Chapter 16

Nikon F3HP / Zoom-Nikkor 35-105 / Kodak Tri-X

Operation – USS Norfolk SSN 714

Multithreaded Programming

Learning Objectives

- *Define the terms "process" and "thread"*
- *Explain the difference between a process and a thread*
- *Explain how time-slicing works*
- *State the definition of the term "multithreading"*
- *List and describe the steps and components required to create a multithreaded program*
- *Create and start threads in a program with the Thread class*
- *Pass arguments to managed threads using ParameterizedThreadStart delegates*
- *Explain the difference between foreground and background threads*
- *Use the BackgroundWorker class to create multithreaded programs*
- *Use delegates to make asynchronous method calls*
- *Pass arguments to and obtain results from asynchronous method calls*
- *Create background threads using the ThreadPool class*
- *Demonstrate your ability to create multithreaded programs*

Introduction

It's time now to add another invaluable tool to your programming toolbelt, and that is the ability to write multi-threaded programs. While the term *multithreaded programming* may sound complicated, it is in reality quite easy to do in C# .NET.

In this chapter, I will explain how multithreading works on your typical, general-purpose computer. You'll learn about the relationship between a process and its threads and how an operating system manages thread execution. I'll then show you how to use the Thread class in your programs to create and start managed threads. Next, I'll show you how you can simplify the creation and management of multiple threads with the help of the BackgroundWorker class. Also, I'll show you how to use ThreadPool threads and how to run any method asynchronously with the help of delegates.

As is the case with any tool, there's a right way to use it and a wrong way. Multithreading, applied thoughtlessly, will render your programs overly complicated, sluggish, unresponsive, and buggy. But, when used with care, multi-threading can significantly increase your application's performance, giving it that hard-to-describe-but-you-know-it-when-you-see-it feeling of professionalism.

Before getting started I need to add a caveat. While I present a lot of material in this chapter, I make no attempt to cover all aspects of multithreaded programming. To do so would bore you to death, and in fact, as it turns out, a lot of what you can do with threads you shouldn't do. Instead, I will focus on those topics that give you a lot of bang for your buck to get you up and running as quickly as possible with multithreaded programming. In some cases, I have postponed the discussion of more obscure threading topics until later in the book where their presentation is more appropriate.

As usual, I recommend that if you want to dive deeper into threads and multithreaded programming consult one of the excellent references I've listed at the end of the chapter.

Multithreading Overview: The Tale Of Two Vacations

As I write this it's approaching the end of October in Northern Virginia. I always get a hankering to go on a vacation right about now. Let's take a look at the concept of vacation from a thread's point of view.

Single-Threaded Vacation

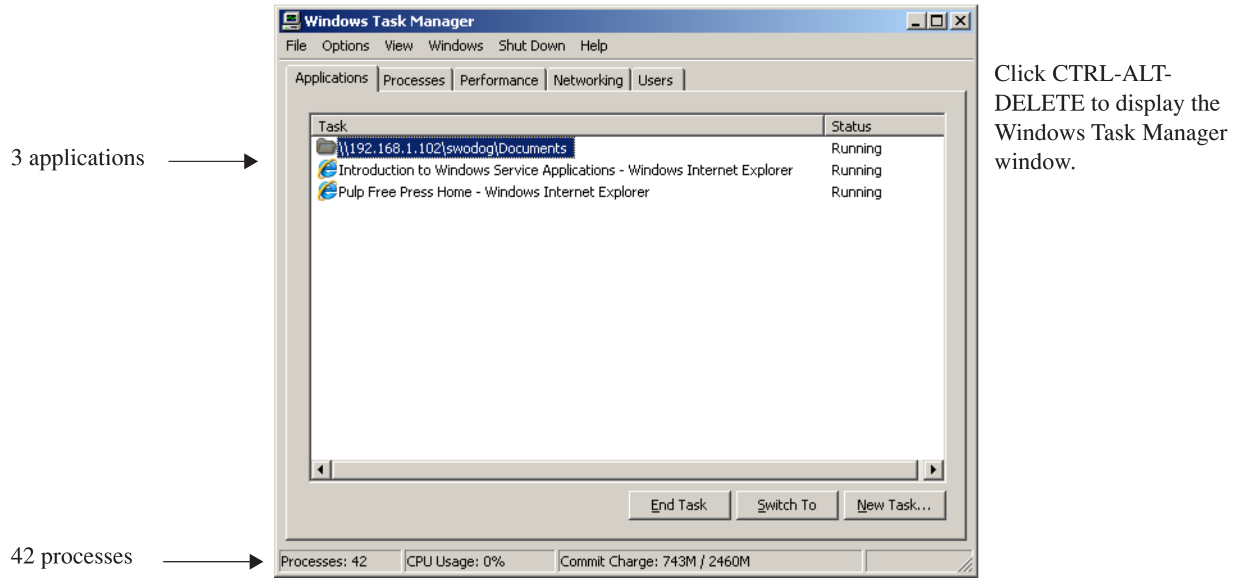
Imagine for a moment you're on vacation, trying to relax on the squeaky white sand of a sun-drenched tropical beach. Your job, since you are on vacation, is to relax, and as you start to drift off for a snooze you get thirsty. You are the only one on the beach and the bar is a mile away! You get up and walk to the bar and buy a drink, no, better make that two drinks, and walk back to your lounge chair. Now you start to relax again, until you get hungry. The grill is a mile in the other direction, so you get up again and walk to the grill. What you really want to do is relax and enjoy the beach, but what you ended up doing was a little relaxing, some drink fetching, and some food hunting. Eventually you'll get back to relaxing. After all, you're on single-threaded vacation.

Multithreaded Vacation

Now imagine that you're on multithreaded vacation. Again, your job is to relax on the beach. This time, however, when you get thirsty, you ask the wait staff to please bring you a drink, which they immediately set out to do, while you immediately return to relaxing. When you get hungry, you again summon the wait staff and off they go to fetch you a little somethin' somethin' from the grill. You immediately return to relaxing. Multithreaded vacation is so much better! If you've ever returned from a vacation and felt like you needed a vacation, you probably didn't take a multi-threaded vacation because you were never allowed to relax!

The Relationship Between A Process And Its Threads

In the tale of two vacations above, you could think of yourself as being a process: the “Relax” process. On a computer, the operating system loads and starts services and applications. Each service or application runs as a separate *process* on the machine. (**Note:** A *service* is a special type of Windows application that runs solely in the background with limited or no user interaction.) Figure 16-1 shows a list of applications running on my machine as I write these words. Figure 16-2 shows a list of processes.


3 applications
42 processes
Click CTRL-ALT-DELETE to display the Windows Task Manager window.
Windows Task Manager
File Options View Windows Shut Down Help
Applications Processes Performance Networking Users
End Task Switch To New Task...

Task	Status
\\192.168.1.102\swodog\Documents	Running
Introduction to Windows Service Applications - Windows Internet Explorer	Running
Pulp Free Press Home - Windows Internet Explorer	Running

Processes: 42 CPU Usage: 0% Commit Charge: 743M / 2460M

Figure 16-1: List of Running Applications

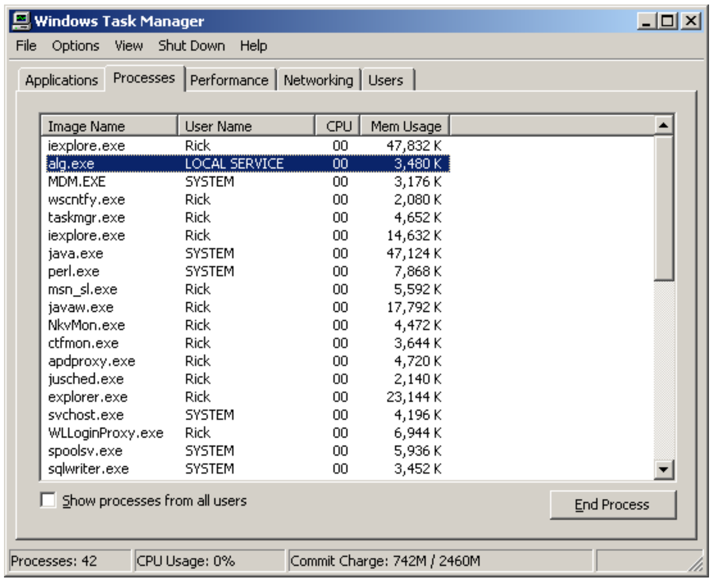

Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Show processes from all users
End Process

Image Name	User Name	CPU	Mem Usage
iexplore.exe	Rick	00	47,832 K
alg.exe	LOCAL SERVICE	00	3,480 K
MDM.EXE	SYSTEM	00	3,176 K
wscntfy.exe	Rick	00	2,080 K
taskmgr.exe	Rick	00	4,652 K
iexplore.exe	Rick	00	14,632 K
java.exe	SYSTEM	00	47,124 K
perl.exe	SYSTEM	00	7,868 K
msn_sl.exe	Rick	00	5,592 K
javaw.exe	Rick	00	17,792 K
NkvMon.exe	Rick	00	4,472 K
ctfmon.exe	Rick	00	3,644 K
apdproxy.exe	Rick	00	4,720 K
jusched.exe	Rick	00	2,140 K
explorer.exe	Rick	00	23,144 K
svchost.exe	SYSTEM	00	4,196 K
WLLoginProxy.exe	Rick	00	6,944 K
spoolsv.exe	SYSTEM	00	5,936 K
sqlwriter.exe	SYSTEM	00	3,452 K

Processes: 42 CPU Usage: 0% Commit Charge: 742M / 2460M

Figure 16-2: Partial List of Processes Running on the Same Computer

Referring to figures 16-1 and 16-2 — there are quite a few more processes actually running than there are applications. Many of the processes are background operating system processes that are started automatically when the computer powers up. There are two databases running: MS/SQL Server Express and Oracle 10G. You can also see in the process list that Java (java.exe) and a Perl interpreter (perl.exe) are running along with the Windows Desktop Explorer (explorer.exe) and two instances of Internet Explorer (iexplore.exe). Each one of these processes is isolated from the others meaning that each process has an allocated memory space all to itself. The management of this process memory space is left to the operating system.

A process consists of one or more *threads of execution*, referred to simply as *threads*. A process always consists of at least one thread, the *Main thread*, (the Main() method's thread of execution) which starts running when the process begins execution. A *single-threaded process* contains only one thread of execution. A *multithreaded process* contains more than one thread. Figure 16-3 offers a representation of processes and threads in a single-processor system.

