Procesos y hebras

En este capítulo utilizaremos la infraestructura que pone a nuestra disposición un sistema operativo multitarea como Windows para dotar de paralelismo a nuestras aplicaciones, aun cuando éstas se ejecuten en un PC convencional, en el que suele haber un único microprocesador.

- Comenzaremos analizando los motivos que nos impulsan a la utilización de hebras y procesos en nuestras aplicaciones, que no son siempre los primeros en los que piensa la gente al hablar de concurrencia.

- A continuación, veremos cómo podemos ejecutar y controlar la ejecución de procesos gracias a la clase `System.Diagnostics.Process` incluida en la biblioteca de clases de la plataforma .NET.

- Acto seguido, pasaremos a estudiar el uso de múltiples hebras en el interior de un proceso. En el caso de la plataforma .NET, la clase `System.Thread` nos proporcionará la funcionalidad necesaria para construir aplicaciones multihebra.

- Finalmente, aprenderemos a usar mecanismos de sincronización que nos permitirán coordinar el trabajo realizado en distintas hebras. En capítulos posteriores, examinaremos con mayor detenimiento algunos de los mecanismos de comunicación entre procesos más utilizados.

Procesos y hebras

¿Por qué usar hebras y procesos?

Los programas secuenciales que constan de una única hebra de control resultan más fáciles de implementar y depurar que los programas con múltiples hebras que comparten, entre otros recursos, un mismo espacio de memoria. Eso resulta evidente. En el caso de existir paralelismo, ya sea a través de procesos independientes o mediante hebras dentro de un proceso, la ejecución de un programa no es determinista porque depende del tiempo de CPU que le asigna el sistema operativo a cada hebra. Como mencionamos en la introducción de esta parte del libro, el no determinismo introducido por el entrelazado de las operaciones de las distintas hebras provoca la aparición de errores difíciles de detectar y más aún de corregir. En definitiva, que la vida del programador resultaría mucho más sencilla si no hiciese falta la concurrencia.

Cuando la concurrencia se hace necesaria...

Supongamos que nuestra aplicación tiene que ocuparse de la realización de copias de seguridad de los datos con los que trabaja. Con una única hebra tendríamos que programar las copias de seguridad fuera del horario habitual de trabajo. ¿Y si el sistema tiene que funcionar las 24 horas del día? Con el uso de hebras, podemos aprovechar los períodos de inactividad de la CPU para ir haciendo la copia de seguridad mientras nuestra aplicación sigue funcionando de cara a sus usuarios.

Imaginemos ahora que nuestra aplicación se encarga de registrar las transacciones efectuadas en las cuestas de un banco (que pueden realizarse a cualquier hora del día gracias a los cajeros automáticos y las tarjetas de crédito). Además, las copias de seguridad han de realizarse muy a menudo, puede que de hora en hora. Para hacer una copia de seguridad, primero habrá que empaquetar los datos (en un archivo ZIP o algo así) y después transmitirlos a un lugar seguro (tal vez, en un refugio alojado en una cueva dentro de una montaña de granito, como hacen algunas empresas de cierta envergadura). Si en preparar los datos se tardan 30 minutos y, en enviarlos, otros 45 minutos, puede que de primeras nos inclinemos a pensar que la copia de seguridad no se puede hacer con la frecuencia solicitada (esto es, cada hora). O... ¿tal vez sí? Si dejamos encargada a una hebra/proceso de empaquetar los datos, a los 30 minutos podremos comenzar su transmisión en otra hebra/proceso, que requerirá otros 45 minutos para terminar la operación la primera vez. Una hora y cuarto en total. Sin embargo, mientras se están transmitiendo los datos, podemos comenzar a empaquetar los datos necesarios para la siguiente transmisión, de forma que cuando termine la transmisión actual tengamos ya preparados los datos correspondientes a la siguiente copia de seguridad. Algo para lo que, en principio, necesitaríamos una hora y cuarto, ahora, gracias al uso de paralelismo, somos capaces de hacerlo cada tres cuartos de hora. Menos incluso de lo que nos habían pedido y sin tener que cambiar el hardware ni contratar más ancho de banda.

