Chapter 13
Working with Threads

Until relatively recently only advanced programmers understood and knew how to employ threads in application programs. Part of the problem was that using threads was not straightforward and presented a significant challenge. Much of the stigma of thread programming has been eliminated with .NET. Threads are not only easy to use, and are actually fun to work with. The big payoff is in the ability to write programs that are better organized, easier to debug, and have better performance. Before we get going I want to review some basic concepts for those readers who have not taken a good course on operating systems.

Processes

A process is a program in execution. A program may exist as a file on your hard disk in either source or binary form, but it is not a process. Once you invoke the program by clicking on an icon, typing the name as a command, or even if the program is started by another program, the operating system has to create a process to run the program. A process is a kind of container for your program provided by the operating system.

The operating system needs to provide at least the following items for a process:

1. A memory image where the program code and data can be loaded.
2. A stack for handling method calls and variables local to those methods.
3. A heap for the allocation of dynamically created objects.
4. Data structures to manage the process such as saving the CPU state when other programs run.
5. Any other resources that the operating system may allocate to a process.

The operating system manages many processes at the same time. Some are processes associated with the system and other utilities you may be running. The Windows Task Manager provides an easy way to examine what processes are running on your system. Figure 13-1 shows a typical display of the processes running on a system. Notice the last column on the right and you will see the count of the number of threads that are running for that process. You may have to use the view menu to get this column to display. You can get an idea of the importance of threads when you see the number of programs with significantly more than one thread.
A Thread is Born

Normally there is one sequence of instructions that are executed when a process is executing. The actual instructions depend on the flow of your program. A thread allows executing more than one part of a program at the same time. Of course true concurrency is only possible when there is more than one CPU present in the system. However, since all versions of Windows starting with Windows 95 support preemptive multi-tasking, the CPU is switched from process to process or thread to thread so rapidly that we have the illusion that the processes or threads are being executed at the same time.

A thread is sometimes called a lightweight process. It bears most of the characteristics of a process except that it belongs to a process and can't stand on its own. A process always has at least one thread. All the threads of a specific process share everything except for the stack. Each thread has its own stack. Of course the operating system must keep track separately of the instruction being executed for each thread and any other CPU state information. What is most important is that all the threads of a process share the same program and data.
Why We Use Threads

There are a number of compelling reasons to use threads. As I mentioned earlier using threads can improve the organization of your program. Key to this is that we can eliminate some very ugly asynchronous programming techniques. I will tell you more on that later. There is a simple and very important reason that you might want to use threads or in some cases must use threads.

Key to a Windows application is that for a good application it must respond to user mouse clicks in a timely manner. Remember that central to a Windows program is the ability to respond to events. Even if you don’t provide a handler for an event that doesn't mean that the .NET environment doesn't. When you click on the close button of an application, Windows generates an event that ultimately terminates your application. The user expects that this action will take place in a timely manner. How often have you seen the familiar message that your application is not longer responding? How does this happen? It's simple. If you don't respond to user input in a timely manner you get this response. Maybe your program is doing something useful or maybe it's hung in an endless loop. The system has no idea.

Figure 13-2 shows the code for a simple program that enters a counting loop when the mouse is clicked in the client area of the form. Depending on the speed of your CPU it takes somewhere around 30 seconds for the loop to run. During this time the cursor changes to the hourglass. While the program is stuck in the mouse event handler it can't respond to other Windows messages. If you try to close the application by clicking several times on the close button you will get the dialog displayed in Figure 13-3. If you click on cancel you are taken back to the program to continue to wait. Eventually the cursor will switch back to the arrow and you will be able to close the form. Alternatively you can terminate the program.