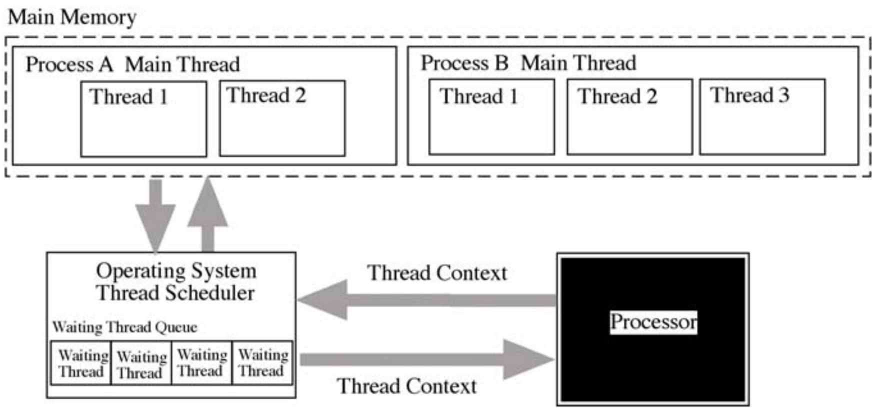


Figure 16-3: Processes and their Threads Executing in a Single-Processor Environment

Referring to Figure 16-3 — two processes A and B are executing. Process A contains three threads: Main, Thread 1 and Thread 2. Process B contains four threads: Main, Thread 1, Thread 2, and Thread 3. A thread is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and *application domain*.

As you can see in Figure 16-3, the operating system thread scheduler coordinates thread execution. Waiting threads sit in a thread queue until they are loaded into the processor. Each thread has a data structure known as a *thread context*. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system, the operating system allocates processor time with a *time-slicing* scheme. Each thread gets a little bit of time to execute before being *preempted* by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor. This diagram makes clear that in a single-processor system, the notion of concurrently executing applications is just an illusion pulled off by the operating system quickly switching threads in and out of the processor. Figure 16-4 shows how things might look on a multiprocessor system. Referring to Figure 16-4 — now we can really get some work done. In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

Returning once again to my earlier vacation analogy, when you're on single-threaded vacation, the relax process does everything related to the vacation in one thread of execution. That's why you must stop relaxing and fetch yourself a drink and something to eat. When you're on multithreaded vacation, the relax process concentrates on relaxing and hands off the chores of drink and food fetching to separate threads, You come away from a multithreaded vacation feeling a lot more relaxed! (*Well, at least until you arrive at the airport anyway.*)

Vacation Gone Bad

There is a possibility, even on multithreaded vacation, for you to return home tense and frustrated. This can occur if the drink and food fetching threads misbehave. How might this happen? Assume for a moment, if you will, that you are not the only process on the beach. Laying next to you is a nasty little someone named "create-hate-and-discontent". He told his food and drink fetching threads they were special and gave them an order to cut in front of the line whenever possible. Your threads get booted from the bar and grill counters more frequently because of the higher priority of create-hate-and-discontent's threads. You suffer because your threads take longer to fetch food and drink. This is only one example of how ill-behaved threads can bring one or more processes to a halt.

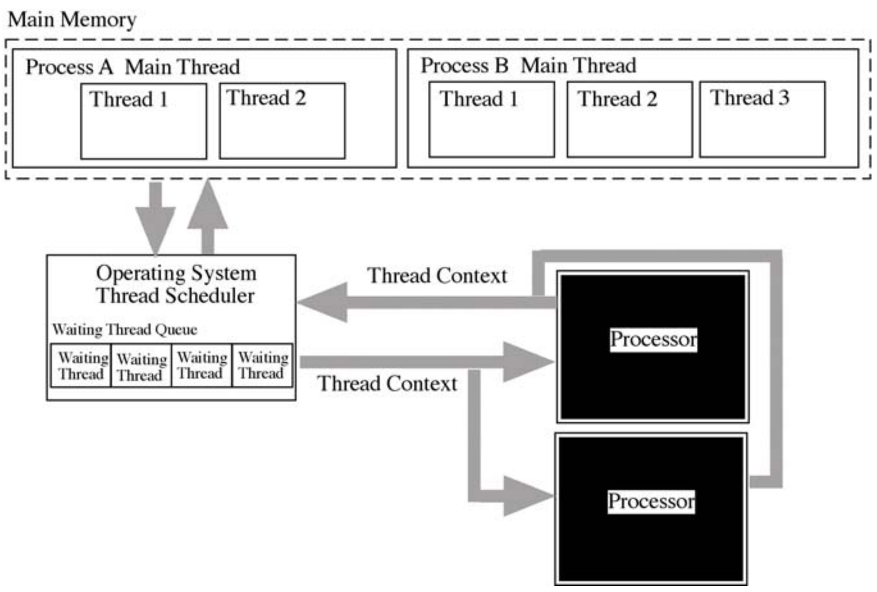


Figure 16-4: Processes and their Threads Executing in a Multiprocessor Environment

Quick Review

A process consists of one or more threads of execution, referred to simply as threads. A process always consists of at least one thread, the Main thread, which starts running when the process begins execution. A single-threaded process contains only one thread of execution. A multithreaded process contains more than one thread.

A thread is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and application domain.

Waiting threads sit in a thread queue until they are loaded into the processor. Each thread has a data structure known as a thread context. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system the operating system allocates processor time with a time-slicing scheme. Each thread gets a little bit of time to execute before being preempted by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor.

In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

Creating Managed Threads With The Thread Class

In this section I will show you how to use the Thread class to create and manage the execution of threads in your programs. You'll find the Thread class, along with a whole lot of other useful stuff, in the System.Threading namespace. The Thread class allows you to create what are referred to as *managed threads*. They are called managed threads because you can directly manipulate each thread you create. You gain a lot of flexibility and power when you manage your own threads. However, with power and flexibility comes the responsibility of ensuring your threads behave well and properly handle exceptional conditions that may arise during their execution lifetime. This aspect of thread management gained increased importance in .NET 2.0 because, in most cases, unhandled exceptions lead to application termination.

The material in this section lays the foundation for the rest of the chapter. Once you understand the issues involved with creating and managing your own threads, you'll better understand why, in most cases, it's a good idea to let the runtime environment manage threads for you. However, before getting started, let's see just how little relaxing one does while on single-threaded vacation.

Single-Threaded Vacation Example

How might the single-threaded vacation analogy be implemented in source code? Example 16.1 offers one possible solution.

16.1 SingleThreadedVacation.cs

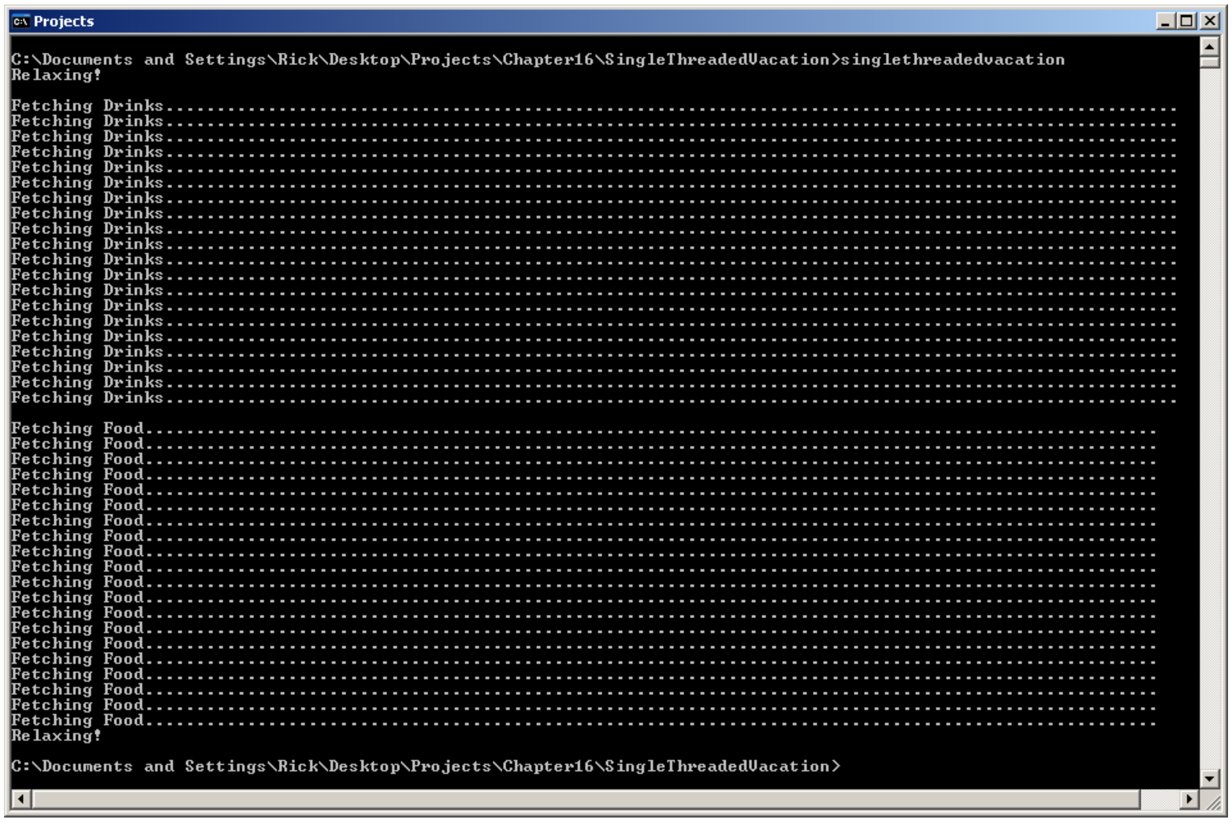
```
1    using System;
2
3    public class SingleThreadedVacation {
4
5      private bool hungry;
6      private bool thirsty;
7
8      public SingleThreadedVacation(){
9        hungry = true;
10       thirsty = true;
11     }
12
13     public void FetchDrink(){
14       int steps_to_the_bar = 1000;
15       for(int i=0; i<steps_to_the_bar*2; i++){
16       if((i%100) == 0){
17         Console.WriteLine();
18         Console.Write("Fetching Drinks");
19       }else{
20         Console.Write(".");
21       }
22      }
23      Console.WriteLine();
24      thirsty = false;
25     }
26
27     public void FetchFood(){
28       int steps_to_the_grill = 1000;
29       for(int i=0; i<steps_to_the_grill*2; i++){
30       if((i%100)==0){
31         Console.WriteLine();
32         Console.Write("Fetching Food");
33       }else{
34         Console.Write(".");
35       }
36      }
37      Console.WriteLine();
38      hungry = false;
39     }
40
41     public static void Main(){
42       SingleThreadedVacation stv = new SingleThreadedVacation();
43       Console.WriteLine("Relaxing!");
44       while(stv.hungry && stv.thirsty){
45       stv.FetchDrink();
46       stv.FetchFood();
47       Console.WriteLine("Relaxing!");
48      }
49     }
50   }
```

Referring to Example 16.1 — the SingleThreadedVacation class contains two fields: hungry and thirsty, of type bool, which are initially set to true. It has two methods: FetchDrink() and FetchFood(). When each method is called, the `for` loop contained in each kills some time by "walking" the number of steps to the bar or grill and back again. Each method prints to the console a status message every 100 steps it takes.

The Main() method starting on line 41 starts by printing a message to the console saying it's "Relaxing!". It then enters the while loop where calls are made to FetchDrink() and FetchFood(). Since the whole program executes in a single thread of execution (*i.e.*, the Main() method's thread,) the FetchDrink() method must run to conclusion before the call to FetchFood() can be made. The FetchFood() method must then execute and return before the message "Relaxing!" can again be printed to the screen. Figure 16-5 shows SingleThreadedVacation in action.

