16.2 Playing with the time in programs – introduction to threads

Operating systems like Windows XP, UNIX, and Linux allow several programs to be executed simultaneously on a computer. These operating systems can share the processor time between several executing programs. The executing programs are represented by independent processes that are controlled by the operating system. An operating system that is able to run several processes simultaneously is called a multitasking operating system. When you work with your personal computer, you may have several windows open on the screen, and each window may belong to a different application or program that is run as an independent process by the operating system. For example, if you run a C# program in a command prompt window, that program is an independent process, and at the same time the operating system can run other independent processes like an Internet browser process or a program editor process.

A multitasking operating system that is capable of running several processes simultaneously does not really execute the processes simultaneously, but it executes a process for a while, then stops the process, and puts the next waiting process into execution. A multitasking operating system has a list of processes it has to execute, and it gives proces-

These are sample pages from Kari Laitinen's book
"A Natural Introduction to Computer Programming with C#".
For more information, please visit
http://www.naturalprogramming.com/csbook.html

of pro- ans are ng sys- lled by cation) are inside the process, and those machine instructions are executed when the operating system decides to give processor time for the process. In addition to the application processes, the operating system executes special system processes that are needed for the proper operation of the computer. In fact, the operating system itself is also a process or a set of processes that get a share of the processor time.

It is essential in the concept of a process that processes are controlled by the operating system. An application process cannot start running by itself. The operating systems starts an application process after the user of the computer has commanded it to start the application. An application process, such as a C# program, can, however, create subprocesses that are called threads. A thread is also an independently running piece of program but it is not so "big a player" as a process is. You can think of threads as subprocesses within a process. As we shall see very soon, it is possible that the **`Main()`** method of a C# program creates threads that run simultaneously with the **`Main()`** method, which is itself a thread. Such a situation is described in Figure 16-4 where a C# application program is running as a process together with other processes, and inside the C# application process there are several threads running simultaneously.

On the following pages, you can find three example programs, **DotsAndDollars.cs**, **Playtime.cs**, and **Clock.cs**, in which the **`Main()`** method creates one or two threads. The method **`Main()`** is itself a thread in the mentioned programs. A thread can be created with the standard class **`Thread`** with a statement like

```
Thread thread_name  =
    new Thread( new ThreadStart( method_for_the_thread ) ) ;
```

This statement creates an object of type **`Thread`** by first creating an instance of type **`ThreadStart`**. If a thread is created with the above statement, and it is started with the statement

```
thread_name.Start() ;
```

a method named **`method_for_the_thread()`** starts executing simultaneously with the method that contains these statements.

`ThreadStart` is a so-called delegate type. A delegate type specifies a certain kind of method. An instance of a delegate type can contain a reference to a method. The `ThreadStart` type is such that only methods that take no parameters and that have void as their return type can be supplied as a parameter when a `ThreadStart` instance is created. Therefore, a method that can be executed concurrently as a separate thread must be of the form

```
static void method_for_the_thread()
{
   ...
}
```

or be a similar non-static method.

When a method that is executed concurrently as a separate thread terminates, the thread terminates, and you cannot restart the thread by calling again the `Start()` method. It is possible, and sometimes even necessary, to create threads that do not terminate. Such a thread can be created by using an infinite loop like

```
while ( true )
{
   ...
}
```

Infinite loops should not exist in conventional programs, but in applications that run several threads in parallel they can be considered appropriate. In programs **DotsAndDollars.cs** and **Clock.cs**, infinite loops are used in the methods that are executed as independent threads. The infinite loops are terminated when threads are aborted with the `Abort()` method. Program **Playtime.cs** shows how threads can be terminated with the help of a boolean variable, without using the `Abort()` method.