Aunque el ejemplo anterior pueda inducir a pensar lo contrario, el uso de paralelismo mediante la utilización de hebras y procesos es algo más común de lo que nos podemos imaginar. La ejecución concurrente de procesos y hebras proporciona una amplia serie de ventajas frente a las limitaciones de los antiguos sistemas monotarea.

El diseño correcto de una aplicación concurrente puede permitir que un programa complete una mayor cantidad de trabajo en el mismo período de tiempo (como sucedía en el ejemplo antes mencionado) pero también sirve para otros menesteres más mundanos, desde la creación de interfaces que respondan mejor a las órdenes del usuario hasta la creación de aplicaciones que den servicio a varios clientes (como puede ser cualquier aplicación web a la que varios usuarios pueden acceder simultáneamente). En los siguientes párrafos intentaremos poner de manifiesto los motivos que nos pueden conducir al uso de hebras y procesos en nuestras aplicaciones.

De cara al usuario

El principal objetivo del uso de hebras o procesos es mejorar el rendimiento del sistema, y éste no siempre se mide en la cantidad de trabajo que se realiza por unidad de tiempo. De hecho, el uso de hebras se usa a menudo para reducir el tiempo de respuesta de una aplicación. En otras palabras, se suelen utilizar hebras para mejorar la interfaz de usuario.

Aunque pueda resultar sorprendente, el usuario es el principal motivo por el que utilizaremos hebras en nuestras aplicaciones con cierta asiduidad. Cuando una aplicación tiene que realizar alguna tarea larga, su interfaz debería seguir respondiendo a las órdenes que el usuario efectúe. La ventana de la aplicación debería refrescarse y no quedarse en blanco (como pasa demasiado a menudo). Los botones existentes para cancelar una operación deberían cancelar la operación de un modo inmediato (y no al cabo de un rato, cuando la operación ya ha terminado de todos modos).

Jakob Nielsen, por ejemplo, en su libro *Usability Engineering*, sugiere que cualquier operación que se pueda efectuar en una décima de segundo puede considerarse inmediata, por lo que no requerirá mayor atención por nuestra parte al implementar la aplicación. Cuando una operación se puede realizar por completo en un segundo, se puede decir que no interrumpe demasiado el trabajo del usuario, si bien resulta conveniente utilizar, al menos, una barra de progreso. Sin embargo, en operaciones que tarden varios segundos, el usuario difícilmente mantendrá su atención y será aconsejable que nuestra aplicación sea capaz de efectuar la operación concurrentemente con otras tareas.

En demasiadas ocasiones, las aplicaciones no le prestan atención debida al usuario. Mientras están ocupadas haciendo algo, no responden como debieran a las solicitudes que realiza el usuario. Con frecuencia, además, la aplicación no responde pese a que lo que queramos hacer no tenga nada que ver con la tarea que nos está obstaculizando el trabajo. Estos bloqueos temporales se suelen deber a que la aplicación está programada con una única hebra y, mientras dura una operación costosa, esa hebra bloquea todo lo demás, congelando la interfaz

de usuario. Afortunadamente, el uso de múltiples hebras puede mejorar enormemente la satisfacción del usuario al manejar nuestras aplicaciones al permitirle continuar su trabajo sin interrupciones.

Un ejemplo típico es el caso de una aplicación con una interfaz gráfica de usuario, en la que hebras o procesos independientes pueden encargarse de realizar las operaciones costosas mientras que la hebra principal de la aplicación sigue gestionando los eventos procedientes de la interfaz de usuario.

De hecho, cuando una operación que ha de realizarse como respuesta a un evento es costosa, es recomendable crear una hebra independiente para realizar dicha operación. De ese modo, el control de la aplicación vuelve a manos del usuario de forma inmediata y se mejora notablemente el tiempo de respuesta de la aplicación, su latencia de cara al usuario. Cualquier operación que pueda bloquear nuestra aplicación durante un período de tiempo apreciable es recomendable que se realice en una hebra independiente, desde el acceso a algún recurso en red hasta la manipulación de ficheros de cierto tamaño que requieran el acceso a dispositivos mecánicos (como discos duros o CD-ROMs), simplemente porque nuestra aplicación debe mostrarse atenta a las peticiones del usuario.

Establecimiento de prioridades

Los sistemas operativos multitarea suelen permitir la posibilidad de asignarle una prioridad a cada proceso o hebra de forma que, en igualdad de condiciones, los procesos/hebras de mayor prioridad dispongan de más tiempo de CPU.