MultiThreaded Vacation Example

Let's now see how much more relaxing you can do on a multithreaded vacation. Example 16.2 gives the code for the MultiThreadedVacation class.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\SingleThreadedVacation>singlethreadedvacation
Relaxing!

Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................
Fetching Drinks...................................................................................................

Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Fetching Food...................................................................................................
Relaxing!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\SingleThreadedVacation>
```

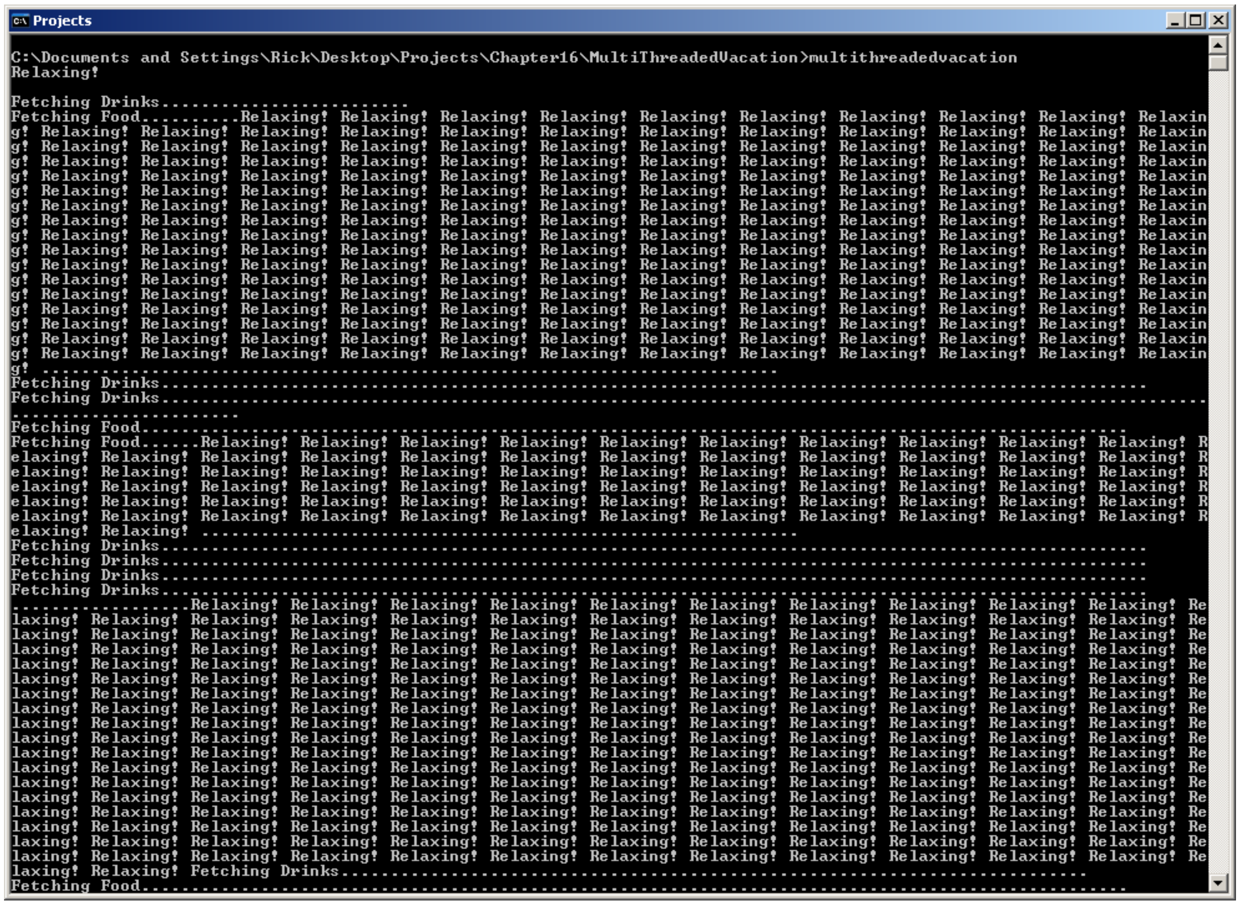
Figure 16-5: SingleThreadedVacation Program Output

16.2 MultiThreadedVacation.cs

```
using System;
using System.Threading;

public class MultiThreadedVacation {

  private bool hungry;
  private bool thirsty;

  public MultiThreadedVacation(){
    hungry = true;
    thirsty = true;
  }

  public void FetchDrink(){
    int steps_to_the_bar = 1000;
    for(int i=0; i<steps_to_the_bar*2; i++){
    if((i%100) == 0){
      Console.WriteLine();
      Console.Write("Fetching Drinks");
    }else{
      Console.Write(".");
    }
   }
   Console.WriteLine();
   thirsty = false;
  }

  public void FetchFood(){
    int steps_to_the_grill = 1000;
    for(int i=0; i<steps_to_the_grill*2; i++){
    if((i%100)==0){
      Console.WriteLine();
      Console.Write("Fetching Food");
    }else{
      Console.Write(".");
    }
   }
   Console.WriteLine();
39      hungry = false;
40     }
41
42     public static void Main(){
43       MultiThreadedVacation mtv = new MultiThreadedVacation();
44      Thread drinkFetcher = new Thread(mtv.FetchDrink);
45      Thread foodFetcher = new Thread(mtv.FetchFood);
46      Console.WriteLine("Relaxing!");
47
48      while(mtv.hungry && mtv.thirsty){
49        if(!drinkFetcher.IsAlive) drinkFetcher.Start();
50        if(!foodFetcher.IsAlive) foodFetcher.Start();
51        Console.Write("Relaxing!");
52     }
53     }
54   }
```

Referring to Example 16.2 — this code is structurally very similar to the previous example. The only changes made were to the insides of the Main() method where two thread objects are created on lines 44 and 45 named drinkFetcher and foodFetcher respectively. Note that to create a thread in this fashion, you supply to the Thread constructor the name of a method you want to execute in the separate thread. (Here the method signatures conform to the ThreadStart delegate signature.) The drinkFetcher thread executes the FetchDrink() method while the foodFetcher thread executes the FetchFood() method.

A check is made in the body of the `while` loop to see if each thread is alive, meaning “Has it been started?” If not, it is started by calling its Thread.Start() method. As soon as these threads are started, the Main() thread can go back to printing the message “Relaxing!” to the console. Figure 16-6 shows a partial listing of the MultiThreadedVacation program’s output. As you’ll note from looking at Figure 16-6 there’s a lot more relaxing going on!



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\MultiThreadedVacation>multithreadedvacation
Relaxing!

Fetching Drinks.........................
Fetching Food..........Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxin
g! .........................................................................
Fetching Drinks.................................................................................................
Fetching Drinks.........................................................................................................
......................
Fetching Food.................................................................................................
Fetching Food......Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! R
elaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! R
elaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! R
elaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! R
elaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! R
elaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! R
elaxing! Relaxing! ...........................................................
Fetching Drinks.................................................................................................
Fetching Drinks.................................................................................................
Fetching Drinks.................................................................................................
Fetching Drinks.................................................................................................
..................Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Relaxing! Re
laxing! Relaxing! Fetching Drinks.........................................................................
Fetching Food.................................................................................................
```

Figure 16-6: MultiThreadedVacation Program Output - Partial Listing

Thread States

A thread can assume several different states during its execution lifetime, as shown in Figure 16-7.

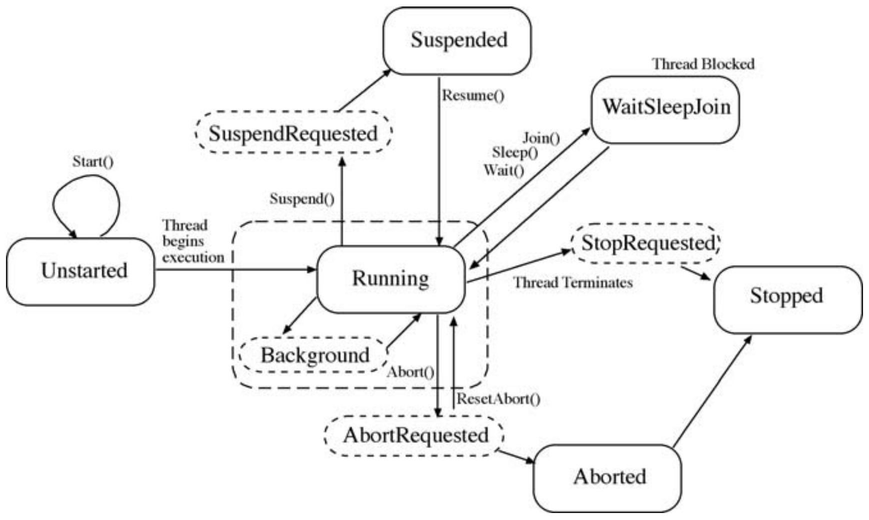


Figure 16-7: Thread States and Transition Initiators

Referring to Figure 16-7 — important points to note include the following: A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() only notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in, or more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are now obsolete.

So where does that leave you with regards to managing your own threads? Well, you can start a thread with the Start() method and block its operation with the Monitor.Wait(), Thread.Sleep() and Thread.Join() methods. You can change a foreground thread into a background thread by setting its IsBackground property to true. As it turns out, this amount of control is really all you need to write well-behaved, multithreaded code. The following sections discuss and demonstrate the use of the more helpful Thread properties and methods.

Creating And Starting Managed Threads

To create a managed thread, pass in to the Thread constructor either a *ThreadStart* delegate or a *Parameter-izedThreadStart* delegate. The ParameterizedThreadStart delegate lets you pass an argument object when you call the thread's Start() method.

ThreadStart Delegate

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass a ThreadStart delegate into the Thread constructor: the *longhand* way and the *shorthand* way. The long-hand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

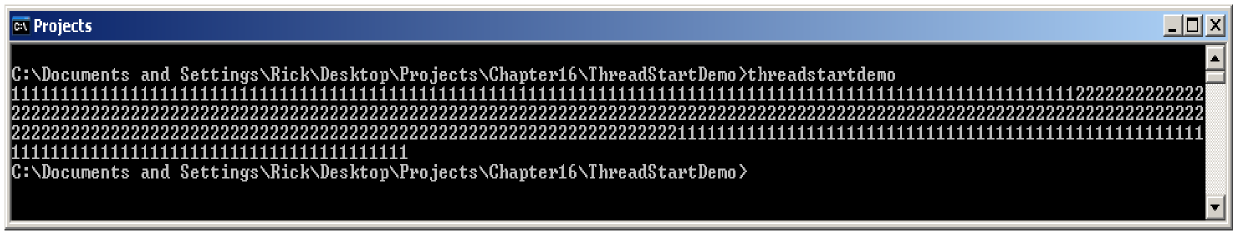
```
Thread thread2 = new Thread(Run); // shorthand
```

Example 16.3 demonstrates the longhand and shorthand way of creating threads.

