



## CHAPTER 1

---

# *Process and thread basics*

- 1.1 Background 2
- 1.2 Multitasking 10
- 1.3 Preemptive multitasking 16
- 1.4 Summary 23

There is a great deal of value in revisiting those things we “know” and exploring them in greater depth. Every developer is familiar with what a program is (we write them, after all) and what threads and processes are.

But it is a good exercise to review the basics, those things which are part of everyday language, before tackling the somewhat daunting topic of multithreaded development.

This chapter, by way of introduction, reviews operating system (OS) concepts, with a focus on processes and threads, and covers the basics of how threads do their work and how the processor switches between them.

The examples throughout this book are written in both C# and Visual Basic .NET, alternating between the two languages. All of the examples are available from the publisher’s web site at [www.manning.com/dennis](http://www.manning.com/dennis).

In this chapter, you’ll see code that’s devoted to relatively abstract concepts. The goal is to present examples that make the abstract concepts clearer and demonstrate them in a practical way.

## 1.1 **BACKGROUND**

A program, as you very well know, is typically defined as a series of instructions that are related in some way. In .NET terms, a program can be defined as an assembly, or group of assemblies, that work together to accomplish a task. Assemblies are nothing more than a way of packaging instructions into maintainable elements. An assembly is generally housed in a dynamic link library (DLL) or an executable.

**Program** A .NET program is an assembly, or group of assemblies, that perform a task. An assembly is nothing more than a packaging mechanism where pieces of related code are grouped into a common container, typically a file.

Closely related to programs are processes and threads. A program's execution occurs on one or more threads contained within a process. Threads allow the OS to exert control over processes and the threads that execute within.

### 1.1.1 **What is a process?**

A process gives a program a place to live, allowing access to memory and resources. It's that simple.

A process is an OS object used to associate one or more paths of execution with required resources, such as memory that stores values manipulated by threads that exist within the process.

A process provides a level of isolation that keeps different applications from inadvertently interacting with each other. Think of it in terms of cans of paint. Imagine you have several different colors of paint. While each color of paint is in its own can it cannot mix with other paints. The can is similar to a process in that it keeps things in the can contained within and things outside of the can out. Every process contains one or more threads. You can think of a thread as the moving part of the process. Without a thread interacting with elements within a process, nothing interesting will happen.

### 1.1.2 **What are threads and why should we care?**

Threads are paths of execution. The threads perform the operations while the process provides the isolation. A single-threaded application has only one thread of execution.

**Thread** A thread is the means by which a series of instructions are executed. A thread is created and managed by the OS based on instructions within the program. Every program will have at least one thread.

Let's take a step back and talk about how a program is loaded into a process. I'm not discussing Microsoft's implementation, but the things that need to occur and their likely order. When an executable is launched, perhaps by typing its name in a command window, the OS creates a process for the executable to run in. The OS then loads the executable into the process's memory and looks for an entry point, a specially marked place to start carrying out the instructions contained within the executable. Think of the entry point as the front door to a restaurant. Every restaurant has one, and front doors are relatively easy to find. Generally speaking, it's impossible to get

into a restaurant without going through the front door. Once the entry point is identified, a thread is created and associated with the process. The thread is started, executing the code located at the entry point. From that point on the thread follows the series of instructions. This first thread is referred to as the main thread of the process.

Listing 1.1 contains the listing of a console application that satisfies the obligatory Hello World example.

#### Listing 1.1 An example of a single-threaded application (VB.NET)

```
Module ModuleHelloWorld
    Sub Main()
        Console.Write("Hello")
        Console.Write(" World")
    End Sub
End Module
```

As a console application, all input and output pass through the command-line environment. Visual Basic console applications utilize the concept of a *module*. A module is a Visual Basic construct that is identical in functionality to a C# class having all static members. This means that the method can be invoked without an instance of the class having been created.

I've found it very beneficial, when dealing with .NET, to examine the Microsoft Intermediate Language (MSIL) the compiler produces. MSIL is an assembly-like language produced by compilers targeting the .NET environment. MSIL is translated to machine instructions by the runtime. MSIL is similar to Java's bytecode. Listing 1.2 contains the MSIL that corresponds to the Main subroutine in listing 1.1.

#### Listing 1.2 The MSIL produced by the Hello World example (MSIL)

```
.method public static void Main() cil managed ❶
{
    .entrypoint ❷
    .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() =
        ( 01 00 00 00 )
    // Code size          25 (0x19)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr      "Hello"
    IL_0006: call       void [mscorlib]System.Console::Write(string)
    IL_000b: nop
    IL_000c: ldstr      " World"
    IL_0011: call       void [mscorlib]System.Console::Write(string)
    IL_0016: nop
    IL_0017: nop
    IL_0018: ret
} // end of method ModuleHelloWorld::Main
```

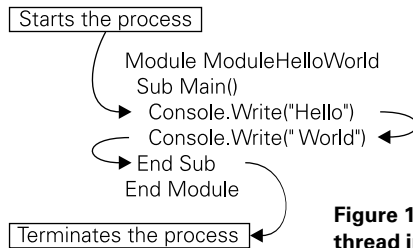
- 1 Notice the `static` keyword. This lets the runtime know that this is a static method. Since the method is defined within a module, it is implicitly shared/static.

Console applications require a static method be the entry point. A common approach is to have the console application contain a static `Main` that creates an instance of a class and invokes a method on that instance.

- 2 The `.entrypoint` directive indicates that this method is the entry point for the application. This tells the framework that this method should be invoked after the assembly is loaded into memory.