Como es lógico, siempre les asignaremos una prioridad mayor a las tareas que requieran una respuesta más rápida, que son las más importantes de cara al usuario. De esta forma, la aplicación responderá rápidamente a las peticiones del usuario y, sólo cuando la CPU esté inactiva, se realizarán las tareas menos prioritarias. Al fin y al cabo, la CPU pasa largos períodos de inactividad en una aplicación convencional, dado que la velocidad como tipógrafo del usuario y su habilidad con el ratón nunca se pueden comparar con la velocidad de un microprocesador actual.

El uso de prioridades nos permite, además, posponer tareas cuya finalización no sea urgente. Esto puede ser útil incluso en la realización de tareas invisibles para el usuario. En un método, en cuanto se tenga un resultado solicitado, se puede devolver el control de la hebra y realizar el trabajo pendiente en una hebra independiente, disminuyendo así la latencia en las llamadas de una parte a otra del sistema. El trabajo pendiente se realizará cuando el sistema esté menos ocupado.

Aprovechamiento de los recursos del sistema

Como ya se ha visto al motivar el uso de hebras en la construcción de interfaces de usuario, cuando se utiliza una sola hebra, el programa debe detener completamente la ejecución mientras espera a que se complete cada tarea. La aplicación se mantiene ocupada pese a que el microprocesador puede estar inactivo (por ejemplo, mientras espera la terminación de una operación de entrada/salida). Cuando se utilizasen varias hebras y/o procesos, los recursos de la máquina pueden aprovecharse mientras tanto.

Un programa implementado de forma eficiente debería ser capaz de hacer algo útil mientras espera que se termine la realización de alguna operación en un dispositivo lento. En este sentido, hemos de tener en cuenta que la CPU es el dispositivo más rápido del ordenador. Desde su punto de vista, todos los demás dispositivos del ordenador son lentos, desde una impresora (que también puede parecernos lenta a nosotros) hasta un disco duro con Ultra-DMA (un modo de acceso directo a memoria pensado precisamente para dejar libre la CPU mientras se realiza la transferencia de datos entre el disco y la memoria principal del ordenador). Cuando nos encontramos en un entorno distribuido, las operaciones de E/S son aún más lentas, ya que el acceso a un recurso disponible en otra máquina a través de una red está limitado por la carga de la máquina a la que accedemos y el ancho de banda del que dispongamos.

La mejora que en su día supusieron los sistemas operativos de tiempo compartido se debe precisamente a que los recursos del ordenador se comparten entre varias tareas y se pueden aprovechan mejor. Mientras preparamos un texto con nuestro procesador de textos favorito, podemos estar escuchando música en formato MP3 y nuestro cliente de correo electrónico puede estar pendiente de avisarnos en cuanto recibamos un nuevo mensaje.

Además, los sistemas operativos multitarea suelen incluir mecanismos específicos que se pueden utilizar para mejorar el aprovechamiento de los recursos del sistema. Los *pipelines* son un ejemplo destacado en este sentido. Se puede construir una cadena de tareas de tal forma que la salida de una tarea se utiliza como entrada de la siguiente. La comunicación entre las tareas se efectúa mediante un buffer (en vez de tener que acceder a disco) de forma que una tarea no tiene que esperar a que la anterior finalice su ejecución por completo. Las tareas pueden avanzar conforme disponen de datos de entrada. La mejora de eficiencia será lineal en un sistema multiprocesador, pero también será notable en un PC normal si las tareas experimentan retrasos producidos por fallos de página, operaciones de E/S o comunicaciones remotas.

Por último, es conveniente mencionar que mejorar el aprovechamiento de los recursos del sistema suele suponer también una mejora en el tiempo de respuesta de las aplicaciones. El uso de múltiples hebras o procesos en un servidor web, por ejemplo, permite que varios usuarios compartan el acceso al servidor y puedan navegar por Internet sin tener que esperar turno.

Paralelismo real

Un sistema multiprocesador dispone de varias CPUs, por lo cual, si nuestra aplicación somos capaces de descomponerla en varias hebras o procesos, el sistema operativo podrá asignar cada una a una de las CPUs del sistema una tarea diferente.