16.3 ThreadStartDemo.cs

```
using System;
using System.Threading;

public class ThreadStartDemo {

  private const int COUNT = 200;

  public static void Run(){
    for(int i=0; i<COUNT; i++){
     Console.Write(Thread.CurrentThread.Name);
   }
  }

  public static void Main(){
    Thread thread1 = new Thread(new ThreadStart(Run)); // longhand way
    Thread thread2 = new Thread(Run); // shorthand way
    thread1.Name = "1";
    thread1.Start();
    thread2.Name = "2";
    thread2.Start();
  }
}
```

Referring to Example 16.3 — two thread objects are created in the Main() method. The first, thread1, is created the longhand way by passing the name of the Run() method to the ThreadStart constructor. The second, thread2, is created the shorthand way by passing the name of the Run() method directly to the Thread constructor. Each thread's Name property is set before calling its Start() method. The name of the thread is printed to the console in the body of the Run() method. Note that in this example the Run() method is static, but it could just as well have been an instance method. Figure 16-8 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ThreadStartDemo>threadstartdemo
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111112222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222221111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ThreadStartDemo>
```

Figure 16-8: Results of Running Example 16.3

ParameterizedThreadStart Delegate: Passing Arguments To Threads

If you need to pass in an argument when you start a thread, your thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

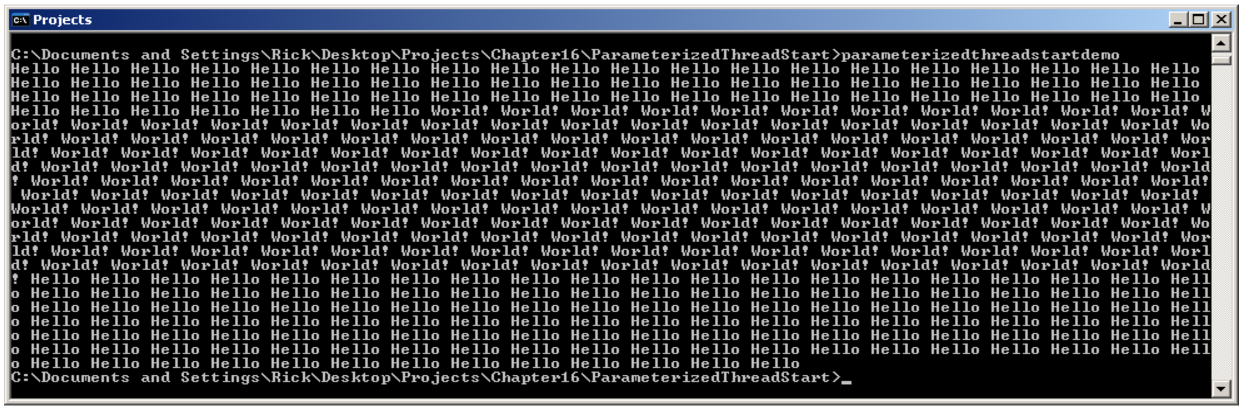
```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Example 16.4 shows the ParameterizedThreadStart delegate in action.

16.4 ParameterizedThreadStartDemo.cs

```
using System;
using System.Threading;

public class ParameterizedThreadStartDemo {

  private const int COUNT = 200;

  public static void Run(object value){
    for(int i=0; i<COUNT; i++){
     Console.Write(value);
   }
  }

  public static void Main(){
    Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
    Thread thread2 = new Thread(Run); // shorthand way
    thread1.Start("Hello ");
    thread2.Start("World! ");
  }
}
```

Referring to Example 16.4 — The static Run() method has been modified to conform to the ParameterizedThreadStart delegate method signature. In this case I am passing the parameter named "value" directly to the Console.Write() method, which will automatically call the Object.ToString() method. (**Note:** Here I'm only targeting the interface as specified by the Object class. If I expect some other type of object I must cast to the expected type.) Pass the argument to the thread when you call its Start() method, as is shown on lines 17 and 18. Figure 16-9 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>parameterizedthreadstartdemo
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
Hello Hello Hello Hello Hello Hello Hello World! World! World! World! World! World! World! World! World! World! World! W
orld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wo
rld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wor
ld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Worl
d! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World
! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World!
 World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World!
World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! W
orld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wo
rld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Wor
ld! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! Worl
d! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World! World
! Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hell
o Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>_
```

Figure 16-9: Results of Running Example 16.4

Blocking A Thread With Thread.Sleep()

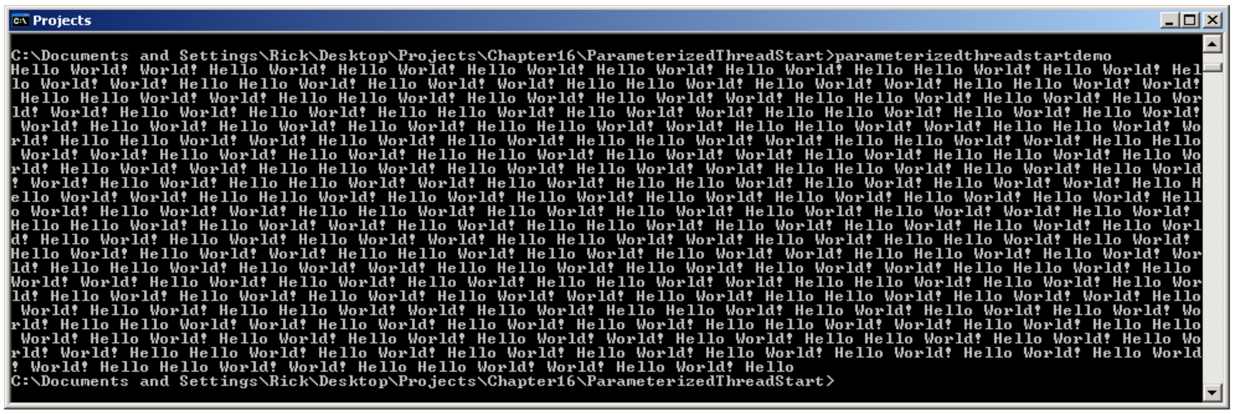
If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system preempts the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Take a good look at Figure 16-9 and you'll see how thread1 prints the message "Hello" over and over until it's swapped out with thread2, which then starts to print "World!".

In many situations, you'll want a thread to do something and then take a short break to let other threads have a go at the processor. Example 16.5 adds a call to Thread.Sleep() to the body of the Run() method.

16.5 ParameterizedThreadStart.cs (With call to Sleep())

```
 using System;
 using System.Threading;

 public class ParameterizedThreadStartDemo {

   private const int COUNT = 200;

   public static void Run(object value){
     for(int i=0; i<COUNT; i++){
      Console.Write(value);
      Thread.Sleep(10);
    }
   }

   public static void Main(){
     Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
    Thread thread2 = new Thread(Run); // shorthand way
    thread1.Start("Hello ");
    thread2.Start("World! ");
   }
 }
```

Referring to Example 16.5 — the call to Thread.Sleep() is made after the value is written to the console. Pass an integer argument to the Sleep() method indicating the time in milliseconds you want the thread to block. You can also pass in a TimeSpan object. Figure 16-10 shows the results of running this program. Note how different the output appears and how much slower the application seems to run because of the increased thread swapping that occurs.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>parameterizedthreadstartdemo
Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hel
lo World! World! Hello Hello World! Hello World! World! Hello Hello World! Hello World! World! Hello Hello World! World!
 Hello Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello World! Hello Wor
ld! World! Hello World! Hello World! Hello Hello World! Hello World! World! Hello Hello World! Hello World! Hello World!
 World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! World! Hello Hello
 World! World! Hello World! Hello World! Hello Hello World! Hello World! World! Hello World! Hello Hello World! Hello Wo
rld! Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello World! Hello World
! World! Hello World! Hello Hello World! World! Hello World! Hello Hello World! Hello World! Hello World! World! Hello H
ello World! World! Hello Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hell
o World! Hello World! World! Hello Hello World! Hello World! Hello World! Hello World! Hello World! World! Hello World!
Hello Hello World! Hello World! World! Hello World! Hello Hello World! Hello World! Hello World! Hello World! Hello Worl
d! Hello World! Hello World! Hello World! World! Hello Hello World! World! Hello World! Hello Hello World! Hello World!
Hello World! Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello World! Wor
ld! Hello Hello World! Hello World! World! Hello Hello World! Hello World! Hello World! World! Hello Hello World! Hello
World! World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! Hello World! Hello Wor
ld! Hello World! Hello World! Hello World! Hello World! World! Hello World! Hello Hello World! Hello World! World! Hello
 World! Hello World! Hello Hello World! World! Hello World! Hello Hello World! Hello World! Hello World! Hello World! Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello Hello
 World! Hello World! Hello World! Hello World! Hello World! World! Hello World! Hello Hello World! Hello World! Hello Wo
rld! World! Hello Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World
! World! Hello Hello World! World! Hello Hello World! World! Hello World! Hello
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ParameterizedThreadStart>
```

Figure 16-10: Results of Running Example 16.5

Blocking A Thread With Thread.Join()

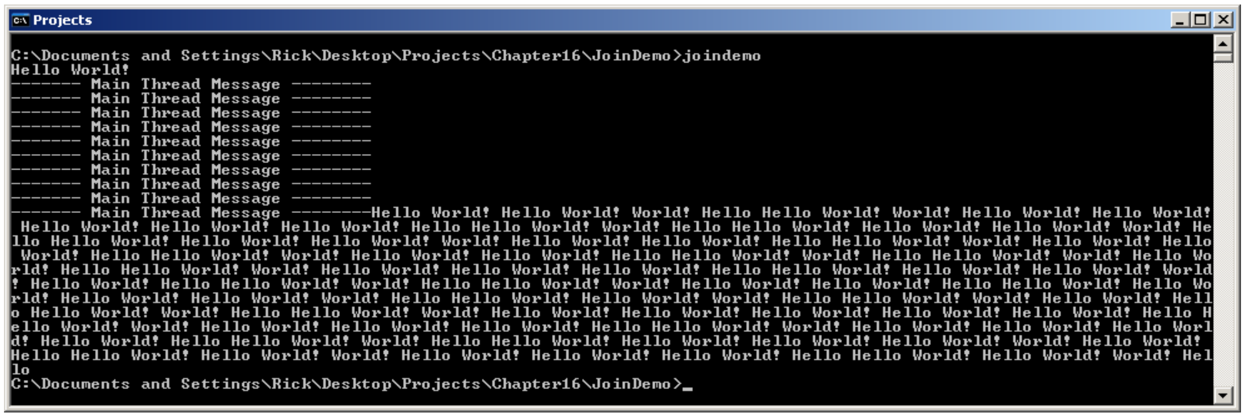
Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution, then in the Main thread you would call thread2.Join(). I want to show you two examples to demonstrate the use of the Join() method. The first, Example 16.6, builds on the previous example and adds a `for` loop in the Main() method that prints a message to the console. I've put the call to the thread2.Join() in the body of the `for` loop but it's commented out in this example.

