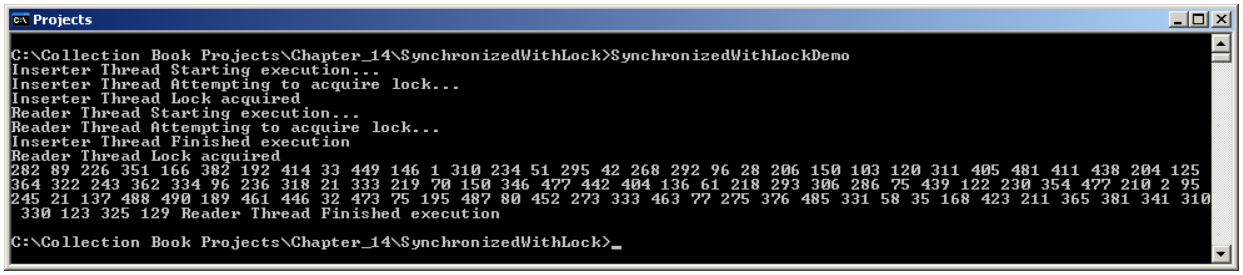

Figure 14-2: Results of Running Example 14.2

Referring to figure 14-2 — when the Inserter thread starts execution it immediately attempts to obtain the lock on the _list object. When the lock is acquired, the Inserter method enters the critical section. The Reader thread then starts execution and attempts to acquire the lock, but since the lock is held by the Inserter thread, it must wait until the Inserter thread completes and releases the lock on the _list object.

Note that in this example each thread runs to completion once it acquires the lock. So long a the Inserter thread runs first there will be items in the collection to enumerate. On the other hand, if the Reader thread manages to run first the _list would be empty. Again, this all depends on thread timing. Generally speaking, since I call t1.Start() first, the t1 thread is first to begin execution. Later I'll show you how to implement fine-grained thread control to handle the case where the Reader thread runs first and finds the _list empty. Before I do that I want to show you how thread synchronization works under the covers in the .NET runtime.

Quick Review

The C# `lock` keyword is the easiest way to protect critical code segments. Use the C# `lock` keyword to obtain a "lock" on an object. Place the code you want to protect within the body of the `lock` statement. **Recommendation: Lock on private field objects only.** Do not lock on the current instance (i.e. `this`). **Warning: Do not lock on value objects.** Value object are boxed into objects when used in a `lock` statement. Thus, multiple threads "locking" on the same value object will actually be acquiring locks on different objects.

Anatomy Of .NET Thread Synchronization

Figure 14-3 shows a diagram of how thread synchronization is implemented in the .NET runtime. I drew this diagram after studying the Microsoft Shared Source Common Language Infrastructure 2.0 (SSCLI 2.0) code which you can download from Microsoft.com. (See the References section.) The SSCLI virtual machine (VM) is implemented in C++. The four files of particular interest include: *Object.h*, *Object.cpp*, *SyncBlock.h*, and *SyncBlock.cpp*.

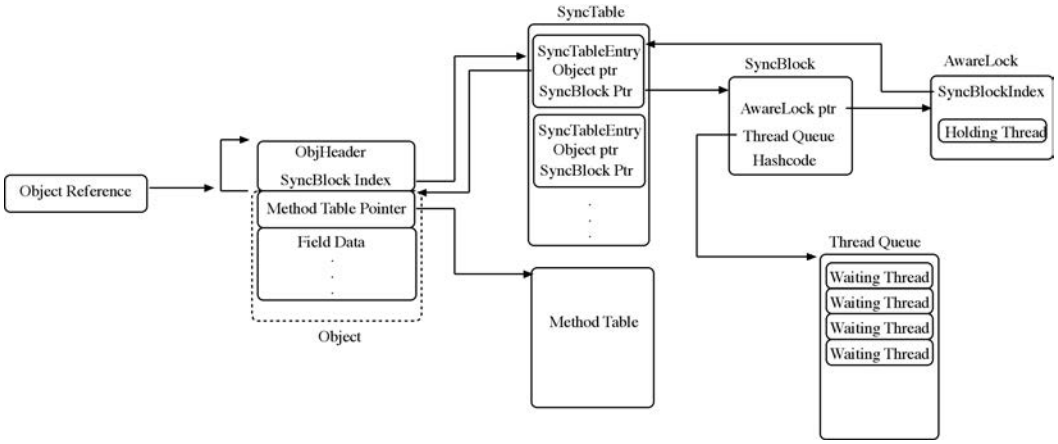

Figure 14-3: Thread Synchronization in the .NET Virtual Machine

Referring to figure 14-3 — the key players in thread synchronization include Object, ObjHeader, SyncTable, SyncTableEntry, SyncBlock, AwareLock, and ThreadQueue. Moving from left to right: an object reference points to an object instance within the virtual machine. This object instance is represented by the Object class as defined in the C++ virtual machine code. An object consists of a method table pointer and field data. At a negative offset from the beginning of the object is an object header (ObjHeader) which contains a data structure that, among other things, contains an index value to an entry into a SyncTable, which is an array of SyncTableEntry objects. For most objects in your program, the value of the SyncBlock index will be 0, meaning the object is not being used as a lock for a particular thread. When your code obtains a lock on a particular object, an unused SyncBlock is fetched from a SyncBlockCache (not shown in the diagram) and a SyncTableEntry is created in the SyncTable. The SyncTableEntry object has an object pointer that points back to the lock object, and a SyncBlock pointer that points to the SycnBlock. The SyncBlock object has a pointer to an AwareLock object and to a ThreadQueue which maintains a list of threads waiting to acquire the lock on the lock object. The bulk of the work is performed by the AwareLock class. Later, when you see how to use the Monitor.Enter() and Monitor.Exit() methods, it's the AwareLock object behind the scenes in the virtual machine that implements these methods as defined by the .NET System.Threading.Monitor class.

Old School – SyncRoot, IsSynchronized, and Synchronized()

The initial release of the .NET framework offered a confusing selection of properties and methods that gave developers a false sense of security with regards to thread synchronization. The ICollection interface provided the SyncRoot property which returns an object that can be used for thread synchronization. Most collections within the System.Collections namespace provide a Synchronized() method which is used to create a Synchronized collection instance. The IsSynchronized property simply returns true or false indicating whether or not a collection is synchronized.

The problem with creating and using a synchronized collection is that while access to certain parts of a collection's methods were synchronized, enumerating the collection's elements was not a thread safe operation. Studying the evolution of the .NET framework, which includes observing how developers learned to use .NET framework over the years since its release, leads me to conclude that it was developer confusion with regards to how to properly implement effective thread synchronization using the tools at hand, vs. any problems with the .NET thread synchronization tools per se.

Example 14.3 shows an example of a synchronized ArrayList created with the Array.Synchronized() method.