No Response - Form1.cs

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace NoResponse
{   public partial class Form1 : Form
{       public Form1()
       {
           InitializeComponent();
       }

       private void Form1_MouseClick(object sender, MouseEventArgs e)
       {

```csharp
```
Cursor = Cursors.WaitCursor;
//Loop for approximately 30 seconds (depends on system speed)
long i = 3000000000;
while (--i>0) {
    Cursor = Cursors.Default;
}
}
}

Figure 13-2

The Thread Class

The System.Threading.Thread class is used to create a new thread. Remember, there is always at least one thread for a process. In fact, the typical Windows Forms program has more than one thread. It is not important what these helper threads do since you are only interested in your main thread, the user interface or UI thread and any worker threads you create. For the curious, one of the helper threads is the garbage collector used by the common language runtime, CLR.

You can get a reference to the thread you are running on using the static Thread.CurrentThread method. You can then perform such operations as getting or setting the name of the thread, determining its priority, or even examining its current state. Of course you need to be careful if you decide to change critical characteristics of your main UI thread or you might wind up crashing your application.

Creating a new thread is very simple. There are several constructors for a Thread, but the basic constructor takes a ThreadStart object as its only parameter. The ThreadStart object is used to tell the thread where it is to start executing when it is started. Figure 13-4 shows a first try at creating a worker thread to perform the looping operation.
Good Response - First Try

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace GoodResponse
{
    public partial class Form1 : Form
    {
        private Thread workerThread;
        private ThreadStart workerStart;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            // Don't create more than one worker thread
            if (workerThread == null)
            {
                workerStart = new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);
                workerThread.Start();
            }
        }

        private void WorkerThread()
        {
            // Loop for approximately 30 seconds (depends on system speed)
            long i = 30000000000;
            while (--i > 0)
            {
                Cursor = Cursors.WaitCursor;
            }
            Cursor = Cursors.Default;
            workerThread = null;
        }
    }
}

Figure 13-4

As you can see the code that was previously in the mouse click handler has been placed in a new method, WorkerThread. These statements create and start the worker thread:
workerStart = new ThreadStart(WorkerThread);
workerThread = new Thread(workerStart);
workerThread.Start();

Unfortunately when we try to run the program using debugging the exception in Figure 13-5 appears. If you run the program without debugging it runs fine. Well not really, but I will get to that later. In versions of Visual Studio prior to Visual Studio 2005 this warning is never generated even though the sin has been committed. Many programs are out there that are not technically correct and they may seemingly run fine until the random crash or lockup takes place. The problem is that we are trying to change the form’s cursor from a thread that is different than the thread that manages the user interface for the form. Before we fix things up let's address a couple of hidden problems with our initial approach.

A worker thread terminates when it returns from the method used to start it. A thread can be forcefully terminated with the `Abort` method, but using that is not normally a good idea. We prefer that threads gracefully terminate themselves.

**Figure 13-5**

A worker thread terminates when it returns from the method used to start it. A thread can be forcefully terminated with the `Abort` method, but using that is not normally a good idea. We prefer that threads gracefully terminate themselves.

**Foreground vs. Background Threads**

By default a new thread is a *foreground* thread. That means that it is of equal statures to your application’s main thread. A program will not terminate until all of its foreground threads have terminated. If you have a foreground worker thread that gets in an infinite loop it will just keep sucking up CPU resources forever or until you manually terminate it, log out, or reboot the system. You won't even know it's there since the application will not appear in the task bar. The solution is very simple, we make the thread a *background* thread with this simple call:
workerThread.IsBackground = true;

Now when we close the application all threads are terminated. This is much safer and you won't have to worry about zombie threads looping away that are hard to kill off.