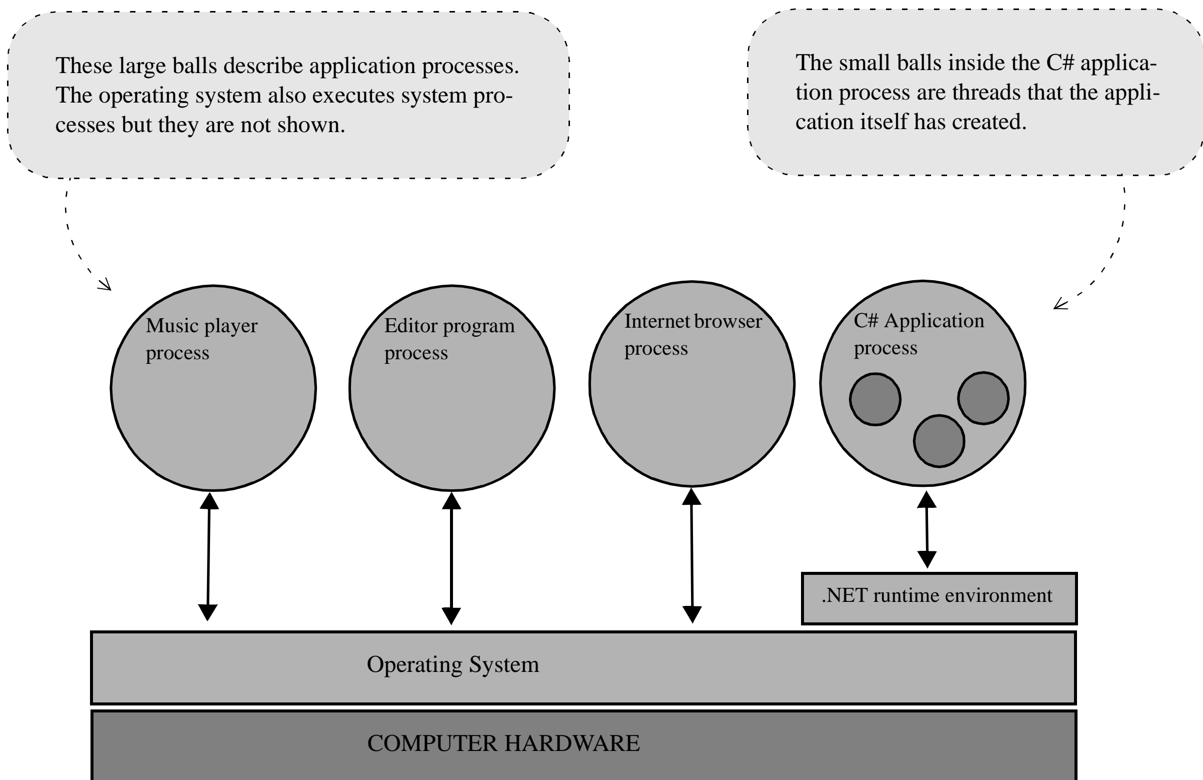

Figure 16-4. An operating system executing application processes concurrently.

```
// DotsAndDollars.cs

using System ;
using System.Threading ;

class DotsAndDollars
{
   static void print_dots()
   {
      while ( true )
      {
         Thread.Sleep( 1000 ) ; // Wait one second.
         Console.Write( " ." ) ;
      }
   }

   static void print_dollar_signs()
   {
      while ( true )
      {
         Thread.Sleep( 4050 ) ; // Wait 4.05 seconds.
         Console.Write( " $" ) ;
      }
   }

   static void Main()
   {
      ThreadStart  method_for_thread_to_print_dots =
                              new  ThreadStart( print_dots ) ;
      Thread thread_to_print_dots  =
               new Thread( method_for_thread_to_print_dots ) ;

      Thread thread_to_print_dollar_signs  =
               new Thread( new ThreadStart( print_dollar_signs ) ) ;

      thread_to_print_dots.Start() ;
      thread_to_print_dollar_signs.Start() ;

      Console.Write( "\n Press the Enter key to stop the program. \n\n" ) ;

      string  any_string_from_keyboard  =  Console.ReadLine() ;

      thread_to_print_dots.Abort() ;
      thread_to_print_dollar_signs.Abort() ;
   }
}
```

Method **print_dots()** represents a thread in this program. After the thread is created by method **Main()**, this method is executed independently. The **while** loops in this program are infinite loops that are terminated by aborting the threads.

Thread.Sleep() is a method with which a tread can suspend itself for a certain period of time. The sleeping times are specified in milliseconds.

This program starts executing like any other program, so that activities begin in the **Main()** method. However, after these statements are executed, there are three threads running in parallel, and each method of this program is executed as an independent thread. The task of method **Main()** is to wait until the user presses the Enter key.

The **Abort()** method of class **Thread** is used to terminate the two threads that are executing in parallel. The thread that runs the **Main()** method terminates automatically when the end of the **Main()** method is reached.

DotsAndDollars.cs - 1.+ A program that runs as three threads.

A new thread can be taken into use by first creating an object of the standard class **Thread**, and then calling method **Start()** for the **Thread** object. Before a **Thread** object can be created, it is necessary to specify which method is to be executed by the new thread. The mechanism for specifying the method is such that you first have to specify a so-called delegate of type **ThreadStart**, and then you supply that delegate to the constructor of class **Thread**.

```
ThreadStart  method_for_thread_to_print_dots =
                            new  ThreadStart( print_dots ) ;
Thread thread_to_print_dots  =
         new Thread( method_for_thread_to_print_dots ) ;

Thread thread_to_print_dollar_signs  =
         new Thread( new ThreadStart( print_dollar_signs ) ) ;
```

The other **Thread** object is created without giving a name to the delegate of type **ThreadStart**. An instance of delegate type **ThreadStart** is created first, and that delegate is passed to the constructor of class **Thread**. This object creation statement specifies that **print_dollar_signs()** is the method that is to be executed when the created thread is activated with the **Start()** method.

DotsAndDollars.cs - 1 - 1. Creation of the two Tread objects.

```
D:\csfiles3>DotsAndDollars

 Press the Enter key to stop the program.

 . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $
 . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $
 . . . . $ . . . . $ . . . . $ . . . . $ . . . . . $ . . . . $ . . . . $ . . . .
 $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . .
 $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . . $ . . . .
 $ . . . . . $ . . . . $ . . .
```

The dots are printed by the thread that executes method **print_dots()** and the dollar signs are printed by the thread that executes method **print_dollar_signs()**. Because the interval between the printing of dollar signs is 4.05 seconds and not exactly 4 seconds, after a certain time method **print_dots()** is executed 5 times before method **print_dollar_signs()** gets its turn.

DotsAndDollars.cs - X. The program has been executing here about 173 seconds.