14.3 OldSchoolDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections;
4
5   public class OldSchoolDemo {
6      private ArrayList _list = new ArrayList();
7      private ArrayList _synchronizedList = null;
8      private const int ITEM_COUNT = 100;
9      private Random _random = new Random();
10
11     public OldSchoolDemo(){
12       _synchronizedList = ArrayList.Synchronized(_list);
13
14     }
15
16     public void PrintListStats(){
17       Console.WriteLine("The _list field IsSynchronized value: "
18                                              + _list.IsSynchronized);
19       Console.WriteLine("The _synchronizedList field IsSynchronized value: "
20                                              + _synchronizedList.IsSynchronized);
21     }
22
23     public void InserterMethod(){
24       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
25       for(int i=0; i<ITEM_COUNT; i++){
26         _synchronizedList.Add(_random.Next(500));
27       }
28
29       Thread.Sleep(10);
30
31       for(int i=0; i<ITEM_COUNT; i++){
32         _synchronizedList.Add(_random.Next(500));
33       }
34       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution...");
35     }
36
37
38     private void ReaderMethod(){
39       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
40       try{
41         foreach(int i in _synchronizedList){
42           Console.Write(i + " ");
43         }
44       }catch(Exception e){
45         Console.WriteLine(e);
46       }
47       Console.WriteLine(Thread.CurrentThread.Name + " Finsihed execution...");
48     }
49
50    public static void Main(){
51      OldSchoolDemo osd = new OldSchoolDemo();
52      osd.PrintListStats();
53      Thread t1 = new Thread(osd.InserterMethod);
54      Thread t2 = new Thread(osd.ReaderMethod);
55      t1.Name = "Inserter thread";
56      t2.Name = "Reader thread";
57      t1.Start();
58      t2.Start();
59      t1.Join();
60      t2.Join();
61    }
62  }
```

Referring to example 14.3 — the OldSchoolDemo class declares and initializes an ArrayList named _list, an integer constant named ITEM_COUNT, and a Random object named _random. The initialization of _synchronizedList is performed in the body of the constructor. Note how the static method Array.Synchronized() is used to create the synchronized version of the array list. On line 16 the PrintListStats() method prints to the console the results obtained via calls to the IsSynchronized property on the _list and _synchronizedList.

The InserterMethod inserts random integers between the values 0 and 500 into the _list. It then sleeps for 10 milliseconds and then inserts more integers into the _list. The ReaderMethod uses the `foreach` method to print the list items to the console.

The Main() method creates two threads named t1 and t2. Thread t1 runs the InserterMethod and thread t2 runs the ReaderMethod(). Thread t1 is named Inserter and thread t2 is named Reader.

Figure 14-4 shows the results of running this program.

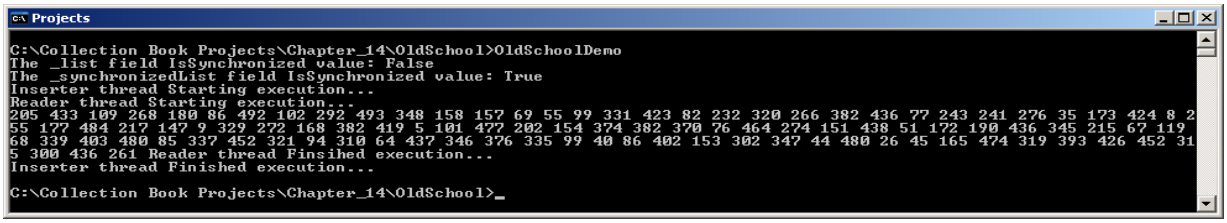

Figure 14-4: One Possible Result of Running Example 14.3

Referring to figure 14-4 — notice that the Inserter thread did not finish execution before the Reader thread started to run. It was by pure luck of timing that an exception was not thrown. Figure 14-5 shows the usual result of running this program repeatedly.

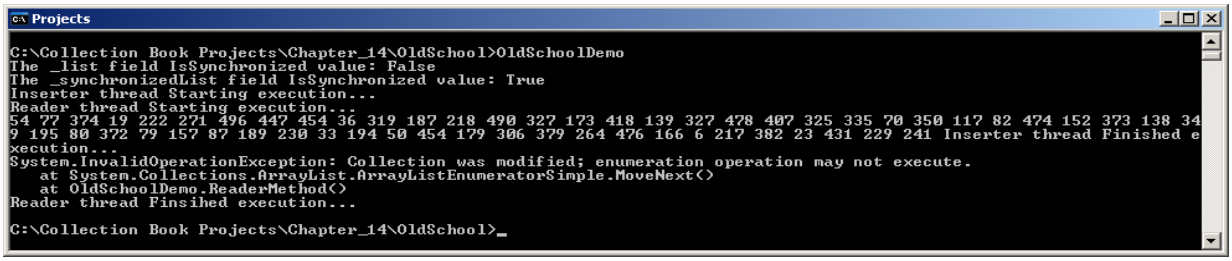

Figure 14-5: The Usual Result of Running Example 14.3

Even though the list is synchronized, you must still take steps to coordinate multithread access to it when enumerating its elements. Example 14.4 shows how the `lock` keyword could be used in conjunction with the _synchronizedList.SyncRoot property.

14.4 OldSchoolSyncRootDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections;
4
5   public class OldSchoolSyncRootDemo {
6      private ArrayList _list = new ArrayList();
7      private ArrayList _synchronizedList = null;
8      private const int ITEM_COUNT = 50;
9      private Random _random = new Random();
10
11     public OldSchoolSyncRootDemo(){
12       _synchronizedList = ArrayList.Synchronized(_list);
13
14     }
15
16     public void PrintListStats(){
17       Console.WriteLine("The _list field IsSynchronized value: "
18                                              + _list.IsSynchronized);
19       Console.WriteLine("The _synchronizedList field IsSynchronized value: "
20                                              + _synchronizedList.IsSynchronized);
21     }
22
23     public void InserterMethod(){
24       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
25        Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire the lock...");
26       lock(_synchronizedList.SyncRoot){
27         Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired...");
28         for(int i=0; i<ITEM_COUNT; i++){
29           _synchronizedList.Add(_random.Next(500));
30         }
31
32         Console.WriteLine(Thread.CurrentThread.Name + " Sleeping...");
33         Thread.Sleep(10);
34
35         for(int i=0; i<ITEM_COUNT; i++){
36           _synchronizedList.Add(_random.Next(500));
37         }
38       }
39       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution...");
40     }
41
42
43     private void ReaderMethod(){
```



```
44        Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
45        lock(_synchronizedList.SyncRoot){
46          try{
47            foreach(int i in _synchronizedList){
48              Console.Write(i + " ");
49              Console.Write(Thread.CurrentThread.Name + " Sleeping...");
50              Thread.Sleep(10);
51            }
52          }catch(Exception e){
53            Console.WriteLine(e);
54          }
55        }
56        Console.WriteLine(Thread.CurrentThread.Name + " Finished execution...");
57      }
58
59    public static void Main(){
60      OldSchoolSyncRootDemo ossrd = new OldSchoolSyncRootDemo();
61      ossrd.PrintListStats();
62      Thread t1 = new Thread(ossrd.InserterMethod);
63      Thread t2 = new Thread(ossrd.ReaderMethod);
64      t1.Name = "Inserter thread";
65      t2.Name = "Reader thread";
66      t1.Start();
67      t2.Start();
68      t1.Join();
69      t2.Join();
70    }
71  }
```

Referring to example 14.4 — this code is similar to example 14.3 except now the `lock` keyword is being used to protect the critical section of the InserterMethod() and the ReaderMethod(). (Lines 26 and 45 respectively.) Note that in this case I'm locking on the _synchronizedList.SyncRoot property which is more than likely just a reference to the _synchronizedList object itself behind the scenes. Figure 14-6 shows one possible result of running this program.