Let's make a simple change to the application that gets us around the exception mentioned above. We can always call up a MessageBox object without fear. We can use that instead of changing the cursor. Figure 13-6 shows our corrected version.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace GoodResponse
{
    public partial class Form1 : Form
    {
        private Thread workerThread;
        private ThreadStart workerStart;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            // Don't create more than one worker thread
            if (workerThread == null)
            {
                workerStart = new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);
                workerThread.IsBackground = true;
                workerThread.Start();
            }
        }

        private void WorkerThread()
        {
            MessageBox.Show("Worker thread started.");
            // Loop for approximately 30 seconds (depends on system speed)
            long i = 3000000000;
            while (--i > 0)
            {
                MessageBox.Show("Worker thread terminated.");
                workerThread = null;
            }
        }
    }
}
```
If you run this program you can now close the application instantly if our worker thread is looping. But this is still not realistic if we consider that usually a user wants to cancel an operation rather than just terminate the entire operation. What we really need is a cancel button. Read on.

**Communicating with the Thread**

The most common method of communicating with a thread is by the use of *shared memory*. If this is done with care we can accomplish many things. The first example I will show is one where we do not have a *race condition*. I will discuss race conditions shortly.

When we perform time consuming operations there is usually some type of looping involved. This might be a result of copying a file or performing some long calculation involving many elements. If we properly design the task the worker thread performs we can insert a check inside the loop that can check to see if the user wants to cancel the operation. If so, then appropriate cleanup steps can be taken and the thread allowed to terminate normally. This is far better than aborting the thread using the *Abort* method.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace CancelThread
{
    public partial class Form1 : Form
    {
        private Thread workerThread;
        private ThreadStart workerStart;
        private bool cancel = false;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Click(object sender, EventArgs e)
        {
            //Don't create more than one worker thread
            if (workerThread == null)
            {
                workerStart = new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);

                //Let's do some work
                workerThread.Start();
            }
            else
            {
                workerThread.Abort();
            }
        }
    }
}
```
workerThread = new Thread(workerStart);
workerThread.IsBackground = true;
cancel = false;
workerThread.Start();
}
}

private void WorkerThread()
{
    MessageBox.Show("Worker thread started.");
    //Loop for approximately 30 seconds (depends on system
    // speed)
    long i = 3000000000;
    while (--i > 0) if(cancel) break;
    MessageBox.Show("Worker thread terminated.");
    workerThread = null;
}

private void cancelButton_Click(object sender, EventArgs e)
{
    cancel = true;
}
}
}

Figure 13-7 and Figure 13-8 demonstrate the use of a shared variable, cancel, that is a bool. It is initially set to false. When the cancel button is clicked this variable is set to true. Inside the worker thread's loop is a test to see when this variable is set to true by the user interface thread. If so we break out of the loop and gracefully terminate.

**Cancel Thread - Form1.cs**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace CancelThread
{
    public partial class Form1 : Form
    {
        private Thread workerThread;
        private ThreadStart workerStart;
        private bool cancel = false;

        public Form1()
        {
            InitializeComponent();
        }
    }
}
private void Form1_Click(object sender, EventArgs e)
{
    // Don't create more than one worker thread
    if (workerThread == null)
    {
        workerStart = new ThreadStart(WorkerThread);
        workerThread = new Thread(workerStart);
        workerThread.IsBackground = true;
        cancel = false;
        workerThread.Start();
    }
}

private void WorkerThread()
{
    MessageBox.Show("Worker thread started.");
    // Loop for approximately 30 seconds (depends on system
    // speed)
    long i = 3000000000;
    while (--i > 0) if(cancel) break;
    MessageBox.Show("Worker thread terminated.");
    workerThread = null;
}

private void cancelButton_Click(object sender, EventArgs e)
{
    cancel = true;
}

Figure 13-7

Figure 13-8
This very simple example demonstrates how simple it is to communicate with a thread if the thread contains a loop of some kind. This is almost always the case with worker threads that are performing time consuming operations. However, it is not always the case when threads are used for such purposes as listening on network ports. I won’t cover all the more sophisticated topics of programming with threads as you may start to get very confused until you gain some experience with thread basics. That’s the objective here and we have plenty more to talk about.