Aunque los multiprocesadores aún no son demasiado comunes, un PC convencional incorpora cierto grado de paralelismo. Se pueden encontrar procesadores especializados en una tarjeta gráfica, coprocesadores matemáticos con sus correspondientes juegos de instrucciones (por ejemplo, las extensiones MMX de Intel) y procesadores digitales de señales en cualquier tarjeta de sonido. Todos estos dispositivos permiten realizar tareas de forma paralela, si bien suelen requerir conocimientos detallados de programación a bajo nivel.

Aparte de los múltiples coprocesadores mencionados, los microprocesadores actuales suelen incluir soporte para el uso de hebras. Por ejemplo, algunas versiones del Pentium 4 de Intel incluyen *Simultaneous MultiThreading* (SMT), que Intel denomina comercialmente *Hyper-Threading*. Con SMT, un único microprocesador puede funcionar como si fuesen varios procesadores desde el punto de vista lógico. No se consigue el mismo rendimiento que con un multiprocesador, pero se mantienen más ocupadas las distintas unidades de ejecución de la CPU, que en cualquier procesador superescalar ya están preparadas para ejecutar varias operaciones en paralelo.

También podemos repartir el trabajo entre distintas máquinas de un sistema distribuido para mejorar el rendimiento de nuestras aplicaciones. Un *cluster*, por ejemplo, es un conjunto de ordenadores conectados entre sí al que se accede como si se tratase de un único ordenador (igual que un multiprocesador incluye varios procesadores controlador por un único sistema operativo).

Tanto si disponemos de un multiprocesador como si disponemos de un procesador SMT o de un cluster, el paralelismo que se obtiene es real, pero su aprovechamiento requiere el uso de hebras y procesos en nuestras aplicaciones .

Modularización: Paralelismo implícito

A veces, los motivos que nos llevan a utilizar hebras o procesos no tienen una justificación física, como puede ser reducir el tiempo de respuesta y aprovechar mejor el hardware del que disponemos. En ciertas ocasiones, un programa puede diseñarse con más comodidad como un conjunto de tareas independientes. Usando hebras o procesos, se simplifica la implementación de un sistema, especialmente si el sistema puede verse, desde el punto de vista lógico, como un conjunto de actividades realizadas concurrentemente.

Por poner un ejemplo más mundano, el problema de perder peso puede verse como una

combinación de dos actividades: hacer ejercicio y mantener una dieta equilibrada. Ambas deben realizarse durante el mismo período de tiempo, aunque no necesariamente a la vez. No tendría demasiado sentido hacer ejercicio una temporada sin mantener una dieta equilibrada y luego dejar de hacer ejercicio para ponernos a dieta.

En situaciones de este tipo, la concurrencia de varias actividades es la solución adecuada para un problema. En este sentido, la solución natural al problema implica concurrencia. Desde un punto de vista más informático, la descomposición de una aplicación en varias hebras o procesos puede resultar muy beneficiosa por varios motivos:

- Se puede simplificar la implementación de un sistema si una secuencia compleja de operaciones se puede descomponer en una serie de tareas independientes que se ejecuten concurrentemente.

- La independencia entre las tareas permite que éstas se implementen por separado y se reduzca el acoplamiento entre las distintas partes del sistema

- Un diseño modular facilita la incorporación de futuras ampliaciones en nuestras aplicaciones.

Aviso importante

El objetivo principal del uso de paralelismo es mejorar el rendimiento del sistema. Salvo que un diseño con hebras y procesos simplifique el trabajo del programador, lo cual no suele pasar a menudo en aplicaciones de gestión, para qué nos vamos a engañar, el diseñador de una aplicación concurrente deberá analizar hasta qué punto compensa utilizar hebras y procesos.

Los programas diseñados para aprovechar el paralelismo existente entre tareas que se ejecutan concurrentemente deben mejorar el tiempo de respuesta o la carga de trabajo que puede soportar el sistema. Esto requiere comprobar que no se esté sobrecargando el sistema al usar hebras y procesos. Un número excesivo de hebras o procesos puede llegar a degradar el rendimiento del sistema si se producen demasiados fallos de página al acceder a memoria o se pierde demasiado tiempo de CPU realizando cambios de contexto. Además, el rendimiento de una aplicación concurrente también puede verse seriamente afectado si no se utilizan adecuadamente los mecanismos de sincronización que son necesarios para permitir que distintas hebras y procesos accedan a recursos compartidos. Tal vez, en un exceso de celo por nuestra parte, hebras y procesos pasen bloqueados más tiempo del estrictamente necesario, con lo cual no se consigue el paralelismo perseguido.