This example contains a single thread of execution that starts by entering the `Main` method and terminates when the `ret`, return, instruction executes. In this example the thread does not contain branching or looping. This makes it easy to see the path the thread will take.

Let's examine listing 1.1 in detail. Figure 1.1 shows the path the main thread of the process takes.



**Figure 1.1** The execution path the main thread in the Hello World example follows

The arrows show the path the thread takes during execution of the Hello World program. We're covering this in such depth because, when doing multithreaded development, it is critical to understand the execution path that a thread follows. When there is more than one path, the complexity increases. Each conditional statement introduces another possible path through the program. When there are a large number of paths, management can become extremely difficult. When the path a thread takes contains branching and looping, following that path often becomes more difficult. As a review, branching occurs when a conditional instruction is encountered. Looping is accomplished by having a branching statement target an instruction that has previously been executed. Listing 1.3 contains a slightly more complex version of the Hello World example.

### Listing 1.3 Hello World with a loop (C#)

```
using System;
namespace HelloWorldAgain
{
    class ClassHelloWorldAgain
    {
        [STAThread]
        static void Main(string[] args)
        {
```

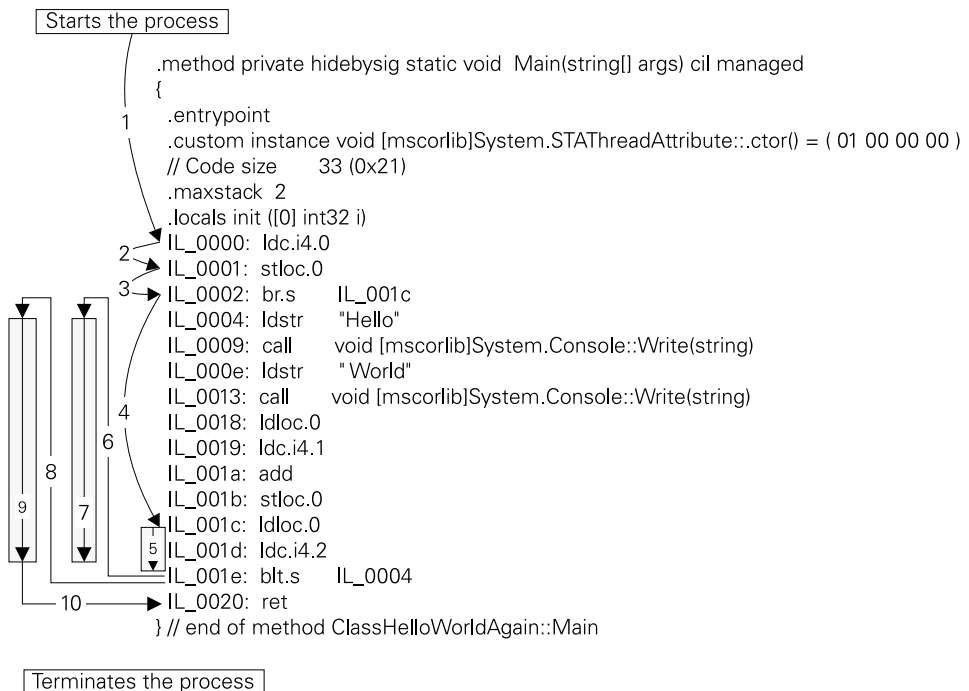
```

    for (int i=0;i<2;i++)
    {
        Console.WriteLine("Hello");
        Console.WriteLine(" World");
    }
}
}
}

```

It's easier to annotate the execution path by using the MSIL. Figure 1.2 contains the generated MSIL from listing 1.3 with numbered arrows indicating execution path.

This example demonstrates that code that is relatively simple can produce an execution path that is somewhat complex. The interesting part of this example is the jump that occurs at step 4. The reason for this jump is that the `for` loop tests to see if the test condition is true before the loop executes. The important thing to take away from this is that the main thread will execute steps 1 through 10. Those steps are the path the thread will take through the code.



**Figure 1.2 An execution path with branching**

### 1.1.3 The cat project

It's helpful to compare abstract things, like threads and processes, to something familiar. Imagine a housecat in a typical family residence. The cat spends most of its time sleeping, but occasionally it wakes up and performs some action, such as eating.

The house shares many characteristics with a process. It contains resources available to beings in it, such as a litter box. These resources are available to things within the house, but generally not to things outside the house. Things in the house are protected from things outside of the house. This level of isolation helps protect resources from misuse. One house can easily be differentiated from another by examining its address. Most important, houses contain things, such as furniture, litter boxes, and cats.

Cats perform actions. A cat interacts with elements in its environment, like the house it lives in. A housecat generally has a name. This helps identify it from other cats that might share the same household. It has access to some or the entire house depending on its owner's permission. A thread's access to elements may also be restricted based on permissions, in this case, the system's security settings. Listing 1.4 contains a class that models a cat.