So we can communicate with a thread using a shared variable as long as the thread is checking periodically, but how do we do that with the user interface thread since we never loop there. Rather, we wait for the next Windows message to arrive. You might be thinking that it would be nice if we could send a message to the thread running our form letting it know that the thread has completed. Such a mechanism is provided for all classes derived from System.Windows.Forms.Control. The Form class is such a class.

**Control.Invoke**

The *Invoke* method can be used to execute a *delegate* on the thread that the control was created on. In the case of our form the thread is the user interface thread. *Invoke* take a single argument which is a *delegate*. I discussed delegates in Chapter <ref>. As a refresher, a *delegate* is similar to a pointer to a function in C++. All we need to do is set up a *delegate* to handle the actions required when the worker thread needs action by the main thread.

### Invoke Example - Form1.cs

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace InvokeExample
{
    public partial class Form1 : Form
    {
        private Thread workerThread;
        private ThreadStart workerStart;
        private bool cancel = false;
        private delegate void Done();
        private Done done;

        public Form1()
        {
            InitializeComponent();
            done = new Done(ThreadDone);
```
private void Form1_Click(object sender, EventArgs e)
{
    // Don't create more than one worker thread
    if (workerThread == null)
    {
        workerStart = new ThreadStart(WorkerThread);
        workerThread = new Thread(workerStart);
        workerThread.IsBackground = true;
        cancel = false;
        Cursor = Cursors.WaitCursor;
        workerThread.Start();
    }
}
private void WorkerThread()
{
    // Loop for approximately 30 seconds (depends on system speed)
    long i = 3000000000;
    while (--i > 0) if (cancel) break;
    Invoke(done);
    // done();
    workerThread = null;
}
private void cancelButton_Click(object sender, EventArgs e)
{
    cancel = true;
}
private void ThreadDone()
{
    Cursor = Cursors.Arrow;
}
}

Figure 13-9

Thread Safety

A very important aspect of programming with threads is what is known as thread safety. What exactly does that mean? It relates to the ramifications of executing a method in more than one thread at a time if that method uses the same data objects. How do we prevent things from going wrong if changes are being made simultaneously by two or more threads? This is known as the critical section problem in operating system texts. A critical section is any segment of code that accesses shared variables. The local variables of a method are never shared because each thread has its own stack. However, fields and properties of
classes exist on behalf of an object instance and can be easily shared between objects and the methods implemented by those objects.

A method is thread safe if it can be executed in more than one thread at a time. Here is an example:

```java
public int AddOne(int i) {
    return i+1;
}
```

This is thread safe because no shared data is being modified. The parameter `i` is actually local to the method since it is a value type and the method actually gets its own private copy. If this method is executed simultaneously by more than one thread they will each get the same copy of `i` and return the same value. Return values are distinct for each method.

**The Race Condition**

Now let's write a method that is not thread safe.

```java
private int i;
public void AddOne() {
    i = i+1;
}
```

At first it may look like this is benign. Unfortunately it isn't. The steps required to perform the increment operation in native code probably go something like this:

1. Copy the field `i` into a temporary CPU register.
2. Add one to the register.
3. Save the register's value back to the field `i`.

Now suppose that two threads call `AddOne` at the same time and that they both execute step one. They both get the same value for `i`. Now suppose that they both complete step two prior to completing step three. The value in each register will be the same. Finally both threads complete step three. So what's wrong? Both threads write the same value in `i`. The actual order doesn't matter. The resulting value would be `i` plus one and it should be `i` plus two. We lost one of the increments. If the steps were not performed at the same time then the value would be correct.

This method is not thread safe. Only a few methods in the framework class library are thread safe. Most are not. That means we need to be very careful how we use them. If they are executed on behalf of different object instances we are usually safe. However, if more than one of the objects has a reference to a common object then we might be in big trouble. The solution to this very challenging issue is not simple and it would be difficult for me to tell you everything you need to know in this introductory chapter. Appreciating the problem and learning some simple solutions will get you started.
Mutual Exclusion and the Lock Keyword

What we are really seeking is something known as mutual exclusion. In other words we want only one thread at a time to manipulate $i$. We want only one thread to have access to $i$ at a time. We want to perform an atomic operation. An atomic operation is one that is executed with mutual exclusion.

C# provides a special keyword, lock, that can be used to guarantee mutual exclusion. While this may initially appear to be a compiler feature it is actually a shortcut for writing code that uses the FCL's Monitor class. Using lock is very simple. We create a reference object to use to protect the shared data we want to manipulate with mutual exclusion. We must use a reference object for this purpose and not a value object. For example, we can't use $i$ itself. The reason is that value types are boxed into reference types and each time the value is boxed we get a different object. This means that we are not obtaining a lock on the same object, but rather independent copies. Here is how we can use lock to solve the problem for the AddOne method:

```csharp
private Object ilock;
private int i;
public void AddOne()
{
    lock(ilock)
    {
        i = i+1;
    }
}
```

We could use the entire class as the object to lock rather than just locking specific fields. We could merely use

```csharp
lock(this)
{
    ....
}
```

This is not usually a good idea since we prevent any method using lock(this) from being called on behalf of the object rather than just those that wish to atomically manipulate $i$. Note, there is no protection unless the method uses the lock keyword. Using lock in one method doesn't protect you from concurrently executing other methods that don't lock the same object. All this is up to you, the programmer, to do the right thing. It is not foolproof.

What happens when lock is executed and another thread is inside its lock? The thread is suspended until the lock is released and then it is allowed to obtain its lock and proceed. The lock keyword eliminates some of the errors that are prevalent when a pair of methods are used and the programmer forgets to release the lock. Lock also handles the issue when the block of code is left due
to an exception or a method return. The actual code generated using \textit{lock} goes something like this:

\begin{verbatim}
System.Threading.Monitor.Enter(obj);
try
{
    //your code goes here
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
\end{verbatim}

where \textit{obj} is the argument to \textit{lock}. The \textit{finally} block is always executed regardless of how the \textit{try} block is left. A very common bug is that the \textit{Enter} and \textit{Exit} methods, or the equivalent in other programming environments, are not always used in a pair. This can result in an apparent lockup or deadlock within an application. Worse is that a race condition is allowed to exist and then completely unpredictable things can happen. Using \textit{lock} makes your code much more reliable.

\textbf{Timer}

The Windows operating system provides a timer mechanism so that threads can suspend themselves for some period of time and then be allowed to continue. The WIN32 \textit{Sleep} function is available as a static method in the \textit{Thread} class. \textit{Thread.Sleep} takes an integer argument representing the number of milliseconds to suspend the process. You can't use this value for exact timing purposes. It is approximate only due to granularity in the underlying mechanism used by the operating system as well as the scheduling delay in waking up the thread once the time expires.

The SleepEx example shown in Figure 13-10 demonstrates how to use the \textit{Thread.Sleep} method and to communicate with your form in a thread safe manner. The worker thread takes approximately 30 seconds to run. It repeatedly sleeps for one second and then advances a progress bar in the form by using a call to Invoke for the delegate \textit{step}. The maximum value for the progress bar was changed to 30 and the step size was changed to one.

\begin{verbatim}
SleepEx - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
\end{verbatim}
namespace SleepEx
{
    public partial class Form1 : Form
    {
        private Thread workerThread;
        private ThreadStart workerStart;
        private delegate void Step();
        private Step step;

        public Form1()
        {
            InitializeComponent();
            step = new Step(StepBar);
        }

        private void startButton_Click(object sender, EventArgs e)
        {
            // Don't create more than one worker thread
            if (workerThread == null)
            {
                pBar.Value = 0; // reset progress bar
                workerStart = new ThreadStart(WorkerThread);