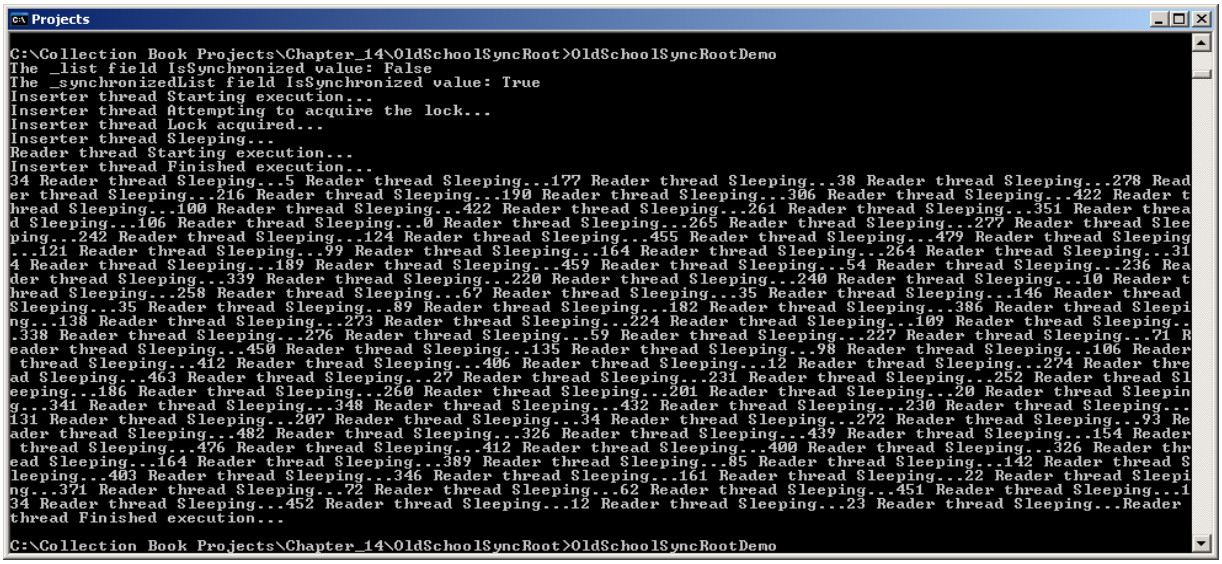

Figure 14-6: One Possible Result of Running Example 14.4

Again, depending on when thread t1 actually starts running, thread t2 may start to run before t1 acquires the lock and gets a chance to insert any items into the _synchronizedList. Figure 14-7 shows another possible result of running example 14.4.

Quick Review

Collection classes in the System.Collections namespace come equipped with the SyncRoot and IsSynchronized properties. These old-school collections also provided a static Synchronized() method which is used to transform an ordinary collection into a synchronized collection. And while individual collection methods may be synchronized, it was still not thread safe to enumerate over a collection. While you can still write good-quality thread-safe code using the SyncRoot property along with the `lock` keyword or the Monitor class, the use of these old-school properties,

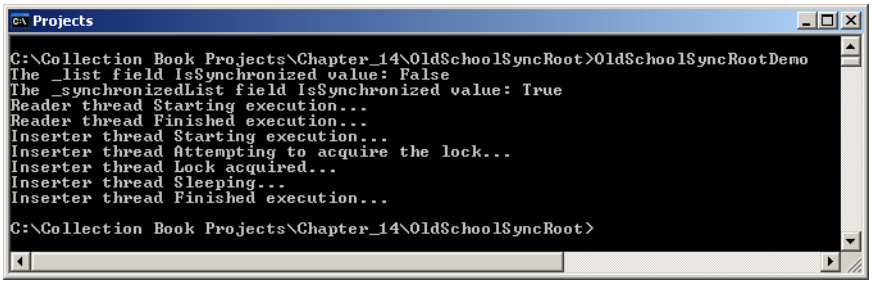

Figure 14-7: Another Possible Result from Running Example 14.4

along with the Synchronized() method is best avoided. Besides, unless you find yourself maintaining legacy C# code, you should be favoring the use of the generic collection classes.

Monitor.Enter() and Monitor.Exit()

The Monitor class can be used to synchronize thread access to critical code sections just like the C# `lock` keyword. In fact, the C# `lock` keyword is translated into Monitor.Enter() and Monitor.Exit() method calls by the compiler. Example 14.5 lists the decompiled intermediate language for the InserterMethod() of example 14.2.

14.5 Decompiled InserterMethod from Example 14.2

```
1    .method public hidebysig instance void  InserterMethod() cil managed
2    {
3      // Code size       247 (0xf7)
4      .maxstack  3
5      .locals init (int32 V_0,
6                bool V_1,
7                class [mscorlib]System.Collections.Generic.List`1<int32> V_2,
8                bool V_3)
9      IL_0000:  nop
10     IL_0001:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
11     IL_0006:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
12     IL_000b:  ldstr      " Starting execution..."
13     IL_0010:  call       string [mscorlib]System.String::Concat(string,
14                                                                 string)
15     IL_0015:  call       void [mscorlib]System.Console::WriteLine(string)
16     IL_001a:  nop
17     IL_001b:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
18     IL_0020:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
19     IL_0025:  ldstr      " Attempting to acquire lock..."
20     IL_002a:  call       string [mscorlib]System.String::Concat(string,
21                                                                 string)
22     IL_002f:  call       void [mscorlib]System.Console::WriteLine(string)
23     IL_0034:  nop
24     IL_0035:  ldc.i4.0
25     IL_0036:  stloc.1
26     .try
27     {
28       IL_0037:  nop
29       IL_0038:  ldarg.0
30       IL_0039:  ldfld      class [mscorlib]System.Collections.Generic.List`1<int32>
SynchronizedWithLockDemo::_list
31       IL_003e:  dup
32       IL_003f:  stloc.2
33       IL_0040:  ldloca.s   V_1
34       IL_0042:  call       void [mscorlib]System.Threading.Monitor::Enter(object,
35                                                                           bool&)
36       IL_0047:  nop
37       IL_0048:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
38       IL_004d:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
39       IL_0052:  ldstr      " Lock acquired"
40       IL_0057:  call       string [mscorlib]System.String::Concat(string,
41                                                                   string)
42       IL_005c:  call       void [mscorlib]System.Console::WriteLine(string)
43       IL_0061:  nop
44       IL_0062:  ldc.i4.0
```



```
45       IL_0063:  stloc.0
46       IL_0064:  br.s       IL_0088
47       IL_0066:  nop
48       IL_0067:  ldarg.0
49       IL_0068:  ldfld      class [mscorlib]System.Collections.Generic.List`1<int32>
SynchronizedWithLockDemo::_list
50       IL_006d:  ldarg.0
51       IL_006e:  ldfld      class [mscorlib]System.Random SynchronizedWithLockDemo::_random
52       IL_0073:  ldc.i4     0x1f4
53       IL_0078:  callvirt   instance int32 [mscorlib]System.Random::Next(int32)
54       IL_007d:  callvirt   instance void class [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
55       IL_0082:  nop
56       IL_0083:  nop
57       IL_0084:  ldloc.0
58       IL_0085:  ldc.i4.1
59       IL_0086:  add
60       IL_0087:  stloc.0
61       IL_0088:  ldloc.0
62       IL_0089:  ldc.i4.s   50
63       IL_008b:  clt
64       IL_008d:  stloc.3
65       IL_008e:  ldloc.3
66       IL_008f:  brtrue.s   IL_0066
67       IL_0091:  ldc.i4.s   10
68       IL_0093:  call       void [mscorlib]System.Threading.Thread::Sleep(int32)
69       IL_0098:  nop
70       IL_0099:  ldc.i4.0
71       IL_009a:  stloc.0
72       IL_009b:  br.s       IL_00bf
73       IL_009d:  nop
74       IL_009e:  ldarg.0
75       IL_009f:  ldfld      class [mscorlib]System.Collections.Generic.List`1<int32>
SynchronizedWithLockDemo::_list
76       IL_00a4:  ldarg.0
77       IL_00a5:  ldfld      class [mscorlib]System.Random SynchronizedWithLockDemo::_random
78       IL_00aa:  ldc.i4     0x1f4
79       IL_00af:  callvirt   instance int32 [mscorlib]System.Random::Next(int32)
80       IL_00b4:  callvirt   instance void class [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
81       IL_00b9:  nop
82       IL_00ba:  nop
83       IL_00bb:  ldloc.0
84       IL_00bc:  ldc.i4.1
85       IL_00bd:  add
86       IL_00be:  stloc.0
87       IL_00bf:  ldloc.0
88       IL_00c0:  ldc.i4.s   50
89       IL_00c2:  clt
90       IL_00c4:  stloc.3
91       IL_00c5:  ldloc.3
92       IL_00c6:  brtrue.s   IL_009d
93       IL_00c8:  nop
94       IL_00c9:  leave.s    IL_00db
95     }  // end .try
96     finally
97     {
98       IL_00cb:  ldloc.1
99       IL_00cc:  ldc.i4.0
100      IL_00cd:  ceq
101      IL_00cf:  stloc.3
102      IL_00d0:  ldloc.3
103      IL_00d1:  brtrue.s   IL_00da
104      IL_00d3:  ldloc.2
105      IL_00d4:  call       void [mscorlib]System.Threading.Monitor::Exit(object)
106      IL_00d9:  nop
107      IL_00da:  endfinally
108    }  // end handler
109    IL_00db:  nop
110    IL_00dc:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
111    IL_00e1:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
112    IL_00e6:  ldstr      " Finished execution"
113    IL_00eb:  call       string [mscorlib]System.String::Concat(string,
114                                                                string)
115    IL_00f0:  call       void [mscorlib]System.Console::WriteLine(string)
116    IL_00f5:  nop
117    IL_00f6:  ret
118  } // end of method SynchronizedWithLockDemo::InserterMethod
```