**Listing 1.4 The ClassCat class models the behavior of a cat (C#).**

```
using System;
using System.Threading;
namespace Cat
{
    public class ClassCat
    {
        public delegate void DidSomething(string message);
        DidSomething notify;
        int sleepTime;
        string name;
        Random rnd;
        string[] actions=
        {
            "Eat",
            "Drink",
            "Take a bath",
            "Wander around",
            "Use litter box",
            "Look out window",
            "Scratch furniture",
            "Scratch carpet",
            "Play with toy",
            "Meow"
        };

        public ClassCat(string name, DidSomething notify )
        {
            sleepTime=1000;
            rnd=new Random(Environment.TickCount);
        }
    }
}
```

**1** The DidSomething delegate is used when an action occurs

**2** A list of possible actions is generated

**3** A name and a DidSomething delegate is passed in

```

        this.name = name;
        this.notify = notify;
    }

    private string WhichAction() ④ A random action
    {                             is chosen
        int which = rnd.Next(actions.Length);
        return actions[which];
    }

    public void DoCatStuff(int howMuch) ⑤ Loop the supplied
    {                                   number of times
        for (int i=0;i< howMuch;i++)
        {
            if(rnd.Next(100) >= 80)
            {
                notify(name + ": " + WhichAction()+ " " );
            }
            else
            {
                notify(name + ": Zzz ");
                Thread.Sleep(sleepTime);
            }
        }
    }
}

```

③ A name and a DidSomething delegate is passed in

- ① Since the cat does things, we need some way of letting the outside world know what it did. To accomplish this we use a *delegate*. A delegate is simply a way of accessing a method through a variable, similar in many ways to function pointers and callbacks. Function pointers and callbacks come from the C++ world. They provide a means of storing the information required to execute a function in a variable or parameter. This allows the function to be invoked indirectly, by accessing the variable or parameter. Cat owners may be wishing that their cat had a delegate available so that they could monitor their cat's activities.
- ② Cats do many things. I did not include sleep in this list of common feline activities since it occurs more frequently than the other activities.
- ③ Unlike the normal process through which cats come into the world, our cat is created when it is allocated using the new statement. The constructor accepts the name of the newly created cat along with a reference to the delegate to call when it does something. The advantage of using a delegate in this way is that the cat class doesn't need to know anything about the class that's utilizing its functionality.
- ④ The actions of a cat have always seemed pseudorandom to me. There may be a more complex algorithm they use to determine their actions but they aren't talking.

- 5 DoCatStuff is the main method used to simulate the cat's actions. It loops the specified number of times. Each loop has an 80 percent chance of the cat doing nothing more interesting than sleeping. The remaining 20 percent involves random selection from the list of actions we discussed earlier.

We're now ready to do something with our cat class. Listing 1.5 contains the code from a console application that utilizes ClassCat.

**Listing 1.5 The console application that uses the ClassCat class**

```
using System;
namespace Cat
{
    class ClassMain
    {
        [STAThread]
        static void Main(string[] args)
        {
            ClassCat theCat;
            ClassCat.DidSomething notify;
            notify = new ClassCat.DidSomething(AddLine);
            theCat = new ClassCat("Tiger", notify);
            theCat.DoCatStuff(250);
        }

        static private void AddLine(string message)
        {
            Console.Write(message);
        }
    }
}
```

1 Contains a reference to ClassCat

2 Creates an instance of ClassCat

Is invoked when an action occurs

- 1 Our cat will perform many actions. In order for the ClassMain class to know that the cat has performed an action, we must supply it with a delegate. The DidSomething delegate that's passed in to the constructor is invoked by the instance of the cat class whenever it accomplishes some task. The instance of the DidSomething delegate that's passed in is associated with the AddLine method. This method accepts a string as its only parameter. It then writes the contents of that string to the console.
- 2 When we create our cat we pass in the instance of the DidSomething delegate along with the cat's name. After we've created Tiger we tell it to do 250 iterations. This occurs on the main thread of the application. Once DoCatStuff completes, the application terminates. The following is a sample of the output produced by the program:

```
"Zzz" "Meow" "Zzz" "Zzz" "Zzz" "Play with toy" "Wander around" "Zzz" "Zzz"
"Take a bath" "Zzz" "Zzz" "Zzz" "Zzz" "Zzz" "Zzz" "Zzz" "Zzz" "Play with
toy" "Zzz"
```

We've explored a simple example of how a thread resembles a cat. In the next section we take a look at processes from the Task Manager perspective.



### 1.1.4 Task Manager

To see examples of processes, you need look no further than the Windows Task Manager, shown in figure 1.3.