                workerThread = new Thread(workerStart);
                workerThread.IsBackground = true;
                workerThread.Start();
            }
        }

        private void WorkerThread()
        {
            for (int i = 0; i < 30; ++i)
            {
                Thread.Sleep(1000);
                Invoke(step);
            }
            workerThread = null;
        }

        private void StepBar()
        {
            pBar.PerformStep();
        }
    }
}

Figure 13-10

Figure 13-11 show the program running. If you run this example on your own machine you might observe that the progress bar doesn't always advance at a precise rate. Sometimes you might see that the time between steps can vary. Two steps might seem close together and then a bit further apart in time. This variance is due to the fact that Windows has other threads to run and either our worker thread or the thread running our form may need to wait its turn to run.
Thread Synchronization

We have seen how a thread can communicate with the thread running your form. Since this thread spends most of its time waiting for events such as mouse clicks we can use Invoke to execute a method in a timely manner. Since a worker thread generally does not have a Windows message loop it can't use the mechanism provided by Invoke. To use Invoke the class derived from Control must have an underlying window handle. The Windows operating system associates your form with a window object internal to the operating system and it is identified by its handle. In order to allow one thread to wait for an action performed by another thread virtually all operating systems implement a synchronization mechanism. One of the most popular is the semaphore. You will find lots of theory on such synchronization techniques in any good book on operating systems. .NET provides not one, but several such mechanisms including semaphores. To cover all the theory and options would be inappropriate in this book, but I want to get you started by introducing one of the easiest to use mechanisms and a fun example.

The AutoResetEvent class allows threads to communicate with each other by signaling. A thread waits for a signal by calling the WaitOne method. If the AutoResetEvent is in the non-signaled state, the thread blocks, waiting for another thread to signal that the event has occurred by calling Set. Calling Set signals AutoResetEvent to release a waiting thread. AutoResetEvent remains signaled until a single waiting thread is released, and then automatically returns to the non-signaled state. If no threads are waiting, the state remains signaled indefinitely. You can control the initial state of an AutoResetEvent by passing a Boolean value to the constructor, true if the initial state is signaled and false otherwise.

My first example is a very simple console application that creates two worker threads that alternate the printing of the words "Flip" and "Flop." One thread prints "Flip" and the other "Flop" five times. Without synchronization we can't predict the output. It depends on the operating system's scheduling algorithm. Let's start by writing the program without synchronization. Then we will add the appropriate use of the AutoResetEvent. Figure 13-12 show the code. To
keep things simple I made everything static so no classes need to be instantiated. As you can see from the output in Figure 13-13 the thread running Flip executes first. Since it takes so little time to execute this loop the thread runs to completion and then Flop runs. If you change the loop counts to a large number you would see an alternation between threads, but not a strict Flip-Flop output.