Referring to example 14.5 — the InserterMethod() in example 14.2 used the C# `lock` keyword to synchronize thread access to its critical section. Line 34 shows how the actual call is made to the Monitor.Enter() and later, on line 105 to Monitor.Exit().

Using Monitor.Enter() and Monitor.Exit()

While the C# `lock` keyword makes thread synchronization easy, the use of the Monitor class demands you pay more attention to what you're doing. You must be sure to call Monitor.Exit() for each call to Monitor.Enter(). The way to ensure this happens is to use the Monitor.Enter() and Monitor.Exit() methods in conjunction with a `try/catch/finally` block. Example 14.6 demonstrates the use of Monitor.Enter() and Monitor.Exit().

14.6 MonitorDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections.Generic;
4
5   public class MonitorDemo {
6
7      private List<int> _list = new List<int>();
8      private Random _random = new Random();
9      private const int ITEM_COUNT = 50;
10
11     public void InserterMethod(){
12       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
13       Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
14       Monitor.Enter(_list);
15        Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
16       try{
17           for(int i=0; i<ITEM_COUNT; i++){
18             _list.Add(_random.Next(500));
19           }
20
21            Console.WriteLine(Thread.CurrentThread.Name + " Sleeping...");
22           Thread.Sleep(10);
23
24           for(int i=0; i<ITEM_COUNT; i++){
25             _list.Add(_random.Next(500));
26           }
27        }catch(Exception e){
28           Console.WriteLine(e);
29       }finally{
30         Monitor.Exit(_list);
31          Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
32       }
33        Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
34     }
35
36     public void ReaderMethod(){
37       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
38       Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
39       Monitor.Enter(_list);
40       Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
41       try{
42         foreach(int i in _list){
43           Console.Write(i + " ");
44            Console.Write(Thread.CurrentThread.Name + " Sleeping...");
45           Thread.Sleep(10);
46         }
47       }catch(Exception e){
48         Console.WriteLine(e);
49       }finally{
50         Monitor.Exit(_list);
51          Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
52       }
53       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
54     }
55
56
57     public static void Main(){
58       MonitorDemo md = new MonitorDemo();
59       Thread t1 = new Thread(md.InserterMethod);
60       Thread t2 = new Thread(md.ReaderMethod);
61       t1.Name = "Inserter Thread";
62       t2.Name = "Reader Thread";
63       t1.Start();
64       t2.Start();
```



```
65        t1.Join();
66        t2.Join();
67      }
68  }
```

Referring to example 14.6 — this program is similar to example 14.2 only the critical section in the InserterMethod() and ReaderMethod() is protected with the help of Monitor.Enter() and Monitor.Exit(). Note that a reference to the lock object is passed to both the Monitor.Enter() and Monitor.Exit() methods. (e.g., Monitor.Enter(_list) and Monitor.Exit(_list))

Let's take a closer look at the use of Monitor.Enter() and Monitor.Exit() in the body of the InserterMethod(). The call to Monitor.Enter(_list) is made on line 14. **The Monitor.Enter() method blocks until a lock is obtained.** This effectively stops execution of the current thread until the thread that owns the lock on _list, which in this example would be the ReaderMethod(), releases its lock on _list. Note too that the call to Monitor.Enter() marks the beginning of the critical section. Figure 14-8 shows the results of running this program.

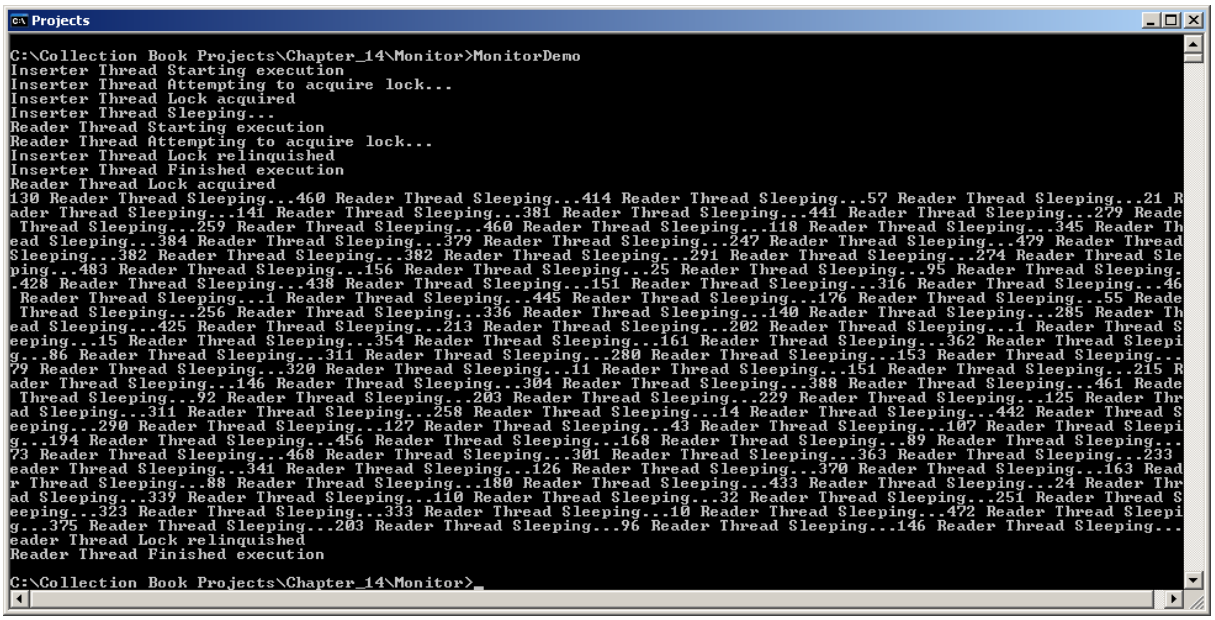

Figure 14-8: Results of Running Example 14.6

Using Overloaded Monitor.Enter() Method

The single-argument version of the Monitor.Enter() method is obsolete as of .NET 4.0 and it's recommended that going forward you use the overloaded version of the method which takes two arguments: a reference to a lock object and a boolean ref variable that is set to true if the lock is acquired. The use of the new overloaded Monitor.Enter() method comes with a new recommended usage structure as well. Example 14.7 demonstrates the use of the overloaded Monitor.Enter() method. This example also demonstrates the use of the Monitor.Wait() and Monitor.Pulse() methods.

14.7 MonitorLockTakenDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections.Generic;
4
5   public class MonitorLockTakenDemo {
6
7      private List<int> _list = new List<int>();
8      private Random _random = new Random();
9      private const int ITEM_COUNT = 50;
10
11
12     public void InserterMethod(){
13      Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
14       bool lockTaken = false;
15       try{
16         Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
```



```
17          Monitor.Enter(_list, ref lockTaken);
18          if(lockTaken){
19            Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
20            for(int i=0; i<ITEM_COUNT; i++){
21              _list.Add(_random.Next(500));
22            }
23
24            Console.WriteLine(Thread.CurrentThread.Name + " Sleeping");
25            Thread.Sleep(10);
26            Console.WriteLine(Thread.CurrentThread.Name + " Pulse waiting threads...");
27            Monitor.Pulse(_list);
28
29            for(int i=0; i<ITEM_COUNT; i++){
30              _list.Add(_random.Next(500));
31            }
32          }
33         }catch(Exception e){
34            Console.WriteLine(e);
35       }finally{
36          if(lockTaken){
37           Monitor.Exit(_list);
38           Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
39         }
40       }
41        Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
42     }
43
44     public void ReaderMethod(){
45        Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
46       bool lockTaken = false;
47       try{
48       while(!lockTaken){
49         Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
50         Monitor.Enter(_list, ref lockTaken);
51         if(lockTaken){
52           Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
53           if(_list.Count == 0){
54             Console.WriteLine(Thread.CurrentThread.Name + " List is currently empty. Releasing the lock.");
55             Monitor.Wait(_list);
56           }
57           foreach(int i in _list){
58             Console.Write(i + " ");
59             Console.Write(Thread.CurrentThread.Name + " Sleeping  ");
60             Thread.Sleep(10);
61           }
62         }
63       }
64      }catch(Exception e){
65        Console.WriteLine(e);
66      }finally{
67        if(lockTaken){
68         Monitor.Exit(_list);
69          Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
70        }
71      }
72       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
73     }
74
75
76     public static void Main(){
77       MonitorLockTakenDemo mltd = new MonitorLockTakenDemo();
78       Thread t1 = new Thread(mltd.InserterMethod);
79       Thread t2 = new Thread(mltd.ReaderMethod);
80       t1.Name = "Inserter Thread";
81       t2.Name = "Reader Thread";
82       t2.Start();
83       Thread.Sleep(10);
84       t1.Start();
85       t1.Join();
86       t2.Join();
87     }
88   }
```