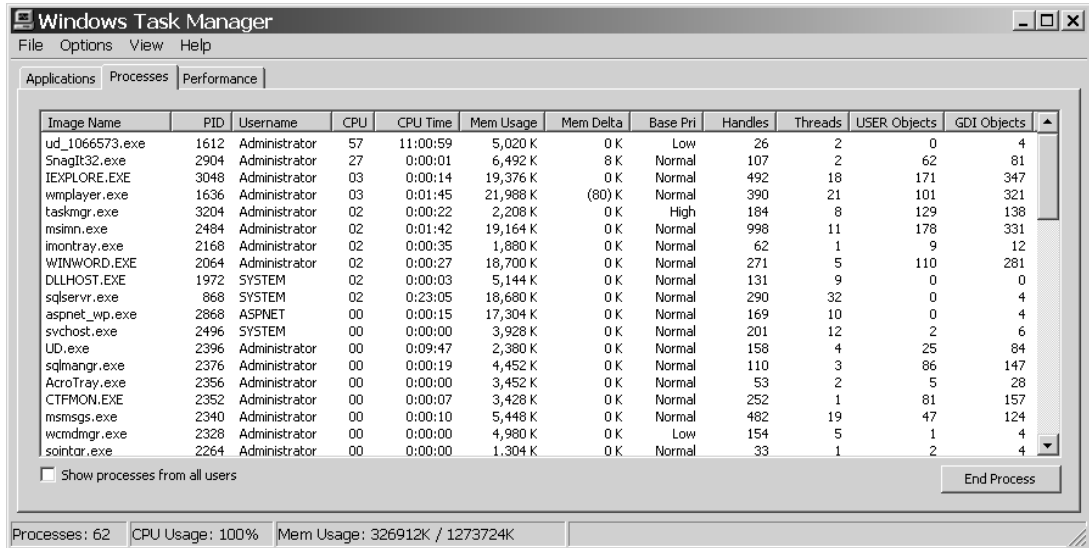


Image Name	PID	Username	CPU	CPU Time	Mem Usage	Mem Delta	Base Pri	Handles	Threads	USER Objects	GDI Objects
ud_1066573.exe	1612	Administrator	57	11:00:59	5,020 K	0 K	Low	26	2	0	4
Snagit32.exe	2904	Administrator	27	0:00:01	6,492 K	8 K	Normal	107	2	62	81
IEXPLORE.EXE	3048	Administrator	03	0:00:14	19,376 K	0 K	Normal	492	18	171	347
wmplayer.exe	1636	Administrator	03	0:01:45	21,988 K	(80) K	Normal	390	21	101	321
taskmgr.exe	3204	Administrator	02	0:00:22	2,208 K	0 K	High	184	8	129	138
msimn.exe	2484	Administrator	02	0:01:42	19,164 K	0 K	Normal	998	11	178	331
imontroy.exe	2168	Administrator	02	0:00:35	1,680 K	0 K	Normal	62	1	9	12
WINWORD.EXE	2064	Administrator	02	0:00:27	18,700 K	0 K	Normal	271	5	110	281
DLLHOST.EXE	1972	SYSTEM	02	0:00:03	5,144 K	0 K	Normal	131	9	0	0
sqlservr.exe	868	SYSTEM	02	0:23:05	18,680 K	0 K	Normal	290	32	0	4
aspnet_wp.exe	2868	ASPNET	00	0:00:15	17,304 K	0 K	Normal	169	10	0	4
svchost.exe	2496	SYSTEM	00	0:00:00	3,928 K	0 K	Normal	201	12	2	6
UD.exe	2396	Administrator	00	0:09:47	2,380 K	0 K	Normal	158	4	25	84
sqlmangr.exe	2376	Administrator	00	0:00:19	4,452 K	0 K	Normal	110	3	86	147
AcroTray.exe	2356	Administrator	00	0:00:00	3,452 K	0 K	Normal	53	2	5	28
CTFMON.EXE	2352	Administrator	00	0:00:07	3,428 K	0 K	Normal	252	1	81	157
msmsgs.exe	2340	Administrator	00	0:00:10	5,448 K	0 K	Normal	482	19	47	124
wcmdmgr.exe	2328	Administrator	00	0:00:00	4,980 K	0 K	Low	154	5	1	4
sointor.exe	2264	Administrator	00	0:00:00	1,304 K	0 K	Normal	33	1	2	4

☐ Show processes from all users

Processes: 62   CPU Usage: 100%   Mem Usage: 326912K / 1273724K

**Figure 1.3** Windows Task Manager lists the processes that are currently executing.

Processes are assigned a priority that is used in scheduling its threads. In figure 1.3 the column Base Pri contains the priority of the process. A process itself does not execute. Instead the threads contained within a process execute. Their execution is controlled in part by their priority. The OS combines each thread's priority with that of the process containing them to determine the order in which the threads should execute. Three of the most common values for base priority—High, Normal, and Low—are listed in figure 1.3.

The columns Mem Usage, Handles, USER Objects, and GDI Objects are examples of memory and resources that a process uses. These resources include things like file handles and Graphical Device Interface (GDI) objects. A file handle is used to interact with a file system file while a GDI object is used to display graphical output, such as circles and lines, on the screen.

Processes allow the actions of one thread in a process to be isolated from all other processes. The goal of this isolation is to increase the overall stability of the system. If a thread in a process encounters an error, the effects of that error should be limited to that process.

## 1.2 **MULTITASKING**

When computers ran only one program at a time, there was no need to be concerned with multitasking. Not that long ago a computer executed only one process—a single task—at a time. In the days of DOS the computer started up to a command prompt. From that prompt you typed the name of the program to execute. This single tasking made it very difficult to interact with multiple programs. Typically users were forced to exit one program, saving their work, and start another. For many it is unimaginable that a computer could run only a single program at once, such as a word processor or spreadsheet. Today users routinely execute a relatively large number of processes at the same time. A typical user may be surfing the Web, chatting using an instant messaging program, listening to an MP3, and checking email simultaneously.

When an OS supports execution of multiple concurrent processes it is said to be multitasking. There are two common forms of multitasking: preemptive and cooperative, which we'll explore next.

### 1.2.1 **Cooperative multitasking**

Cooperative multitasking is based on the assumption that all processes in a system will share the computer fairly. Each process is expected to yield control back to the system at a frequent interval. Windows 3.x was a cooperative multitasking system.

The problem with cooperative multitasking is that not all software developers followed the rules. A program that didn't return control to the system, or did so infrequently, could make the entire system unusable. That's why Windows 3.x would occasionally "freeze up," becoming unresponsive. This occurred because the entire OS shared a common thread processing messages. When Windows 3.x started a new application, that application was invoked from the main thread. The OS would pass control to the application with the understanding it would be returned quickly. If the application failed to return control to the OS in a timely fashion, all other applications, as well as the OS, could no longer execute instructions.

Development of applications for Window 3.x was more difficult than newer versions because of the requirements of cooperative multitasking. The developer was required to process Windows messages on a frequent basis, requiring that checks to the message loop be performed regularly. To perform long-running operations, such as looping 100 times, required performing a small unit of work, and then posting a message back to yourself indicating what you should do next. This required that all work be broken up into small units, something that isn't always feasible.