```
namespace FlipFlop
{
    class FlipFlop
    {
        static void Main(string[] args)
        {
            Thread flip = new Thread(new ThreadStart(flipT));
            Thread flop = new Thread(new ThreadStart(flopT));
            flip.Start();
            flop.Start();
            flip.Join();
            flop.Join();
            Console.WriteLine("Finished");
        }
        static void flipT()
        {
            for(int i=0; i<5; ++i)
            {
                Console.WriteLine("Flip");
            }
        }
        static void flopT()
        {
            for (int i = 0; i < 5; ++i)
            {
                Console.WriteLine("Flop");
            }
        }
    }
}
```

Figure 13-12
Figure 13-14 shows the addition of the `AutoResetEvent` objects and their use. Notice that the event for `Flip` is initialized to `true` thus assuring that `Flip` will not block waiting for `Flop`. On the other hand, `Flop` is required to wait for `Flip` to complete one loop. If we didn't initialize in this manner then we would have a **deadlock** between the two threads, each waiting for the other. Figure 13-15 shows the output.

### FlipFlop - FlipFlop.cs
```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.IO;

namespace FlipFlop
{
    class FlipFlop
    {
        static AutoResetEvent flipE = new AutoResetEvent(true);
        static AutoResetEvent flopE = new AutoResetEvent(false);
        static void Main(string[] args)
        {
            Thread flip = new Thread(new ThreadStart(flipT));
            Thread flop = new Thread(new ThreadStart(flopT));
            flip.Start();
            flop.Start();
            flip.Join();
            flop.Join();
            Console.WriteLine("Finished");
        }
        static void flipT()
        {
            for(int i=0; i<5; ++i)
            {
                flipE.WaitOne();
                flopE.Set();
                Console.WriteLine("Flip");
            }
        }
    }
```
```csharp
static void flopT()
{
    for (int i = 0; i < 5; ++i)
    {
        flopE.WaitOne();
        Console.WriteLine("Flop");
        flipE.Set();
    }
}
```

The last aspect of this simple program is that the main thread waits for both the *Flip* and *Flop* threads to complete before outputting "Finished." It does this using the *Join* method of the thread class. Calling *Join* on a thread blocks the caller until the thread terminates.

**The Background Worker Thread Class**

The .NET FCL version 2.0 added a very convenient class for managing worker threads, *BackgroundWorkerThread*. It is available as an object in the toolbox. Just drag it to your form and you will see it below your form. It is not a visible control, but you can click on it to bring up the properties window. The appropriate code is added to the designer’s class to create a worker thread for you. There is very little you have to do to run code in the new worker thread.

*BackgroundWorkerThread* supports three key events as shown in Table 13-1. You will need *DoWork* and probably at least *RunWorkerCompleted*. Add these to your form in the usual way from the properties window for the *BackgroundWorkerThread* class. Place all the code you want the worker thread to execute in the *DoWork* event handler. No need to create a *ThreadStart* object or anything else.
Table 13-1 - Background Worker Thread Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoWork</td>
<td>Use this event handler to place the code to be executed by the worker thread.</td>
</tr>
<tr>
<td>ProgressChanged</td>
<td>This event can be triggered when the worker thread wants to report an incremental change in its progress.</td>
</tr>
<tr>
<td>RunWorkerCompleted</td>
<td>Triggered when the worker thread completes.</td>
</tr>
</tbody>
</table>

To start the worker thread call its `RunWorkerAsync` method this way:

```csharp
if (!bwt.IsBusy) bwt.RunWorkerAsync();
```

Checking `IsBusy` prevents starting up more than one worker thread at a time. The worker thread will run to completion. You can check `IsBusy` at any time to check on the progress, but it is probably more convenient to be notified asynchronously using the `RunWorkerCompleted` event. It will be triggered when the thread returns.

One more nifty event is provided specifically to aid in implementing progress bars or other progress monitors. The worker thread can trigger this event at any time and pass a percentage of completion as an event argument. It calls the `ReportProgress` method to trigger the event.

Figure 13-16 shows the code for a simple example to demonstrate all these features. It is yet another rework of our earlier example. A mouse click starts a worker thread that merely delays by counting on an integer. It reports its progress for every percent of its progress. A progress bar in the form is advanced for visual feedback as well as the cursor being set to the hour glass. In order to support the progress and cancel features two lines of code are necessary:

```csharp
bwt.WorkerReportsProgress = true;
bwt.WorkerSupportsCancellation = true;
```

I use a keyboard event handler to check for the escape key. If it is pressed the thread is alerted by calling `CancelAsync`. This sets `CancellationPending` to `true`. The worker thread must periodically check this property to determine if it should cancel.

```csharp
BackgroundWT – Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace BackgroundWT
{
    public partial class Form1 : Form
```
Figure 13-16

Figure 13-17 shows the form when the worker thread is running. Figure 13-18 shows what happens if the escape key is pressed. Note that the cursor has returned to the default arrow. If the worker thread completes without being cancelled the progress bar would be completely filled.
The Dining Philosophers
<to be added>