16.6 JoinDemo.cs (Version 1)

```
1    using System;
2    using System.Threading;
3
4    public class JoinDemo {
5
6      private const int COUNT = 100;
7
8      public static void Run(object value){
9        for(int i=0; i<COUNT; i++){
10        Console.Write(value);
11        Thread.Sleep(10);
12      }
13     }
14
15     public static void Main(){
16       Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
17      Thread thread2 = new Thread(Run); // shorthand way
18      thread1.Start("Hello ");
19      thread2.Start("World! ");
20      for(int i = 0; i< 10; i++){
21        Console.Write("\n------- Main Thread Message --------");
22       //if(i==1) thread2.Join();
23     }
24     }
25   }
```

Referring to Example 16.6 — I've added a `for` loop to the end of the Main() method that loops ten times printing a message to the console. I've commented out line 22 for now so you can compare the output of this program with the output of the next example. Figure 16-11 shows the results of running this program. Referring to Figure 16-11 — note how thread1 and thread2 each print a message before sleeping. When the Main thread gets its chance to execute, it runs to completion.

Example 16.7 gives the JoinDemo program with line 22 in action. Figure 16-12 shows the results of running the program. Note the difference in the output between figures 16-11 and 16-12. The `for` loop in the Main thread makes it through two loops before being told to block until thread2 completes execution. (*i.e.*, thread2.Join())

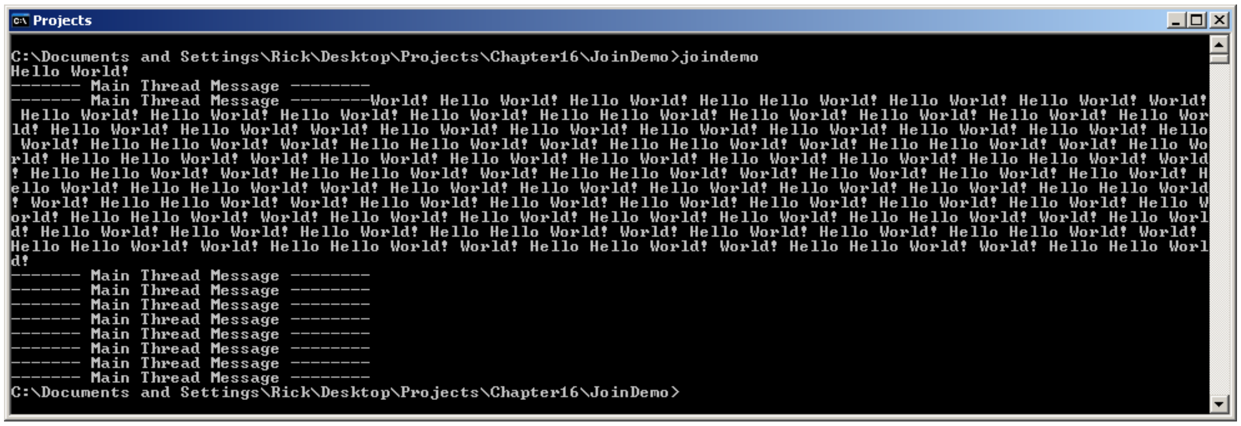


```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>joindemo
Hello World!
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------Hello World! Hello World! World! Hello Hello World! World! Hello World! Hello World!
 Hello World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! Hello World! Hello World! World! He
llo Hello World! Hello World! Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello
 World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello Hello World! World! Hello Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hello World! World
! Hello World! Hello Hello World! World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! Hello Wo
rld! Hello World! Hello World! World! Hello Hello World! Hello World! World! Hello Hello World! World! Hello World! Hell
o Hello World! World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello H
ello World! World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello World! Hello Worl
d! Hello World! Hello Hello World! World! Hello Hello World! Hello World! Hello World! Hello World! World! Hello World!
Hello Hello World! Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! World! Hel
lo
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>_
```

Figure 16-11: Results of Running Example 16.6

16.7 JoinDemo.cs (Version 2)

```
using System;
using System.Threading;

public class JoinDemo {

  private const int COUNT = 100;

  public static void Run(object value){
    for(int i=0; i<COUNT; i++){
    Console.Write(value);
    Thread.Sleep(10);
   }
  }

  public static void Main(){
    Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
    Thread thread2 = new Thread(Run); // shorthand way
    thread1.Start("Hello ");
    thread2.Start("World! ");
    for(int i = 0; i< 10; i++){
      Console.Write("\n------- Main Thread Message --------");
      if(i==1) thread2.Join(); // the Main thread will block on thread2 after second loop
   }
  }
}
```



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>joindemo
Hello World!
------- Main Thread Message --------
------- Main Thread Message --------World! Hello World! Hello World! Hello Hello World! Hello World! Hello World! World!
 Hello World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hello World! Hello World! Hello Wor
ld! Hello World! Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello
 World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Wo
rld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! World
! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello World! Hello World! Hello World! H
ello World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello Hello World
! World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello W
orld! Hello Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! World! Hello Worl
d! Hello World! Hello World! Hello World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World!
Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Hello World! World! Hello Hello Worl
d!
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
------- Main Thread Message --------
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\JoinDemo>
```

Figure 16-12: Results of Running Example 16.7

Foreground vs. Background Threads

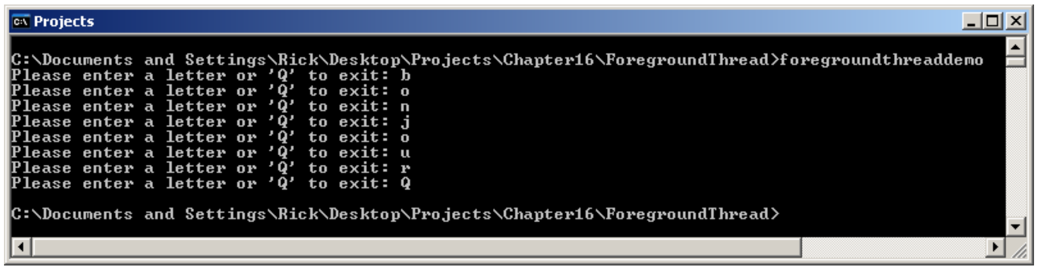
A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread will keep the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Managed threads are created as foreground threads. Example 16.8 gives an example of a foreground thread.

16.8 ForegroundThreadDemo.cs

```
using System;
using System.Threading;

public class ForegroundThreadDemo {

   public static void Run(){
     bool keepgoing = true;
     while(keepgoing){
       Console.Write("Please enter a letter or 'Q' to exit: ");
       String s = Console.ReadLine();
         switch(s[ 0] ){
           case 'Q': keepgoing = false;
                     break;
           default: break;
        }
      }
    }

   public static void Main(){
     Thread thread1 = new Thread(Run);
     thread1.Start();
   }
 }
```

Referring to Example 16.8 — the Main() method exits right after calling thread1.Start(). The Run() method loops continuously reading input from the console until the user enters the letter ‘Q’. Since thread1 is a foreground thread, it keeps the .NET runtime running as long as it’s executing. Figure 16-13 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ForegroundThread>foregroundthreaddemo
Please enter a letter or 'Q' to exit: b
Please enter a letter or 'Q' to exit: o
Please enter a letter or 'Q' to exit: n
Please enter a letter or 'Q' to exit: j
Please enter a letter or 'Q' to exit: o
Please enter a letter or 'Q' to exit: u
Please enter a letter or 'Q' to exit: r
Please enter a letter or 'Q' to exit: Q

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ForegroundThread>
```

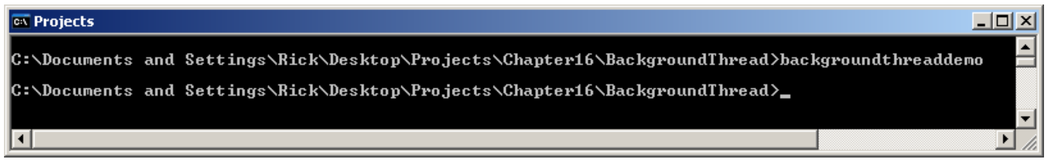
Figure 16-13: Results of Running Example 16.8

To change a foreground thread to a background thread, set the thread’s IsBackground property to true. Example 16.9 provides a slight modification to the previous example and makes thread1 a background thread.

16.9 BackgroundThreadDemo.cs

```
using System;
using System.Threading;

public class BackgroundThreadDemo {

   public static void Run(){
     bool keepgoing = true;
     while(keepgoing){
       Console.Write("Please enter a letter or 'Q' to exit: ");
       String s = Console.ReadLine();
       switch(s[ 0] ){
         case 'Q': keepgoing = false;
                   break;
         default: break;
       }
     }
   }

  public static void Main(){
    Thread thread1 = new Thread(Run);
    thread1.IsBackground = true;
    thread1.Start();
  }
}
```

Referring to Example 16.9 — on line 21, thread1's IsBackground property is set to true. Its Start() method is called on the next line and the Main() method exits. Thus, thread1 is stopped along with the .NET runtime execution environment. Figure 16-14 shows the very brief results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\BackgroundThread>backgroundthreaddemo

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\BackgroundThread>_
```

Figure 16-14: Results of Running Example 16.9

Quick Review

A thread can assume several different states during its execution lifetime. These states include: *Unstarted*, *Running*, *Background*, *SuspendRequested*, *Suspended*, *WaitSleepJoin*, *StopRequested*, *Stopped*, *AbortRequested*, and *Aborted*.

A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() only notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in or, more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are now obsolete.

To create a managed thread, pass to the Thread constructor either a ThreadStart delegate or a ParameterizedThreadStart delegate.

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass the ThreadStart delegate to the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

If you need to pass in an argument when you start a thread, the thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Pass the argument to the thread via its Start() method. Remember to cast the argument to the appropriate type in the body of the thread's execution method.

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system *preempts* the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Call the Thread.Sleep() method to force your thread to block and give other threads a chance to execute.

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution, then in the Main thread you would call thread2.Join().

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread will keep the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Creating Threads With The BackgroundWorker Class

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.*BackgroundWorker* class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads.

The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. Example 16.10 shows the BackgroundWorker class in action. This program displays a small window with three buttons and three labels. When you click one of the buttons it fires the background worker to do that particular task. The tasks, in this case, are to simply print a short message to the console and update the color of the label when the task starts running and when it completes.

16.10 BackgroundWorkerDemo.cs

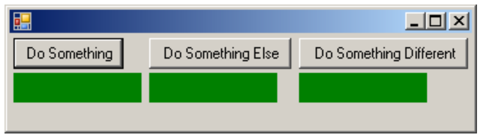
```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using System.ComponentModel;

public class BackgroundWorkerDemo : Form {

   private Button button1;
   private Button button2;
   private Button button3;
   private Label label1;
   private Label label2;
   private Label label3;
   private BackgroundWorker bw1;
   private BackgroundWorker bw2;
   private BackgroundWorker bw3;

   public BackgroundWorkerDemo(){
     InitializeComponents();
   }

   private void InitializeComponents(){
     button1 = new Button();
     button2 = new Button();
     button3 = new Button();
     label1 = new Label();
     label2 = new Label();
     label3 = new Label();
     bw1 = new BackgroundWorker();
     bw2 = new BackgroundWorker();
     bw3 = new BackgroundWorker();

     button1.Text = "Do Something";
     button1.AutoSize = true;
     button1.Click += ButtonOne_Click;
     label1.BackColor = Color.Green;
     bw1.DoWork += DoWorkOne;
     bw1.RunWorkerCompleted += ResetLabelOne;

     button2.Text = "Do Something Else";
     button2.AutoSize = true;
     button2.Click += ButtonTwo_Click;
     label2.BackColor = Color.Green;
     bw2.DoWork += DoWorkTwo;
     bw2.RunWorkerCompleted += ResetLabelTwo;

     button3.Text = "Do Something Different";
     button3.AutoSize = true;
     button3.Click += ButtonThree_Click;
     label3.BackColor = Color.Green;
     bw3.DoWork += DoWorkThree;
     bw3.RunWorkerCompleted += ResetLabelThree;