Let's review the way that current Windows applications function. The main thread executes a loop called a message pump. This loop checks a message queue to see if there's work to do. If so, it performs the work. The click event, which occurs when a user clicks a control such as a button, enters work into the message queue indicating which method should be executed in response to the user's click. This method is known as an event handler. While the loop is executing an event handler, it cannot process additional messages.

Think of the message pump as a person whose job is repairing appliances. Imagine this person has an answering machine at his place of business. When people need the technician, they call the answering machine and leave a message. This is essentially what happens when an event is entered into the message queue. The technician then retrieves messages from the answering machine, and, hopefully, responds in the order they were received. Generally, while the technician is on a service call he cannot start working on additional service calls. He must finish the current job and return to the office to check for messages.

Suppose a repair is taking a long time to complete. The client might tell the technician, go back to your office, check your messages, and do one job. Once you've finished it, come back here and finish this job. This is what the `Application.DoEvents` method does. It makes a call back to the message pump to retrieve messages.

Listing 1.6 contains the class that controls the sharing of the processor in a cooperative multitasking application.

#### Listing 1.6 A cooperative multitasking controlling class (C#)

```
using System;
using System.Collections;
namespace CooperativeMultitasking
{
    public class Sharing
    {
        public bool timeToStop=false;

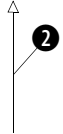
        ArrayList workers; ❶ An ArrayList is
        int current;         used to store
        public Sharing()     the workers
        {
            workers=new ArrayList(); ❶ An ArrayList is
            current=-1;             used to store
        }                          the workers
        public void Add(WorkerBase worker) ❶ An ArrayList is
        {                                  used to store
            workers.Add(worker);           the workers
        }

        public void Run()
        {
            if (workers.Count ==0)
            {
                return;
            }
            while (!timeToStop)
            {
                current++;
                if (current+1 > workers.Count)
                {
                    current= 0;
                }
            }
        }
    }
}
```

```

        WorkerBase worker;
        worker=(WorkerBase)workers[current];
        worker.DoWork(this);
    }
}
}

```



**2 Each worker is given a chance to work**

- ❶ Since multitasking involves multiple elements we need some way of storing them. In this example we use an `ArrayList` to store instances of classes derived from `WorkerBase`. An `ArrayList` is a dynamic array that manages the memory required to store its elements. To add an entry to the list you use the `Add` method. We discuss `WorkerBase` in listing 1.7.
- ❷ The heart of the `Sharing` class is the `Run` method which executes until the value of `timeToStop` becomes `true`. On each pass the variable `current`'s contents are incremented. This counter is used to choose which worker will be allowed to do a portion of its work. The worker is extracted from the `ArrayList` and its `DoWork` method is invoked.

`WorkerBase` is an abstract base class. All instances of classes that are managed by the `Sharing` class must be derived from the `WorkerBase` class, either directly or indirectly. Listing 1.7 contains the `WorkerBase` class.

**Listing 1.7 WorkerBase is the foundation for all classes controlled by the Sharing class (C#).**

```

namespace CooperativeMultitasking
{
    public abstract class WorkerBase
    {
        public abstract void DoWork(Sharing controller);
    }
}

```

Because `WorkerBase` contains an abstract method `DoWork`, all classes derived from it must implement that method. The `Sharing` class calls the `DoWork` method each time it's the worker class's turn. To perform some work we need a class that's derived from `WorkerBase` that does something. Listing 1.8 contains a class that writes out a greeting based on a string passed to its constructor.

### Listing 1.8 A cooperative greeter (VB.NET)

```
Public Class Hello
    Inherits WorkerBase

    Private name As String

    Public Sub New(ByVal name As String)
        Me.name = name
    End Sub

    Public Overrides Sub DoWork(ByVal controller As Sharing)
        Console.Write("Hello " + name)
    End Sub
End Class
```

Notice that the `DoWork` method is overridden to perform a simple action. Each time an instance of this class has a chance to perform its action, it will simply write out the greeting “Hello” followed by the name passed into the constructor.

To control termination we introduce a class that limits the number of times it is invoked (listing 1.9). This keeps our example relatively simple and shows another derived worker.

### Listing 1.9 A worker who signals it’s time to stop all processing (C#)

```
using System;
namespace CooperativeMultitasking
{
    public class Die : WorkerBase
    {
        int howManyAllowed;
        int workUnits;
        public Die(int howManyAllowed)
        {
            workUnits=0;
            this.howManyAllowed= howManyAllowed;
        }
        public override void DoWork(Sharing controller)
        {
            workUnits++;
            if (workUnits > howManyAllowed)
            {
                controller.timeToStop=true;
            }
        }
    }
}
```

The problem with cooperative multitasking is when one of the elements being controlled executes for an excessive amount of time. Listing 1.10 contains an example of a class that contains an infinite loop in its `DoWork` method.

#### Listing 1.10 A worker that uses more processing time than he should (VB.NET)

```
Public Class Bad
    Inherits WorkerBase
    Public Overrides Sub DoWork(ByVal controller As Sharing)
        While (True)
            End While
        End Sub
    End Class
```

---

We're now ready to see all the pieces tied together. The controlling part of this example is in listing 1.11. Notice that the line adding the `badWorker` is commented out.