Referring to example 14.7 — this example, while similar to the previous examples, is structured differently. It still consists of two primary threads, t1 and t2. Thread t1 is the Inserter thread and t2 is the Reader thread. However, the Main() method starts t2 first to demonstrate what happens when the ReaderMethod() finds the _list empty.

Referring to the ReaderMethod() which begins on line 44 — a local variable named lockTaken is declared and initialized to false on line 46. The `try` block begins on the next line which includes a `while` loop that checks the value of lockTaken. If lockTaken is false, a call to the overloaded Monitor.Enter() method is made passing in a refer-

ence to the _list as the first argument and the lockTaken variable passed in using the ref keyword as the second argument. **If a lock already exists on _list, the call to Monitor.Enter() will block until the lock is released and acquired.** When the lock is acquired, the lockTaken variable is set to true and the `if` statement on line 51 is entered. The `if` statement on line 53 checks the value of _list.Count and if it finds the list empty it releases the lock with a call to Monitor.Wait(_list). **The call to Monitor.Wait() blocks until the lock is again acquired.** When the lock is reacquired, the `foreach` statement on line 57 executes and enumerates through the collection printing the items to the console, making a call to Thread.Sleep(10) during each iteration.

Referring to the InserterMethod() on line 12 — a local variable named lockTaken is declared and initialized to false on line 14. On line 17 the overloaded version of Monitor.Enter() is called. When the lock becomes available, the InserterMethod() will start to insert integers into the _list. After the first `for` statement the thread is put to sleep with a call to Thread.Sleep(10) followed by a call to Monitor.Pulse(_list) which signals threads waiting to obtain a lock on the _list object to wake up and try to obtain the lock.

In the Main() method which begins on line 76, thread t2 is started first followed by a call to Thread.Sleep(10), which puts the Main thread to sleep, giving a chance for the t2 thread to get going before calling t1.Start(). Figure 14-9 shows the results of running this program.

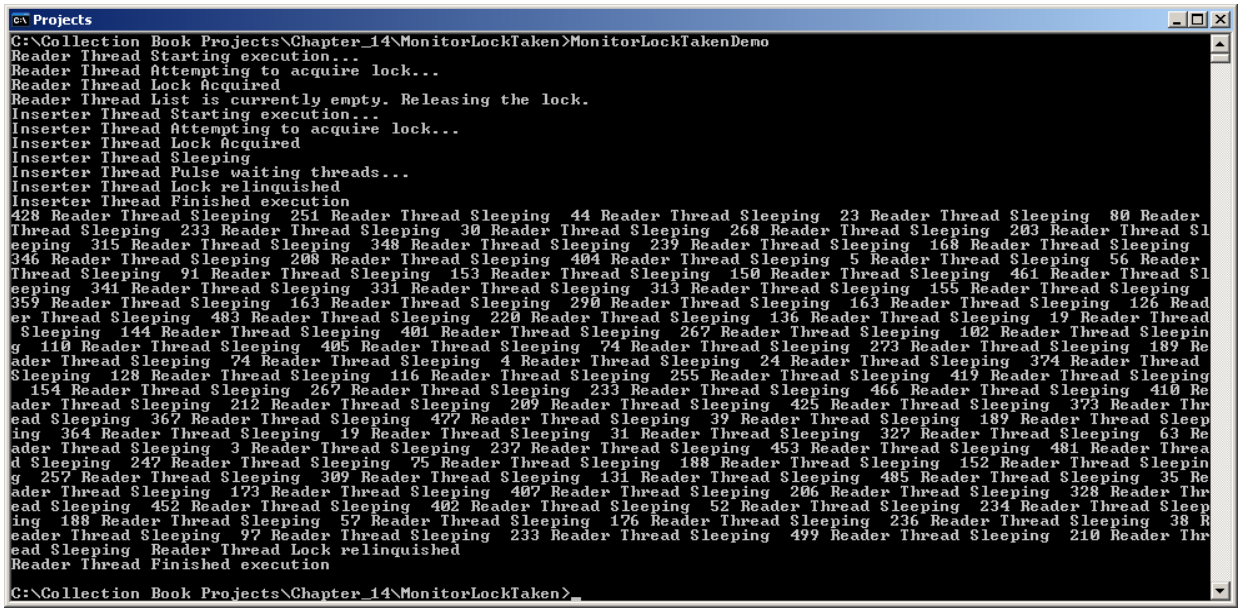

Figure 14-9: Results of Running Example 14.7

Non-Blocking Monitor.TryEnter()

The Monitor.TryEnter() method is a non-blocking method, which means that regardless of whether or not the lock is acquired, the method will immediately return. This method is also overloaded and the use of the two-argument version is recommend going forward. Example 14.8 demonstrates the use of the Monitor.TryEnter() method.