     TableLayoutPanel tlp1 = new TableLayoutPanel();
     tlp1.RowCount = 2;
     tlp1.ColumnCount = 3;
     tlp1.SuspendLayout();
     this.SuspendLayout();
     tlp1.AutoSize = true;
     tlp1.Dock = DockStyle.Left;
     tlp1.Controls.Add(button1);
     tlp1.Controls.Add(button2);
     tlp1.Controls.Add(button3);
     tlp1.Controls.Add(label1);
     tlp1.Controls.Add(label2);
     tlp1.Controls.Add(label3);
     this.Controls.Add(tlp1);
     this.AutoSize = true;
     this.AutoSizeMode = AutoSizeMode.GrowOnly;
     this.Height = tlp1.Height;
     tlp1.ResumeLayout();
     this.ResumeLayout();
   }

   private void ButtonOne_Click(Object sender, EventArgs e){
     if(!bw1.IsBusy){
        bw1.RunWorkerAsync(((Button)sender).Text);
     }
   }

   private void ButtonTwo_Click(Object sender, EventArgs e){
     if(!bw2.IsBusy){
        bw2.RunWorkerAsync(((Button)sender).Text);
     }
   }

   private void ButtonThree_Click(Object sender, EventArgs e){
     if(!bw3.IsBusy){
        bw3.RunWorkerAsync(((Button)sender).Text);
     }
   }

   private void DoWorkOne(Object sender, DoWorkEventArgs e){
     label1.BackColor = Color.Black;
     for(int i=0; i<30000; i++){
        Console.Write(e.Argument + “ “);
     }
   }

    private void DoWorkTwo(Object sender, DoWorkEventArgs e){
     label2.BackColor = Color.Black;
        for(int i=0; i<30000; i++){
          Console.Write(e.Argument + “ “);
     }
   }

    private void DoWorkThree(Object sender, DoWorkEventArgs e){
     label3.BackColor = Color.Black;
     for(int i=0; i<30000; i++){
        Console.Write(e.Argument + “ “);
     }
   }

   private void ResetLabelOne(Object sender, RunWorkerCompletedEventArgs e){
     label1.BackColor = Color.Green;
   }

   private void ResetLabelTwo(Object sender, RunWorkerCompletedEventArgs e){
     label2.BackColor = Color.Green;
   }

   private void ResetLabelThree(Object sender, RunWorkerCompletedEventArgs e){
     label3.BackColor = Color.Green;
   }


  [STAThread]
  public static void Main(){
    Application.Run(new BackgroundWorkerDemo());
  }// end Main

}  // end class definition
```

Referring to Example 16.10 — in this code I create three buttons, three labels, and three BackgroundWorker objects named bw1, bw2, and bw3 respectively. To each background worker's *DoWork* event I assign a method that conforms to the *DoWorkEventHandler* delegate. These methods are named DoWorkOne(), DoWorkTwo(), and DoWorkThree(). To each background worker's *RunWorkerCompleted* event I assign a method that conforms to the *RunWorkerCompletedEventHandler* delegate. I named these methods ResetLabelOne(), ResetLabelTwo(), and Reset-LabelThree().

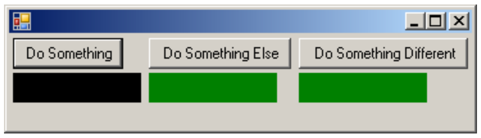
To each button's Click event I assign methods that conform to the EventHandler delegate. I've named these methods ButtonOne_Click(), ButtonTwo_Click(), and ButtonThree_Click(). A click on each button calls its assigned event handler method. The event handler method kicks off a background worker thread by calling its Run-WorkAsync() method. In this case, I'm passing in to the call to the RunWorkAsync() method the text of the clicked button.

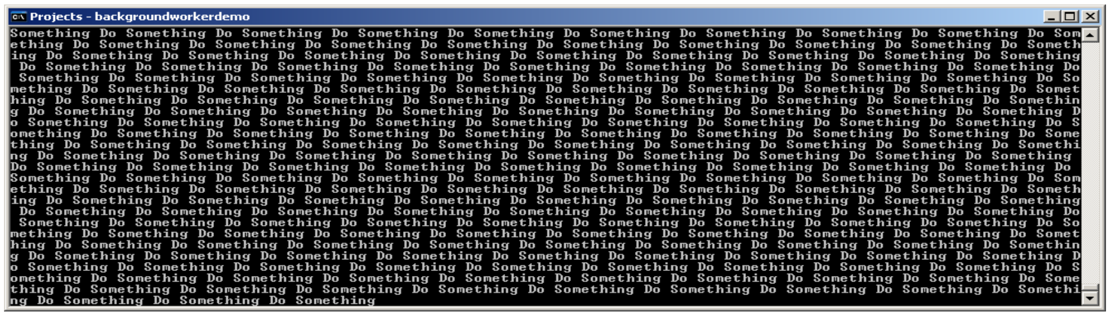
A call to a background worker's RunWorkAsync() method fires its DoWork event. Any DoWorkEventHandlers assigned to the background worker's DoWork event are then called. Before actually making the call to RunWorkerA-sync(), I check to see if the background worker is busy by polling its IsBusy property. If the background worker is currently running an asynchronous operation, the IsBusy property returns true.

When the background worker thread completes, its RunWorkerCompleted event fires resulting in a call to any assigned RunWorkerCompletedEventHandler methods. Figure 16-15 shows the results of running this program.



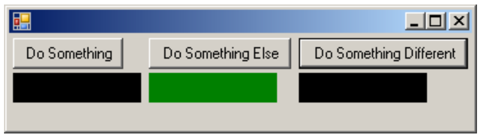
Click the Do Something button.

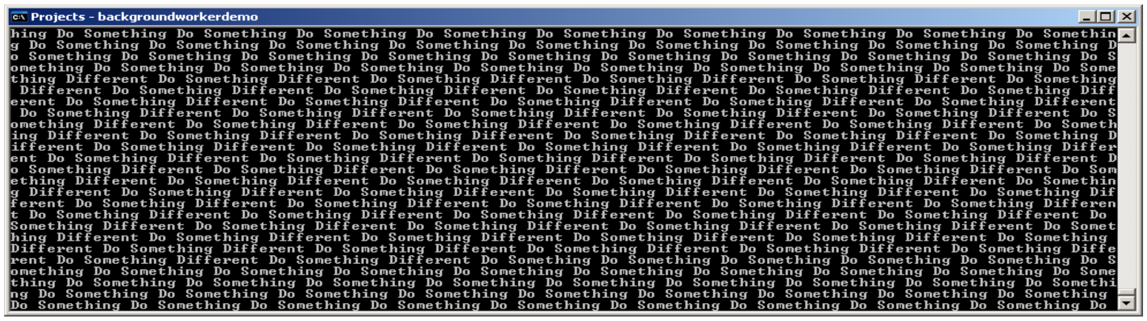




```
Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Som
ething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Someth
ing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something
 Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do
 Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do So
mething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somet
hing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somethin
g Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something D
o Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do S
omething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Some
thing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somethi
ng Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something
Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do
Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Som
ething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Someth
ing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something
 Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do
 Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do So
mething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somet
hing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somethin
g Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something D
o Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do S
omething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Some
thing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somethi
ng Do Something Do Something Do Something
```

Then click the Do Something Different button.





```
hing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somethin
g Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something D
o Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do S
omething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Some
thing Different Do Something Different Do Something Different Do Something Different Do Something Different Do Something
 Different Do Something Different Do Something Different Do Something Different Do Something Different Do Something Diff
erent Do Something Different Do Something Different Do Something Different Do Something Different Do Something Different
 Do Something Different Do Something Different Do Something Different Do Something Different Do Something Different Do S
omething Different Do Something Different Do Something Different Do Something Different Do Something Different Do Someth
ing Different Do Something Different Do Something Different Do Something Different Do Something Different Do Something D
ifferent Do Something Different Do Something Different Do Something Different Do Something Different Do Something Differ
ent Do Something Different Do Something Different Do Something Different Do Something Different Do Something Different D
o Something Different Do Something Different Do Something Different Do Something Different Do Something Different Do Som
ething Different Do Something Different Do Something Different Do Something Different Do Something Different Do Somethin
g Different Do Something Different Do Something Different Do Something Different Do Something Different Do Something Dif
ferent Do Something Different Do Something Different Do Something Different Do Something Different Do Something Differen
t Do Something Different Do Something Different Do Something Different Do Something Different Do Something Different Do
Something Different Do Something Different Do Something Different Do Something Different Do Something Different Do Somet
hing Different Do Something Different Do Something Different Do Something Different Do Something Different Do Something
Different Do Something Different Do Something Different Do Something Different Do Something Different Do Something Diffe
rent Do Something Different Do Something Different Do Something Do Something Do Something Do Something Do Something Do S
omething Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Some
thing Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Somethi
ng Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something
Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do
```

When both threads complete, each label's color is reset to green.

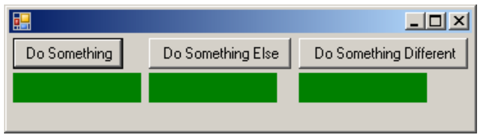


Figure 16-15: One Particular Result of Running Example 16.10

Quick Review

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.*BackgroundWorker* class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads. The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. A call to a BackgroundWorker's RunWorkAsync() method fires its DoWork event.

Thread Pools

The .NET runtime execution environment maintains and manages a *pool* of background threads for each application. You have access to these threads via the static methods of the ThreadPool class.

Beginning with .NET 2.0 Service Pack 1, each application's thread pool contains, by default, 250 worker threads per processor and 500 I/O completion port threads per processor. In this section, I will only show you how to use thread pool worker threads.

Important things to know about the application thread pool include the following:

- The ThreadPool class is static; its functionality is meant only to be used via its static methods.
- You can adjust the maximum number of threads in the pool via the ThreadPool.SetMaxThreads() method.
- To start a thread, pass the name of an execution method to the ThreadPool.QueueUserWorkItem() method.
- The thread pool contains a certain number of idle threads that are ready to execute. This number is adjusted via the ThreadPool.SetMinThreads() method. Too many idle threads extract a performance penalty because each idle thread requires stack space and other resources.
- The creation of new ThreadPool threads is throttled to one every 500 milliseconds. If you are spawning a large number of threads, you'll need to keep this throttling activity in mind.
- ThreadPool managed threads are background threads and will be terminated when your application exits.
- You have no control over a ThreadPool thread other than its initial creation.

Example 16.11 shows how easy it is to use ThreadPool threads. In this example, I spawn 45 separate threads with the help of the ThreadPool class. Following the creation of each thread, I print out the number of available threads.

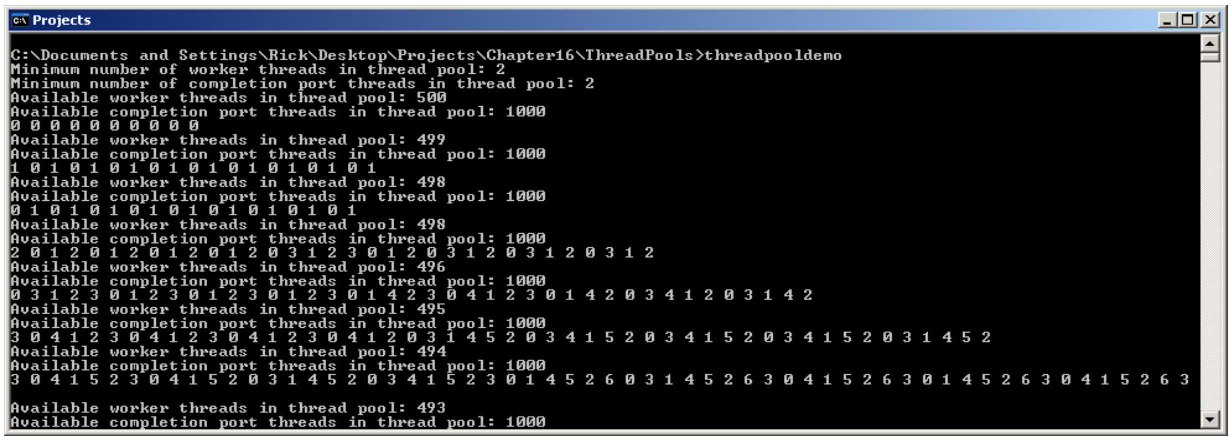
16.11 ThreadPoolDemo.cs