#### Listing 1.11 The Sharing example main class (C#)

```
using System;
namespace CooperativeMultitasking
{
    class ClassMain
    {
        [STAThread]
        static void Main(string[] args)
        {
            Sharing controller;
            controller = new Sharing();
            Hello hiNewton = new Hello("Newton ");
            Hello hiCayle = new Hello("Cayle ");
            Die terminator = new Die(10);
            Bad badWorker = new Bad();
            controller.Add(hiNewton );
            controller.Add(hiCayle );
            // controller.Add(badWorker);
            controller.Add(terminator);
            controller.Run();
        }
    }
}
```

---

This program produces the following output:

```
Hello Newton Hello Cayle Hello Newton Hello Cayle Hello Newton Hello Cayle
Hello Newton Hello Cayle Hello Newton Hello Cayle Hello Newton Hello Cayle
Hello Newton Hello Cayle Hello Newton Hello Cayle Hello Newton Hello Cayle
Hello Newton Hello Cayle Hello Newton Hello Cayle
```

Notice that the greetings alternate as each worker is given a chance to do his work. When the `badWorker` is present in the collection of workers, the following output is produced:

```
Hello Newton Hello Cayle
```

Since the `badWorker`'s `DoWork` method never returns, the entire cooperative system is destabilized. This kind of failure is why Windows 3.x would occasionally freeze, requiring a reboot of the computer to recover.

We've discussed the challenges of developing applications under cooperative multitasking. The biggest problem is that if one or more applications doesn't follow the rules, the entire OS is affected. It's not surprising that all modern multitasking OSs use preemptive multitasking.

## 1.2.2 Preemptive

Preemptive multitasking is the more common form of multitasking in use today. Instead of relying on the programs to return control to the system at regular intervals, the OS takes it. Listing 1.12 contains an example of a program that uses threads and relies on preemptive multitasking.

**Listing 1.12 Using a different thread to perform the work (C#)**

```
private void button1_Click(object sender, System.EventArgs e)
{
    System.Threading.WaitCallback callback;
    callback = new System.Threading.WaitCallback(Loop);
    System.Threading.ThreadPool.QueueUserWorkItem(callback); ❶ Adds to the
                                                                ThreadPool
}

private void Loop(object state) ❷ Defines the method
{
    for (int i=1;i<100;i++)
    {
        for (int k=0;k< 100;k++)
        {
            double d;
            d = (double)k/(double)i;
            SetLabel(d.ToString()); ❸ Sets the text
                                                                of the label
        }
    }
    SetLabel("Finished"); ❸ Sets the text
                                                                of the label
}

private delegate void SetLabelDelegate(string s);
private void SetLabel(string s) ❸ Sets the text
{
    if (label1.InvokeRequired)
    {
        label1.Invoke(new SetLabelDelegate(SetLabel),new object[] {s});
    }
    else
    {
        label1.Text=s;
    }
}
```

The key element in this example is that the `button1_Click` method doesn't do the actual looping; instead it creates a work item that's entered into a thread pool. A thread pool is an easy way to do multithreading. As with most things, this simplicity results in a less flexible way of doing things. This execution occurs on a separate thread and is periodically interrupted by the OS to allow other threads a chance to get work done.

- ❶ Thread pools are a great way to perform multithreaded programming. Chapter 10 covers thread pools in detail. Thread pools perform their work using the `WaitCallback` delegate. A method that accepts a single parameter is associated with the `WaitCallback`. That method, `Loop`, is invoked on a thread controlled by the thread pool.
- ❷ The `Loop` method performs the actual work. It is very similar to the method in listing 1.6. The most notable difference is that there is no call to `Application.DoEvents`. Additionally, some type casting is being performed to make the output more interesting.
- ❸ Instead of accessing the label directly to output the results, we use the `SetLabel` method. `SetLabel` ensures that the label is accessed on the same thread that created the form. It does this because Windows Forms are not thread-safe. The potential exists that something undesirable will occur if one thread—or more—manipulates a control on a Windows Form.

It's important to understand that this example would not work on a cooperative multitasking OS because there is no call to service the message pump or to yield control. In the next section we discuss how preemptive multitasking is done.

## 1.3 **PREEMPTIVE MULTITASKING**

When more than one application is executing, there must be some means of determining whose turn it is to execute. This is generally referred to as scheduling. Scheduling involves an element in one of two states: currently executing and waiting to execute. Under modern OSs scheduling is performed on a per-thread basis. This allows a single thread to be paused and then resumed. Only one thread can be executing at a given point in time. All other threads are waiting for their turn to execute. This allows the OS to exert a high degree of control over applications by controlling the execution of their threads.

### 1.3.1 **Time slice, or quantum**

Things are often not what they seem. When we go see a movie in a theater, the images seem to flow from one to another in a seamless way. In reality, many separate images are presented on the screen and our brain maps them together to form a continuous image.

OSs do a similar sleight of hand with threads. Multiple threads seem to execute at the same time. This is accomplished by giving each thread in the system a tiny amount of time to do its work and then switching to another one. This happens very quickly, and the user of the system is typically unaware that a switch has occurred. The amount of time a thread has to do its work is called a time slice, or quantum. The duration of



the time slice varies based on the OS installed and the speed of the central processor. Listing 1.13 demonstrates that threads are periodically interrupted.

**Listing 1.13 Detecting threads sharing a processor (VB.NET)**

```
Module ModuleTimeSlice
    Sub Main()
        Dim whatToOutput As String
        Dim i As Integer
        Dim lastTick As Long
        Dim newTickCount As Long
        Dim opsPerTick As Long
        Dim offbyone As Long
        lastTick = System.Environment.TickCount
        opsPerTick = 0
        offbyone = 0
        whatToOutput = ""
        For i = 1 To 1000000
            newTickCount = System.Environment.TickCount
            If (lastTick = newTickCount) Then
                opsPerTick += 1
            Else
                If (lastTick = (newTickCount + 1)) Then
                    offbyone += 1
                    opsPerTick += 1
                    lastTick = newTickCount
                Else
                    Dim output As String
                    Dim numTicks As Long
                    numTicks = newTickCount - lastTick
                    output = String.Format("{0} {1}", numTicks, opsPerTick)
                    whatToOutput += output + vbCrLf
                    opsPerTick = 0
                    lastTick = newTickCount
                End If
            End If
        Next
        Console.WriteLine("OffByOne = " + offbyone.ToString())
        Console.WriteLine(whatToOutput)
    End Sub
End Module
```