14.8 MonitorTryEnterDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections.Generic;
4
5   public class MonitorTryEnterDemo {
6
7      private List<int> _list = new List<int>();
8      private Random _random = new Random();
9      private const int ITEM_COUNT = 50;
10
11
12     public void InserterMethod(){
13       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
14       bool lockTaken = false;
15       try{
16         Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
```



```
17          Monitor.TryEnter(_list, ref lockTaken);
18          if(lockTaken){
19            Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
20            for(int i=0; i<ITEM_COUNT; i++){
21              _list.Add(_random.Next(500));
22            }
23
24            Console.WriteLine(Thread.CurrentThread.Name + " Sleeping");
25            Thread.Sleep(10);
26             Console.WriteLine(Thread.CurrentThread.Name + " Pulse waiting threads...");
27            Monitor.Pulse(_list);
28
29            for(int i=0; i<ITEM_COUNT; i++){
30              _list.Add(_random.Next(500));
31            }
32          }
33         }catch(Exception e){
34            Console.WriteLine(e);
35       }finally{
36          if(lockTaken){
37           Monitor.Exit(_list);
38            Console.WriteLine(Thread.CurrentThread.Name + " Relinquish the lock");
39         }
40       }
41       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
42     }
43
44     public void ReaderMethod(){
45       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
46       bool lockTaken = false;
47       try{
48       while(!lockTaken){
49         Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
50         Monitor.TryEnter(_list, ref lockTaken);
51         if(lockTaken){
52           Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
53           if(_list.Count == 0){
54             Console.WriteLine(Thread.CurrentThread.Name + " List is currently empty. Releasing the lock.");
55             Monitor.Wait(_list);
56           }
57           foreach(int i in _list){
58             Console.Write(i + " ");
59             Console.Write(Thread.CurrentThread.Name + " Sleeping  ");
60             Thread.Sleep(10);
61           }
62         }
63       }
64      }catch(Exception e){
65        Console.WriteLine(e);
66      }finally{
67        if(lockTaken){
68         Monitor.Exit(_list);
69         Console.WriteLine(Thread.CurrentThread.Name + " Relinquish the lock");
70        }
71      }
72      Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
73     }
74
75
76     public static void Main(){
77       MonitorTryEnterDemo mted = new MonitorTryEnterDemo();
78       Thread t1 = new Thread(mted.InserterMethod);
79       Thread t2 = new Thread(mted.ReaderMethod);
80       t1.Name = "Inserter Thread";
81       t2.Name = "Reader Thread";
82       t2.Start();
83       Thread.Sleep(10);
84       t1.Start();
85       t1.Join();
86       t2.Join();
87     }
88  }
```

Referring to example 14.8 — this program is similar to the previous example, only the Monitor.TryEnter() method is used in place of the Monitor.Enter() method. Note that even though I'm starting thread t2 first, there is no guarantee it will start first. (And this applies to the previous example as well.) Figures 14-10 and 14-11 show two possible outcomes from running this program repeatedly.

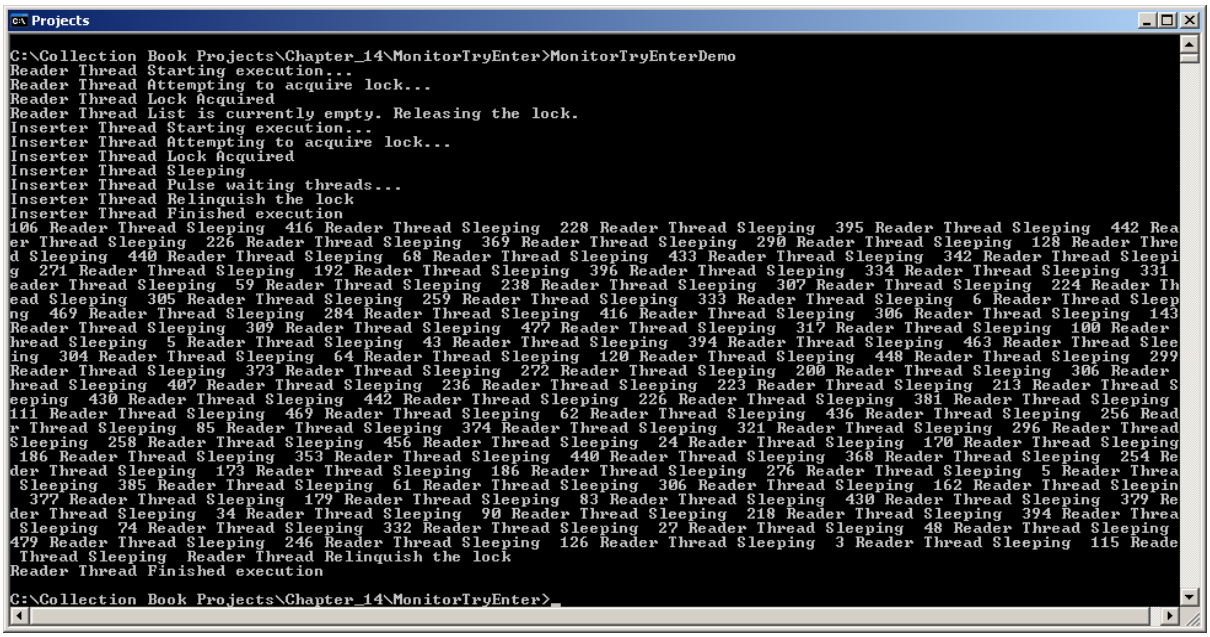

Figure 14-10: Results of Running Example 14.8

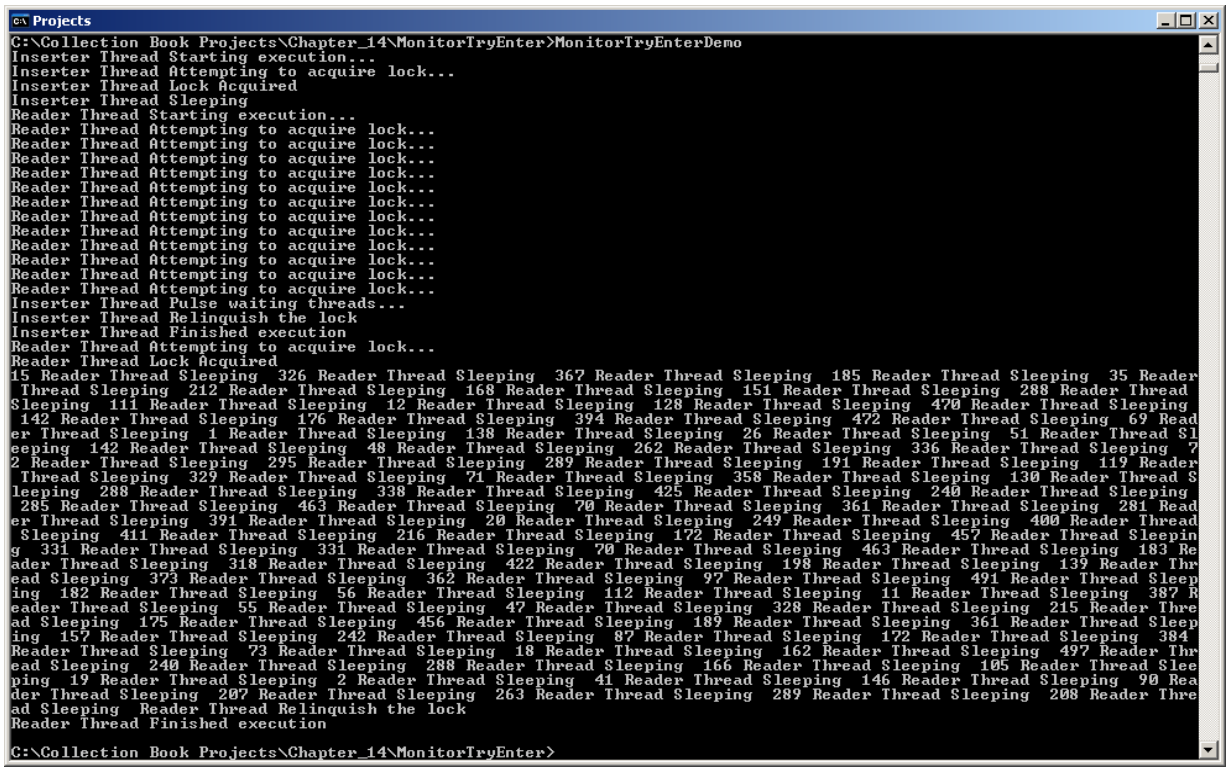

Figure 14-11: Another Possible Result of Running Example 14.8

Quick Review

The static Monitor class allows you to implement fine grained thread synchronization. You must be sure that for each call to Monitor.Enter(_lockObject) is followed by a call to Monitor.Exit(_lockObject). Failure to do so may

result in deadlock as waiting threads will never acquire an unreleased lock. The critical code section begins with a call to Monitor.Enter(). Place the call to Monitor.Exit() in the body of the `finally` clause of a `try/catch/finally` block. **The Monitor.Enter() method blocks until it acquires the lock.** The Monitor.Enter() method is overloaded. Favor the use of the two-argument version of Monitor.Enter() going forward.

The Monitor.TryEnter() method is a non-blocking method that returns immediately after it's called regardless of whether or not the lock is acquired. You must take this immediate return behavior into account in your code. Use the overloaded two-argument version of the Monitor.TryEnter() method to test whether or not the lock was acquired.

If a thread needs to give up the lock because it has nothing to do, call the Monitor.Wait() method. To signal waiting threads of a change in lock status, call the Monitor.Pulse() method to move the next waiting thread into the ready queue.

Synchronizing Entire Methods

If you're using the C# `lock` keyword to synchronize significant portions of a method's body, you can alternatively tag the entire method as being synchronized using the [MethodImpl(MethodImplOptions.Synchronized)] attribute. It's easy to use. Simply apply the attribute to each method you want to synchronize.

14.9 SynchronizedMethodDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5
6   public class SynchronizedMethodDemo {
7
8      private List<int> _list = new List<int>();
9      private Random _random = new Random();
10     private const int ITEM_COUNT = 50;
11
12     [MethodImpl(MethodImplOptions.Synchronized)]
13     public void InserterMethod(){
14       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
15       try {
16         for(int i=0; i<ITEM_COUNT; i++){
17           _list.Add(_random.Next(500));
18         }
19
20         Thread.Sleep(10);
21
22         for(int i=0; i<ITEM_COUNT; i++){
23           _list.Add(_random.Next(500));
24         }
25       }catch(Exception e){
26         Console.WriteLine(e);
27       }
28       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
29     }
30
31     [MethodImpl(MethodImplOptions.Synchronized)]
32     public void ReaderMethod(){
33       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
34
35       try{
36         foreach(int i in _list){
37           Console.Write(i + " ");
38           Thread.Sleep(10);
39         }
40       }catch(Exception e){
41         Console.WriteLine(e);
42       }
43
44       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
45     }
46
47
48     public static void Main(){
49       SynchronizedMethodDemo smd = new SynchronizedMethodDemo();
50       Thread t1 = new Thread(smd.InserterMethod);
51       Thread t2 = new Thread(smd.ReaderMethod);
52       t1.Name = "Inserter Thread";
53       t2.Name = "Reader Thread";
```



```
54        t1.Start();
55        t2.Start();
56        t1.Join();
57        t2.Join();
58      }
59  }
```