```
using System;
using System.Threading;

public class ThreadPoolDemo {

  private const int COUNT = 20000;

  public static void Run(object stateInfo){
    for(int i=0; i<COUNT; i++){
    Console.Write(stateInfo + " ");
    Thread.Sleep(100);
   }
  }

  public static void Main(){
     int workerThreads = 0;
    int completionPortThreads = 0;
    ThreadPool.GetMinThreads(out workerThreads, out completionPortThreads);
    Console.WriteLine("Minimum number of worker threads in thread pool: {0} ", workerThreads);
    Console.WriteLine("Minimum number of completion port threads in thread pool: {0} ",
                     completionPortThreads);
    ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
    Console.WriteLine("Available worker threads in thread pool: {0} ", workerThreads);
    Console.WriteLine("Available completion port threads in thread pool: {0} ", completionPortThreads);

    for(int i = 0; i<45; i++){
      ThreadPool.QueueUserWorkItem(new WaitCallback(Run), i);
28          Thread.Sleep(1000); // sleep twice as long as it takes to start a threadpool thread
29          ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
30          Console.Write("\nAvailable worker threads in thread pool: {0}  ", workerThreads);
31          Console.WriteLine("\nAvailable completion port threads in thread pool: {0}", completionPortThreads);
32        }
33      } // end Main() method
34    } // end class definition
```

Referring to Example 16.11 — each new thread is created in the body of the `for` loop that begins on line 26. Note on line 27 that the ThreadPool.QueueUserWorkItem() method requires a WaitCallBack object. I have supplied the name of the thread execution method to the WaitCallBack constructor and pass the resulting object as an argument to the QueueUserWorkItem() method. On line 28, I put the Main() method thread to sleep for twice as long as it takes to create a new ThreadPool thread, and then print the number of available threads to the console.

In this example, I have modified the signature of the Run() method to conform to the WaitCallBack delegate. This allows me to pass arguments to the Run() method when I kick off each thread with the QueueUserWorkItem() method.

Figure 16.16 shows a partial result of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\ThreadPools>threadpooldemo
Minimum number of worker threads in thread pool: 2
Minimum number of completion port threads in thread pool: 2
Available worker threads in thread pool: 500
Available completion port threads in thread pool: 1000
0 0 0 0 0 0 0 0 0 0
Available worker threads in thread pool: 499
Available completion port threads in thread pool: 1000
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Available worker threads in thread pool: 498
Available completion port threads in thread pool: 1000
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Available worker threads in thread pool: 498
Available completion port threads in thread pool: 1000
2 0 1 2 0 1 2 0 1 2 0 1 2 0 3 1 2 3 0 1 2 0 3 1 2 0 3 1 2 0 3 1 2
Available worker threads in thread pool: 496
Available completion port threads in thread pool: 1000
0 3 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 4 2 3 0 4 1 2 3 0 1 4 2 0 3 4 1 2 0 3 1 4 2
Available worker threads in thread pool: 495
Available completion port threads in thread pool: 1000
3 0 4 1 2 3 0 4 1 2 3 0 4 1 2 3 0 4 1 2 0 3 1 4 5 2 0 3 4 1 5 2 0 3 4 1 5 2 0 3 4 1 5 2 0 3 1 4 5 2
Available worker threads in thread pool: 494
Available completion port threads in thread pool: 1000
3 0 4 1 5 2 3 0 4 1 5 2 0 3 1 4 5 2 0 3 4 1 5 2 3 0 1 4 5 2 6 0 3 1 4 5 2 6 3 0 4 1 5 2 6 3 0 1 4 5 2 6 3 0 4 1 5 2 6 3

Available worker threads in thread pool: 493
Available completion port threads in thread pool: 1000
```

Figure 16-16: Partial Result of Running Example 16.11

Quick Review

The .NET runtime execution environment maintains and manages a pool of background threads for each application. You have access to these threads via the static methods of the ThreadPool class. By default, each application's thread pool contains 250 worker threads per processor and 500 I/O completion port threads per processor. Pass the name of your thread execution method to the WaitCallBack constructor; pass the WaitCallBack object to the ThreadPool.QueueUserWorkItem() method.

Asynchronous Method Calls

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method asynchronously with the help of a delegate. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

To make an asynchronous method call follow these steps:

- Create a new delegate type that specifies the method signature of your thread execution method.
- Create your thread execution method making sure its method signature matches that of the delegate you created in the first step.
- Create an instance of the delegate, passing the name of the thread execution method to its constructor.

- Call the BeginInvoke() method on the delegate object, supplying any necessary thread execution method arguments and two additional arguments of type AsyncCallback and an Object respectively. I will discuss the purpose of the AsyncCallback and Object parameters shortly.
- The call to BeginInvoke() returns an IAsyncResult object that can be used to query the state of the asynchronous method call's execution progress. The IAsyncResult.AsyncState property is a reference to the last object supplied in the call to the BeginInvoke() method.
- Do any required work in the calling method while the asynchronous method call executes.
- Call the EndInvoke() method to properly wrap-up the asynchronous method call and fetch the results.

Example 16.12 shows the asynchronous call mechanism in action.

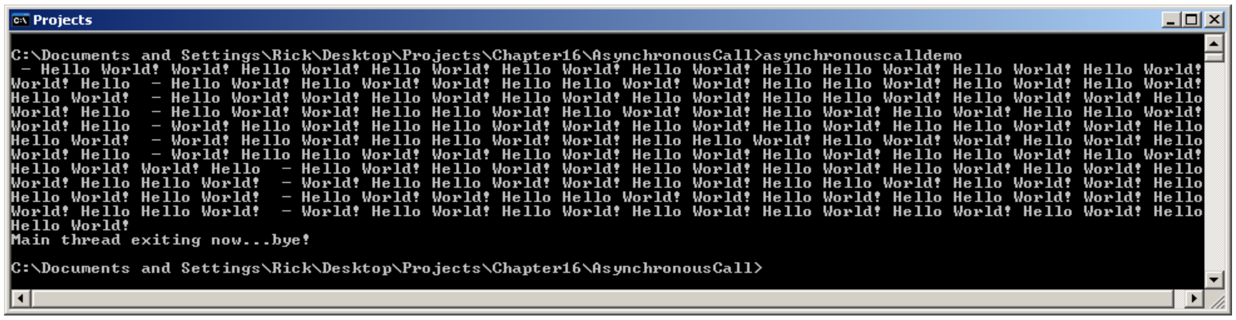
16.12 AsynchronousCallDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class AsynchronousCallDemo {
5
6      private const int COUNT = 100;
7      public delegate void RunDelegate(String message);
8
9      public static void Run(String message){
10       for(int i=0; i<COUNT; i++){
11        Console.Write(message + " ");
12        Thread.Sleep(100);
13      }
14     }
15
16     public static void Main(){
17      RunDelegate runDelegate1 = new RunDelegate(Run);
18      RunDelegate runDelegate2 = new RunDelegate(Run);
19      IAsyncResult result1 = runDelegate1.BeginInvoke("Hello", null, null);
20      IAsyncResult result2 = runDelegate2.BeginInvoke("World!", null, null);
21      while(!result1.IsCompleted && !result2.IsCompleted){
22        Console.Write(" - ");
23        Thread.Sleep(1000);
24      }
25      runDelegate1.EndInvoke(result1);
26      runDelegate2.EndInvoke(result2);
27      Console.WriteLine("\nMain thread exiting now...bye!");
28     } // end Main() method
29   } // end class definition
```

Referring to Example 16.12 — on line 7, a new delegate type is declared named RunDelegate. The RunDelegate specifies a method that takes one String argument. The Run() method on line 9 conforms to the RunDelegate method signature specification. In the Main() method, I created two RunDelegate instances named runDelegate1 and runDelegate2. In the call to the RunDelegate constructor, I pass the name of the Run() method. I start the asynchronous methods by calling the BeginInvoke() method on each delegate instance passing in the required string argument and two null values representing the AsyncCallback and AsyncState objects, which are not being used in this case.

The `while` statement on line 21 loops until both IAsyncResult.IsCompleted properties are true. It prints the '-' character to the console and then sleeps for 1000 milliseconds to let the other two threads have a go at the processor.

On lines 25 and 26, the EndInvoke() method is called on each delegate instance, passing in the appropriate IAsyncResult reference. Figure 16-17 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>asynchronouscalldemo
 - Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! Hello World!
World! Hello  - Hello World! Hello World! World! Hello Hello World! World! Hello Hello World! Hello World! Hello World!
Hello World!  - Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World! World! Hello
World! Hello  - Hello World! World! Hello Hello World! Hello World! World! Hello World! Hello World! Hello Hello World!
World! Hello  - World! Hello World! Hello Hello World! World! Hello World! Hello World! Hello Hello World! World! Hello
Hello World!  - World! Hello World! Hello Hello World! World! Hello Hello World! Hello World! World! Hello World! Hello
World! Hello  - World! Hello Hello World! World! Hello World! Hello World! Hello World! Hello Hello World! Hello World!
Hello World! World! Hello  - Hello World! Hello World! World! Hello World! Hello World! Hello World! Hello World! Hello
World! Hello Hello World!  - World! Hello Hello World! World! Hello World! Hello Hello World! Hello World! World! Hello
Hello World! Hello World!  - Hello World! World! Hello Hello World! World! Hello World! Hello Hello World! World! Hello
World! Hello Hello World!  - World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello World! Hello
Hello World!
Main thread exiting now...bye!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>
```

Figure 16-17: Results of Running Example 16.12

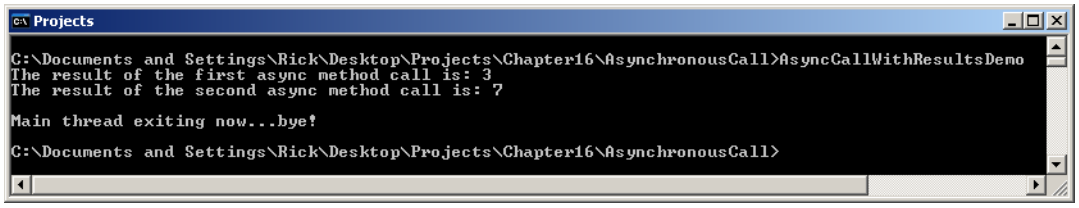
Obtaining Results From An Asynchronous Method Call

The are several ways to obtain results from an asynchronous method call. If the method returns a value, the call to the delegate's EndInvoke() method returns that value. If the method takes one or more `out` or `ref` parameters, these will be included in the EndInvoke() method's parameter list as well. (**Note:** Remember, a delegate's BeginInvoke() and EndInvoke() methods are automatically generated.) Example 16.13 demonstrates the use of the EndInvoke() method to retrieve an asynchronous method call's return value.