En cualquier caso, ya hemos visto que el uso de paralelismo, y de hebras en particular, es más común de lo que podría pensarse en un principio (y menos de lo que debería). Una vez motivado su estudio, sólo nos falta ver cómo se pueden utilizar procesos y hebras en la plataforma .NET, tema al que dedicaremos el resto de este capítulo.

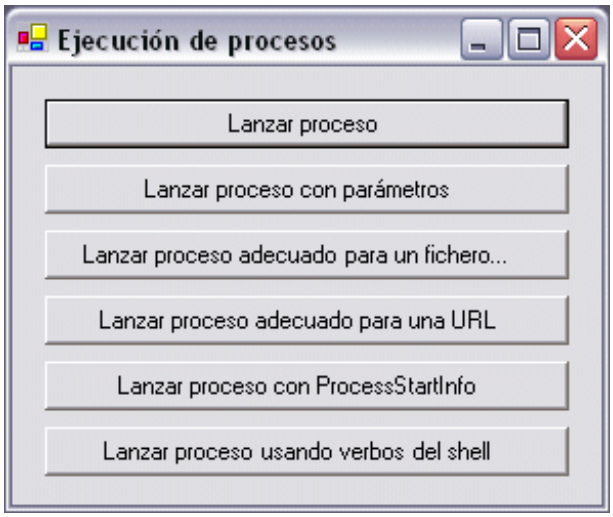
Procesos

La biblioteca de clases de la plataforma .NET incluye una clase, la clase `System.Diagnostics.Process`, que es la que nos permite crear procesos, controlar en cierto modo su ejecución e incluso acceder a información acerca de los procesos que se estén ejecutando en el sistema (la misma información a la que se puede acceder a través del Administrador de Tareas de Windows).

Ejecución de procesos

La clase `Process` incluye un método estático que podemos utilizar para comenzar la ejecución de un proceso de distintas formas: el método `Process.Start()`.

El uso del método `Process.Start()` equivale a la realización de una llamada a la función `ShellExecute`. Esta función, perteneciente al API estándar de Windows (Win32), es la que se utiliza en la familia de sistemas operativos de Microsoft para lanzar un proceso. Este hecho convierte al método `Process.Start()` en un método muy versátil que se puede utilizar de distintas formas, tal como veremos a continuación:



Distintas formas de ejecutar un proceso en Windows.

La forma más sencilla de utilizar el método `Process.Start()` es indicarle el nombre del fichero ejecutable correspondiente al proceso que queremos ejecutar, usualmente un fichero con extensión `.exe`, si bien también puede tener otras extensiones, algunas de las cuales se muestran en la tabla adjunta (que además se corresponden, precisamente, con las extensiones de los ficheros que no debemos abrir cuando nos llegan adjuntos a un mensaje de correo electrónico).

Extensión	Tipo de fichero
`.bat`	Macro (archivo de comandos MS-DOS)
`.cmd`	Macro (archivo de comandos NT)
`.com`	Ejecutable para MS-DOS
`.cpl`	Applet para el panel de control
`.dll`	Biblioteca de enlace dinámico
`.exe`	Aplicación
`.lnk`	Acceso directo (un fichero con el nombre de otro)
`.msi`	Paquete de instalación
`.pif`	Acceso directo a un programa MS-DOS
`.scr`	Salvapantallas

Para ejecutar un proceso basta, pues, con indicar el fichero que contiene el punto de entrada a la aplicación. Si este fichero está en un directorio incluido en el `PATH` del sistema, será suficiente con indicar el nombre del fichero. No hará falta poner su ruta de acceso porque el `PATH` es una variable de entorno que le indica al sistema operativo en qué carpetas o directorios ha de buscar los programas. El ejemplo siguiente muestra cómo podemos abrir una ventana del Internet Explorer desde nuestra aplicación de un modo trivial:

```
System.Diagnostics.Process.Start("iexplore.exe");
```