Referring to example 14.9 — the [MethodImpl(MethodImplOptions.Synchronized)] attribute is applied to both thread methods. The use of the [MethodImpl(MethodImplOptions.Synchronized)] attribute is essentially applying the Monitor.Enter()/Monitor.Exit() thread synchronization mechanism to the entire body of the method, locking on the instance (i.e., Monitor.Enter(this)/Monitor.Exit(this)). Figure 14-11 shows the results of running this program.

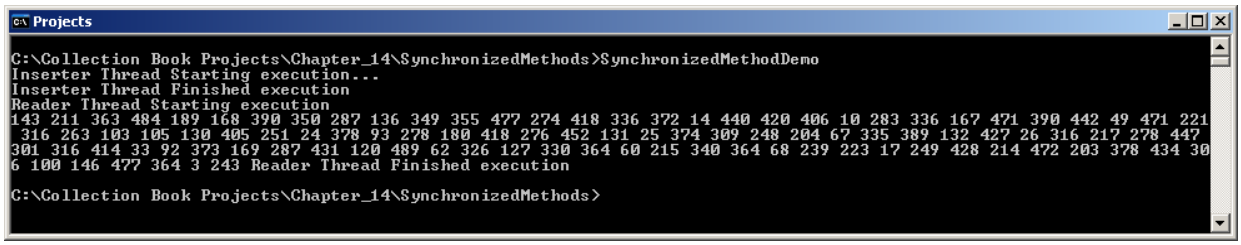

Figure 14-12: Results of Running Example 14.9

Quick Review

Use the [MethodImpl(MethodImplOptions.Synchronized)] attribute to synchronize entire methods. However, I recommend using this attribute sparingly. Generally speaking, the finer grained you can make your thread synchronization scheme, the better off you'll be.

Synchronized Collections In The System.Collections.Generic Namespace

The System.Collections.Generic namespace contains three "synchronized" collections named: SynchronizedCollection<T>, SynchronizedKeyedCollection<T>, and SynchronizedReadOnlyCollection<T>.

I put quotes around the word "synchronized" because even though these collections start with the word Synchronized, and the .NET documentation describes each class as a "...thread-safe collection...", the documentation also says a little further down the page "Any instance members are not guaranteed to be thead safe."

So, what's so special about these collections? Well, nothing really, except that each provides a SyncRoot property that can be set via the constructor. If the default constructor is used, the SyncRoot property returns a reference to a default Object instance.

I will leave it to you to explore the use of these synchronized collections as you see fit.

Thread Synchronization – Recommendations For Usage

Thread synchronization in any form is a cooperative affair. When locking on an object, lock on the same object, otherwise the threads are synchronized on different objects, which means multiple threads might gain access to shared resources you assumed were protected. Also, lock on private field objects. In the chapter examples I locked on the list itself (e.g., _list). In a programming team environment you'll want it understood between all members upon what object within individual classes to synchronize. You may decide to define a private member object field within a class for the sole purpose of locking.

Use the C# `lock` keyword for convenience and if you don't need finer-grained thread synchronization control. You can, however, use the `lock` keyword in conjunction with the Monitor.Wait() and Monitor.Pulse() methods.

The single-argument Monitor.Enter() method is obsolete as of .NET 4.0. Going forward favor the use of the overloaded two-argument version which uses a boolean value to indicate whether or not the lock has been taken.

Other than that, this chapter has only presented and demonstrated a small sampling of the thread synchronization mechanisms available to you in the .NET platform. However, you can accomplish a lot with thread synchronization using the various methods of the Monitor class.

With regards to collections, the important thing to remember is that an exception will be thrown when attempting to enumerate a collection that is being simultaneously modified by another thread.

Thread Synchronization Usage Table

Table 14-1 lists and summarizes the thread synchronization mechanisms presented in this chapter.

Synchronization Primitive	Category	Usage	Comments

C# `lock` keyword	locking	`lock(_lockObject){`
		`  //critical section`
		`}`
			Translates into Monitor.Enter() and Monitor.Exit() calls under the covers.

Monitor class Monitor.Enter() Monitor.Exit() (basic usage)	locking	`Monitor.Enter(_lockObject);`
		`try{`
		`  //critical code section`
		`}catch(Exception e){`
		`  //exception handler code`
		`}finally{`
		`  Monitor.Exit(_lockObject);`
		`}`
			Obsolete as of .NET 4.0. (Source: Compiler warning csc version 4.0.21006.1)
			If a lock already exists on _lockObject the thread blocks until the lock on _lockObject is released.