**1** Retrieves the TickCount before the start of the loop

**2** Loops a large number of times

**3** Compares the current TickCount to the last one

**4** Checks to see if the last TickCount is one tick greater than the current tick

**5** Records the number of operations performed

- 1** We start by retrieving the current tick from the OS. The `TickCount` property returns the number of milliseconds since the OS was rebooted. We store that value in the `lastTick` variable.
- 2** To see the breaks in execution, we loop for a large number of times. Too small of a number here would not demonstrate the breaks in execution, since the task could be completed quickly.

- ❸ The first thing we do on each iteration is retrieve and store the current tick count. The idea is to capture how many milliseconds have passed since the last time we retrieved the value. We then check to see if the value has changed. If it hasn't we increment the number of operations that have been performed while the values were equal.
- ❹ If the values have changed we check to see if the new value is one greater than the old value. This would indicate that we moved from one millisecond to the next greater one. In my testing this didn't occur. This is as an indication that the amount of time the processor gives a thread is smaller than 1 millisecond.
- ❺ When a break of more than 1 millisecond occurs we determine the number of milliseconds that have elapsed and then record the results to a string and reset the counters. The frequency of this occurrence is a product of the load of the system, the power of the processor, and the number of iterations in the loop.

Listing 1.13 produces the following output:

OffByOne = 0	
16	177655
31	0
16	220041
15	395763

The first column contains how many milliseconds have passed when a break in the tick count occurred. The second column contains the number of iterations that were completed without a break occurring. If the thread had a processor dedicated to it there would be very even breaks, or not at all, in the tick count. As you can see, the breaks that do occur have a small amount of time between them. The amount of time a thread gets is based on the priority of the process it is executing in along with the priority associated with the thread.

A time slice is a very small unit of time. This helps provide the illusion that a thread has exclusive use of a processor. Each time that a processor switches from one thread to another is referred to as a context switch. In the next section we discuss context switching.

### 1.3.2 Context and context switching

There are many threads in existence in a typical system at any given point. A count of the threads from figure 1.3 yields over a hundred. Fortunately newer versions of Windows are good at dealing with multiple threads. A single processor executes one thread at a time. The thread has the processor's attention for one quantum, a time slice. After each quantum unit passes, the processor checks to see if another thread should have the processor. When the processor decides that a different thread should be executed, it saves the information the current thread requires to continue and switches to a different thread. This is called a context switch.

A high level of context switching is an indication of system load. A system that is switching excessively is said to be thrashing. The implication is that the processor is spending a great deal of time switching between threads and not performing as much work as if it were switching less frequently. High levels of context switching are generally associated with a shared resource being overutilized. When a resource isn't available, the OS pauses the thread that's requesting it. This allows other threads, which most likely aren't waiting for a resource, to execute.

One way that a context switch occurs is when a thread indicates that it has finished processing and that some other thread should be given the remainder of its time. This is accomplished using the `Sleep` method of the `Thread` class.

We'll discuss this in greater detail in section 5.3, but for now think of `Sleep` as a way for a thread to let the OS know that it would like to be idle for some period of time. The idea is that the thread detects that it should pause for a small amount of time to allow other things to happen. For example, if a thread is tasked with keeping a queue empty, it might pause periodically to allow multiple entries to be entered into the queue.

`Sleep` accepts several different types of parameters. One version of `Sleep` accepts an `Integer` indicating how many milliseconds the thread would like to be idle. If zero is passed in, it indicates that the thread wishes to yield the remainder of its time slice and continue executing on the next available time slice. This causes a context switch to occur. Listing 1.14 contains a class that uses a thread pool to execute a method on a different thread. The method continues to execute until changing the value of a Boolean flag stops it. The method calls `Sleep` with zero, which forces the thread to release the remainder of the current time slice to the operating system, forcing a context switch.

**Listing 1.14** A class that generates a large number of context switches (C#)

```
using System;
using System.Threading;
namespace ContextSwitching
{
    public class Switching
    {
        private bool itsTimeToStop ;
        public bool TimeToStop
        {
            get {return itsTimeToStop; }
            set {itsTimeToStop=value; }
        }

        public Switching()
        {
            itsTimeToStop=false;
            WaitCallback callback;
            callback = new WaitCallback(Loop);
            ThreadPool.QueueUserWorkItem(callback);
        }
    }
}
```

**1** `itsTimeToStop` controls the Loop method

**1** `itsTimeToStop` controls the Loop method

**2** `WaitCallback` is used with thread pools

```

        private void Loop(object state)
        {
            Thread.Sleep(500);
            while (!itsTimeToStop)
            {
                Thread.Sleep(0);
            }
        }
    }
}

```

**3** The Loop method executes until itsTimeToStop is true

- 1** An important element of any thread is being able to control its termination. We use the `itsTimeToStop` flag to control the termination of the thread. Initially `itsTimeToStop` is set to `false`, indicating that the `Loop` method should continue executing. To avoid interacting with the variable directly we use a property to manipulate its value. This is a good practice in general, and very important when dealing with multi-threaded development. This allows for a higher degree of control.
- 2** To create a separate thread of execution we use a thread pool. These are the same steps we used in listing 1.12.
- 3** The `Loop` method contains a `Sleep` statement that pauses execution for half of a second and then enters a loop where the current thread continually yields its time slice to the processor. To test the effects of this class on a system, we use a simple console application. Listing 1.15 contains the code of the console application that creates instances of the `Switching` class.