16.13 AsyncCallWithResultsDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class AsyncCallWithResultsDemo {
5
6      private const int COUNT = 100;
7      public delegate int SumDelegate(int a, int b);
8
9      public static int Sum(int a, int b){
10       return a + b;
11     }
12
13     public static void Main(){
14      SumDelegate sumDelegate1 = new SumDelegate(Sum);
15      SumDelegate sumDelegate2 = new SumDelegate(Sum);
16      IAsyncResult result1 = sumDelegate1.BeginInvoke(1, 2, null, null);
17      IAsyncResult result2 = sumDelegate2.BeginInvoke(3, 4, null, null);
18      while(!result1.IsCompleted && !result2.IsCompleted){
19        Thread.Sleep(100);
20      }
21      int sum1 = sumDelegate1.EndInvoke(result1);
22      int sum2 = sumDelegate2.EndInvoke(result2);
23      Console.WriteLine("The result of the first async method call is: {0}", sum1);
24      Console.WriteLine("The result of the second async method call is: {0}", sum2);
25      Console.WriteLine("\nMain thread exiting now...bye!");
26     } // end Main() method
27   } // end class definition
```

Referring to Example 16.13 — I defined a delegate on line 7 named SumDelegate that takes two integer arguments and returns an integer value. The Sum() method on line 9 conforms to the SumDelegate signature. In the Main() method, two SumDelegate objects are created named sumDelegate1 and sumDelegate2. The BeginInvoke() method is called on each delegate. Note how the multiple arguments are passed to the asynchronous method call. On line 18, the `while` loop spins until both method calls complete, which in this case doesn't take too long because of the simplicity of the Sum() method. On lines 21 and 22, the results of each method call are obtained via the call to each delegate's EndInvoke() method and the values written to the console. Figure 16-18 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>AsyncCallWithResultsDemo
The result of the first async method call is: 3
The result of the second async method call is: 7

Main thread exiting now...bye!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>
```

Figure 16-18: Results of Running Example 16.13

Providing A CallBack Method To BeginInvoke()

The BeginInvoke() method allows you to pass in a callback method that is automatically called when the asynchronous method completes execution. It also allows you to pass in an object argument to that callback method. Note that up until now I have been calling the BeginInvoke() method with the last two arguments set to null. (*i.e.,* sumDelegate1.BeginInvoke(1, 2, null, null)) To pass in a callback method, you'll need to write a method that conforms to the AsyncCallBack delegate method signature, which returns `void` and takes one argument of type IAsyncResult as the following code snippet shows:

```
void MethodName(IAsyncResult result)
```

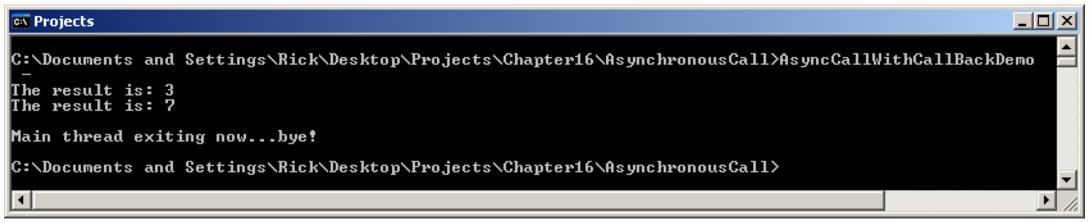
The IAsyncResult interface specifies an AsyncState property of type Object, meaning it can contain any type of object. You can pass in whatever your heart desires! To use this object in the callback method, you'll need to access the IAsyncResult.AsyncState property and cast it to the expected type. Example 16.14 demonstrates the use of a callback method.

16.14 AsyncCallWithCallBackDemo.cs

```
1    using System;
2    using System.Threading;
3
4    public class AsyncCallWithCallBackDemo {
5
6      private const int COUNT = 100;
7      public delegate int SumDelegate(int a, int b);
8
9      public static int Sum(int a, int b){
10       return a + b;
11     }
12
13     public static void WrapUp(IAsyncResult result){
14       SumDelegate sumDelegate = (SumDelegate)result.AsyncState;
15      int sum = sumDelegate.EndInvoke(result);
16      Console.WriteLine("The result is: {0} ", sum);
17     }
18
19     public static void Main(){
20      SumDelegate sumDelegate1 = new SumDelegate(Sum);
21      SumDelegate sumDelegate2 = new SumDelegate(Sum);
22      IAsyncResult result1 = sumDelegate1.BeginInvoke(1, 2, new AsyncCallback(WrapUp), sumDelegate1);
23      IAsyncResult result2 = sumDelegate2.BeginInvoke(3, 4, new AsyncCallback(WrapUp), sumDelegate2);
24      while(!result1.IsCompleted && (!result2.IsCompleted)){
25        Console.WriteLine(" - ");
26        Thread.Sleep(10);
27      }
28      Console.WriteLine("\nMain thread exiting now...bye!");
29     } // end Main() method
30   } // end class definition
```

Referring to Example 16.14 — I have added a method on line 13 named WrapUp() that conforms to the AsyncCallBack delegate method signature. In this example, I'm using the WrapUp() method to make the call to a SumDelegate's EndInvoke() method. To do this, I must pass in a reference to a SumDelegate, which I do as the last argument to each SumDelegate's BeginInvoke() method call shown on lines 22 and 23.

So, what's going on here? I'm executing two asynchronous method calls via two SumDelegate references. When each asynchronously executed method returns, the method supplied as the callback method is automatically called. It's a nice way to call and forget. However, since this is a simple console application, and the threads being created to execute the asynchronous method calls are thread pool background threads, the Main() method must hang on for a while and do some stuff, for if it exits right away, the background threads will be destroyed before they get a chance to execute. You generally don't have this problem when you're writing a GUI application. Figure 16-19 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>AsyncCallWithCallBackDemo
 -
The result is: 3
The result is: 7

Main thread exiting now...bye!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter16\AsynchronousCall>
```

Figure 16-19: Results of Running Example 16.14

Quick Review

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method *asynchronously* with the help of a delegate. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

Summary

A *process* consists of one or more threads of execution, referred to simply as *threads*. A process always consists of at least one thread, the Main thread, which starts running when the process begins execution. A single-threaded process contains only one thread of execution. A multithreaded process contains more than one thread.

A *thread* is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and application domain.

Waiting threads sit in a *thread queue* until they are loaded into the processor. Each thread has a data structure known as a *thread context*. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system the operating system allocates processor time with a *time-slicing* scheme. Each thread gets a little bit of time to execute before being *preempted* by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor.

In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

A thread can assume several different states during its execution lifetime. These states include: *Unstarted*, *Running*, *Background*, *SuspendRequested*, *Suspended*, *WaitSleepJoin*, *StopRequested*, *Stopped*, *AbortRequested*, and *Aborted*.

A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() simply notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in, or more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are obsolete.

To create a managed thread, pass to the Thread constructor either a ThreadStart delegate or a ParameterizedThreadStart delegate.

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass the ThreadStart delegate to the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

If you need to pass in an argument when you start a thread, the thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Pass the argument to the thread via its Start() method. Remember to cast the argument to the appropriate type in the body of the thread's execution method.

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system preempts the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Call the Thread.Sleep() method to force a thread to *block* and give other threads a chance to execute.

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution then in the Main thread you would call thread2.Join().

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread keeps the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.BackgroundWorker class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads. The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. A call to a BackgroundWorker's RunWorkAsync() method fires its DoWork event.

The .NET runtime execution environment maintains and manages a *pool* of background threads for each application. You have access to these threads via the static methods of the ThreadPool class. By default, each application's thread pool contains 250 worker threads per processor and 500 I/O completion port threads per processor. Pass the name of the thread execution method to the WaitCallBack constructor; pass the WaitCallBack object to the ThreadPool.QueueUserWorkItem() method.

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method *asynchronously* with the help of a *delegate*. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

Skill-Building Exercises

1. **API Drill:** Explore the System.Threading namespace. List each class and describe its purpose.

2. **API Drill:** Explore the .NET Framework documentation and search for information about thread synchronization mechanisms. Pay particular attention to the *Interlocked* class, the *lock* keyword, the *Monitor* class, the *Mutex* class, and the *Semaphore* class. Describe in your own words how each synchronization mechanism works.

4. **Web Research:** Search the web for information about *deadlock* and *race* conditions. Briefly explain how each of the thread synchronization mechanisms you learned about in the previous exercise can be used to avoid either deadlock or race conditions.

5. **Web Research:** Procure and read the paper titled *Race Conditions: A Case Study* by Steve Carr, et. al.

6. **Programming Exercise:** Compile and run the sample programs presented in this chapter. Experiment by making various modifications to the programs and note the results of their execution.

7. **API Drill:** Explore the System.Collections.Generic namespace. Study each class and note how to use it in a multithreaded program.

Suggested Projects

1. **Multithreaded Water Tank:** Revisit the Automated Water Tank program given in Chapter 13, Examples 13.6 through 13.11, and make it a multithreaded program.

Self-Test Questions

1. (True/False) An application always has at least one thread.

2. What happens to background threads when an application exits?

3. What's the difference between a background thread and a foreground thread?

4. (True/False) A managed thread immediately starts running when you call its Start() method.

5. Which two Thread methods are considered obsolete?

6. (True/False) It's generally considered a good idea to try to manage multiple threads by manipulating their states.

7. Beginning with .NET 2.0 Service Pack 1, how many ThreadPool worker threads are available per processor?

8. What's the difference between a ThreadStart delegate and a ParameterizedThreadStart delegate?

9. Which two delegate methods are used together to run methods asynchronously?

10. What's the difference between a process and a thread?

11. How does the typical operating system coordinate thread execution?

12. What term is used to describe what happens when one thread is removed from the processor in favor of another?

References

Atul Gupta, *How Many Threads Have I Got?*, [http://infosysblogs.com/microsoft/2007/04/how_many_threads_have_i_got.html]

Steve Carr, et. al, *Race Conditions: A Case Study*

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Notes