Monitor class Monitor.Enter() Monitor.Exit() (overloaded method usage with lockTaken boolean argument)	locking	`bool lockTaken = false;`
		`try{`
		`  Monitor.Enter(_lockObject,`
		`  ref lockTaken);`
		`  if(lockTaken){`
		`  // do this if lock taken`
		`  }else{`
		`  // alternative processing`
		`  }`
		`}catch(Exception e){`
		
		`}finally{`
		`  if(lockTaken){`
		`  Monitor.Exit(_lockObject);`
		`  }`
		`}`
			Preferred use as of .NET 4.0. (Source: Compiler warning csc version 4.0.21006.1)
			If a lock already exists on _lockObject the thread blocks until the lock on _lockObject is released.
			The value of the lockTaken argument is set even if an exception is thrown when attempting to acquire the lock on _lockObject.

Table 14-1: Synchronization Primitives Reference Table

Synchronization Primitive	Category	Usage	Comments

Monitor class
Monitor.Enter()
Monitor.Exit()
(Fine grain control with Monitor.Wait() and Monitor.Pulse())
	locking	`public void MethodA(){`
		`  bool lockTaken = false;`
		`  try{`
		`  Monitor.Enter(_lockObject,`
		`  ref lockTaken);`
		`  if(lockTaken){`
		`  // do this if lock taken`
		`  // if resource not available`
		`  // block until it is...`
		`  Monitor.Wait(_lockObject);`
		`  // when lock reacquired`
		`  // continue processing`
		`  }else{`
		`  // alternative processing`
		`  }`
		`  }catch(Exception e){`
		
		`  }finally{`
		`  if(lockTaken){`
		`  Monitor.Exit(_lockObject);`
		`  }`
		`  }`
		`} // end MethodA()`
		
		`public void MethodB(){`
		`  bool lockTaken = false;`
		`  try{`
		`  Monitor.Enter(_lockObject,`
		`  ref lockTaken);`
		`  if(lockTaken){`
		`  // do this if lock taken`
		`  // if you want to relinquish`
		`  // the lock for a while...`
		`  Monitor.Pulse(lockObject);`
		`  Thread.Sleep(10);`
		`  // to give other threads`
		`  // a chance to execute`
		`  }else{`
		`  // alternative processing`
		`  }`
		`  }catch(Exception e){`
		
		`  }finally{`
		`  if(lockTaken){`
		`  Monitor.Exit(_lockObject);`
		`  }`
		`  }`
		`} // end MethodB()`
			The thread that currently owns the lock on an object calls Monitor.Wait(object) to relinquish the lock and block until it can reacquire the lock.
			Another thread must make a call to Monitor.Pulse(object) to signal blocked threads that are waiting on the lock object to move to the ready queue.
			Note: This is a cooperative scheme. If one thread calls Wait() without another thread's corresponding call to Pulse() then deadlock can occur because one thread is blocked indefinitely waiting for the other thread to signal it to move to the ready queue.

Table 14-1: Synchronization Primitives Reference Table

Synchronization Primitive	Category	Usage	Comments

Monitor class
Monitor.TryEnter()
Monitor.Exit()
	locking	public void MethodA(){
		  bool lockTaken = false;
		  try{
		  Monitor.TryEnter(_lockObject,
		  ref lockTaken);
		  if(lockTaken){
		  // do this if lock taken
		  // if resource not available
		  // block until it is...
		  Monitor.Wait(_lockObject);
		  // when lock reacquired
		  // continue processing
		  }else{
		  // alternative processing
		  }
		  }catch(Exception e){
		
		  }finally{
		  if(lockTaken){
		  Monitor.Exit(_lockObject);
		  }
		  }
		} // end MethodA()
		
		public void MethodB(){
		  bool lockTaken = false;
		  try{
		  Monitor.TryEnter(_lockObject,
		  ref lockTaken);
		  if(lockTaken){
		  // do this if lock taken
		  // if you want to relinquish
		  // the lock for a while...
		  Monitor.Pulse(lockObject);
		  Thread.Sleep(10);
		  // to give other threads
		  // a chance to execute
		  }else{
		  // alternative processing
		  }
		  }catch(Exception e){
		
		  }finally{
		  if(lockTaken){
		  Monitor.Exit(_lockObject);
		  }
		  }
		} // end MethodB()
			The Monitor.TryEnter() method does not block. It returns immediately

Table 14-1: Synchronization Primitives Reference Table

Synchronization Primitive	Category	Usage	Comments

MethodImplOptions. Synchronized Attribute	Contextual	`[MethodImpl(MethodImplOptions.Synchro-nized)]`
		`public void MethodName(){`
		`  // the entire method is synchronized`
		`}`
			Synchronizes the entire method.

Table 14-1: Synchronization Primitives Reference Table

Summary

The need for thread synchronization arises when multiple threads of execution may access shared resources or shared code segments, which, if unsynchronized, would destabilize the code or leave the code in an invalid state. The .NET framework and the C# language provide various thread synchronization primitives and strategies that enable you to synchronize thread access to critical code segments.

The C# `lock` keyword is the easiest way to protect critical code segments. Use the C# lock keyword to obtain a "lock" on an object. Place the code you want to protect within the body of the `lock` statement. **Recommendation: Lock on private field objects only.** Do not lock on the current instance (i.e. this). **Warning: Do not lock on value objects.** Value object are boxed into objects when used in a `lock` statement. Thus, multiple threads "locking" on the same value object will actually be acquiring locks on different objects.

Collection classes in the System.Collections namespace come equipped with the SyncRoot and IsSynchronized properties. These old-school collections also provided a static Synchronized() method which is used to transform an ordinary collection into a synchronized collection. And while individual collection methods may be synchronized, it was still not thread safe to enumerate over a collection. While you can still write good-quality thread-safe code using the SyncRoot property along with the `lock` keyword or the Monitor class, the use of these old-school properties, along with the Synchronized() method is best avoided. Besides, unless you find yourself maintaining legacy C# code, you should be favoring the use of the generic collection classes.

The static Monitor class allows you to implement fine grained thread synchronization. You must be sure that for each call to Monitor.Enter(_lockObject) is followed by a call to Monitor.Exit(_lockObject). Failure to do so may result in deadlock as waiting threads will never acquire an unreleased lock. The critical code section begins with a call to Monitor.Enter(). Place the call to Monitor.Exit() in the body of the `finally` clause of a `try/catch/finally` block. **The Monitor.Enter() method blocks until it acquires the lock.** The Monitor.Enter() method is overloaded. Favor the use of the two-argument version of Monitor.Enter() going forward.

The Monitor.TryEnter() method is a non-blocking method that returns immediately after its called regardless of whether or not the lock is acquired. You must take this immediate return behavior into account in your code. Use the overloaded two-argument version of the Monitor.TryEnter() method to test whether or not the lock was acquired.

If a thread needs to give up the lock because it has nothing to do, call the Monitor.Wait() method. To signal waiting threads of a change in lock status, call the Monitor.Pulse() method to move the next waiting thread into the ready queue.

Use the [MethodImpl(MethodImplOptions.Synchronized)] attribute to synchronize entire methods. However, I recommend using this attribute sparingly. Generally speaking, the finer grained you can make your thread synchronization scheme, the better off you'll be.

Reference

Microsoft Developer Network (MSDN) *.NET Framework 3.0, 3.5, and 4.0 Reference Documentation* [www.msdn.com]

Microsoft *Shared Source Common Language Infrastructure 2.0* Release (SSCLI 2.0)(Codename: Rotor)[http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en]

Notes