#### Listing 1.15 Console application that demonstrates context switching (C#)

```

using System;
namespace ContextSwitching
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            RunTest(10);
            RunTest(5);
            RunTest(3);
            RunTest(1);
            RunTest(0);
        }
        static void RunTest(int numberOfWorkers )
        {
            string howMany;
            howMany= numberOfWorkers.ToString();
            long i;
            Switching[] switcher;

```

**1** The RunTest method is called with different parameters

**2** An array of Switching class is created

```

        switcher = new Switching(numberOfWorkers);
        for (i = 0; i < switcher.Length; i++)
        {
            switcher[i] = new Switching();
        }
        Console.WriteLine("Created " + howMany + " workers");
        System.Threading.Thread.Sleep(5000);
        for (i = 0; i < switcher.Length; i++)
        {
            switcher[i].TimeToStop = true;
        }
        Console.WriteLine("Stopped " + howMany + " workers");
        System.Threading.Thread.Sleep(5000);
    }
}

```

- ❶ We call the `RunTest` method with a different parameter to create a different number of workers. This demonstrates a varying level of context switching.
- ❷ The `RunTest` method creates an array of `Switching` objects, from listing 1.14. We then pause the main thread for five seconds. This gives time for the other threads to execute. After five seconds we set the `TimeToStop` property to false for each `Switching` object.

This program writes the following output to the console:

```

Created 10 workers
Stopped 10 workers
Created 5 workers
Stopped 5 workers
Created 3 workers
Stopped 3 workers
Created 1 workers
Stopped 1 workers
Created 0 workers
Stopped 0 workers

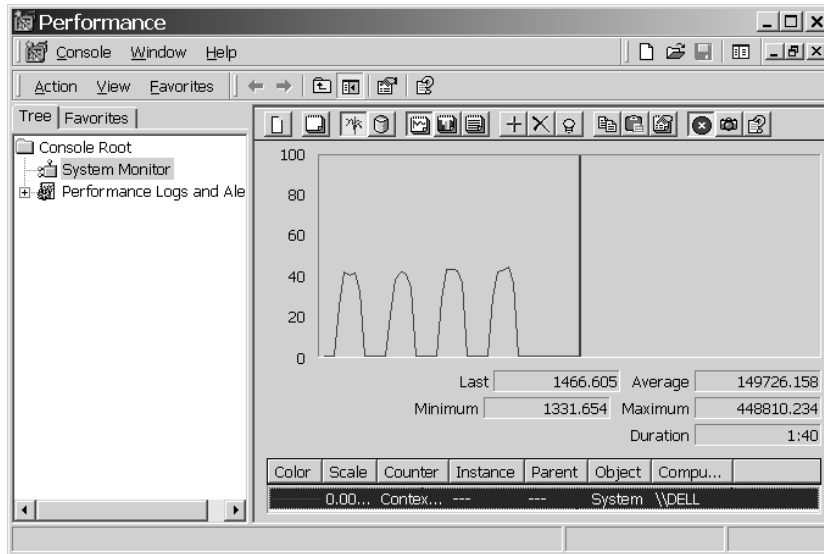
```

We've reviewed what a context switch is; now let's examine how we can measure them.

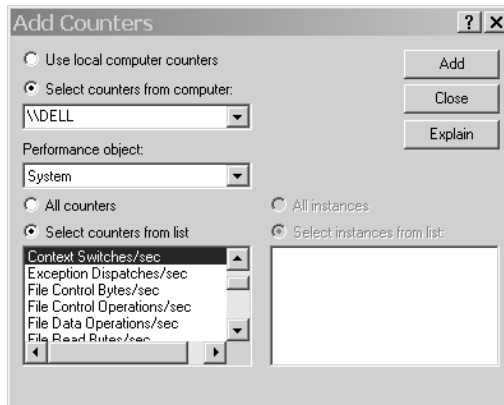
### 1.3.3 Detecting context switching

The Performance Monitoring program (`perfmon.exe`) is useful in determining how many context switches are occurring per second. In Windows 2000 the Performance Monitoring program is located in the Administrative Tools group under Programs in the Start menu. Figure 1.4 shows the impact of executing the program in listing 1.9.

The four “bumps” in the graph occurred during the time between when Created x Workers was written to the console and when Stopped x Workers was written to the console. Not surprisingly, the execution of zero workers did not produce a bump.



**Figure 1.4** Performance Monitor during listing 1.9. The “bumps” correspond to the time between Created and Stopped.



**Figure 1.5**  
The Add Counter dialog box used to add Context Switches / sec to a Performance Monitor graph

Measuring the number of context switches that occur per second is a good way of troubleshooting an application. Figure 1.5 shows how to add the measure to Performance Monitor.

The OS determines when a context switch occurs. A thread can give the scheduler a hint that it has finished performing its operations, but it’s up to the scheduler to determine if it will perform the context switch.

For more information on context switches, time slices, and thread scheduling, consult any book that covers the Windows platform.

## **1.4 SUMMARY**

This chapter serves as a review of the basic operating system concepts that relate to multithreaded development. It is by no means an exhaustive discussion but does serve to introduce the concepts. Understanding the underlying processes and threads is very important when you're doing multithreaded development. By being aware of how the OS interacts with threads you can develop programs that work with the OS rather than against it. By understanding what causes excessive context switching, you can develop programs that avoid that performance bottleneck.

In the next chapter we discuss the .NET framework from a multithreaded perspective.