Obviamente, la ejecución de un proceso también se puede realizar indicándole una serie de parámetros. Estos parámetros son los que recibirá la función `main` de nuestro programa en forma vector de cadenas de caracteres:

```
string proceso = "iexplore.exe";
string args = "http://csharp.ikor.org/";

System.Diagnostics.Process.Start(proceso, args);
```

Además de permitirnos indicar los parámetros que queremos pasarle a un proceso, podemos especificar algunas propiedades adicionales sobre la ejecución de un proceso. Para ello, podemos recurrir a la clase `ProcessStartInfo`, que al igual que `Process`, también se encuentra en el espacio de nombres `System.Diagnostics`. Mediante esta clase podemos especificar el directorio en el que se ejecutará el proceso (`WorkingDirectory`) o el estado de la ventana que se utilizará al iniciar el proceso (`WindowStyle`), tal como muestra el siguiente ejemplo:

```
using System.Diagnostics;
...

ProcessStartInfo info = new ProcessStartInfo();

info.FileName = "iexplore.exe";
info.Arguments = "http://csharp.ikor.org/";
info.WindowStyle = ProcessWindowStyle.Maximized;

Process.Start(info);
```

Como ya mencionamos anteriormente, el método `Process.Start()` realiza una llamada a la función `ShellExec` del shell de Windows. Esta función es la que nos permite, por ejemplo, pinchar sobre un documento con extensión `.doc` y que automáticamente se abra el Word. Cuando el fichero no es ejecutable, la función `ShellExec` busca un programa que resulte adecuado para manipular el fichero indicado. De esta forma, el usuario no tiene que abrir primero un programa y luego acceder al documento con el que quiera trabajar, sino que basta con seleccionar el documento y automáticamente se abrirá con el programa adecuado. Este programa se determina a partir de las extensiones de archivo previamente registradas en el registro de Windows. El siguiente ejemplo muestra cómo podemos seleccionar un fichero cualquiera y que el sistema operativo se encargue de "ejecutarlo" de la forma adecuada lanzando un proceso que nos permita manipular el fichero en función de su extensión:

```
OpenFileDialog openFileDialog = new OpenFileDialog();

// Directorio de trabajo
// - Se comienza la exploración en C:\
openFileDialog.InitialDirectory = "c:\\";
// - Al final, se vuelve al directorio donde nos encontrásemos
openFileDialog.RestoreDirectory = true;

// Filtros
openFileDialog.Filter = "Ficheros de texto (*.txt)|*.txt"
                      +"|Todos los ficheros (*.*)|*.*";
openFileDialog.FilterIndex = 2 ;

if (openFileDialog.ShowDialog() == DialogResult.OK) {
   System.Diagnostics.Process.Start(openFileDialog.FileName);
}
```

En el ejemplo hemos utilizado una de las ventanas de diálogo típicas de Windows, al que en la plataforma .NET se accede mediante la clase `System.Windows.Forms.OpenFileDialog`. Esta clase nos permite seleccionar un fichero, especificando incluso los filtros que le permitimos usar al usuario para ver determinados tipos de ficheros. Estos filtros se especifican en el formato *descripción|plantilla*, donde la *descripción* corresponde a la cadena de caracteres que le aparecerá al usuario en una desplegable y la *plantilla* indica el nombre que han de tener los ficheros del tipo indicado por el filtro.

> Igual que existe un componente para mostrar un diálogo que nos permita seleccionar un fichero (`OpenFileDialog`), existen otros diálogos estándar que nos permiten guardar un fichero (`SaveFileDialog`), seleccionar una carpeta (`FolderBrowserDialog`), escoger un color (`ColorDialog`), elegir un tipo de letra (`FontDialog`), especificar dónde y cómo queremos imprimir algo (`PrintDialog`) o configurar la página para un trabajo de impresión (`PageSetupDialog`). Todos estos componentes forman parte de la biblioteca de "diálogos comunes" de Windows y heredan de la clase `System.Windows.Forms.CommonDialog`.

Igual que se puede abrir un fichero cualquiera utilizando `Process.Start()`, siempre y cuando la extensión del fichero haya sido registrada previamente en el Registro de Windows, también se puede abrir una URL cualquiera. Las URLs comienzan por la especificación de un protocolo y los protocolos más comunes también aparecen en el Registro de Windows: `http:` y `https:` para acceder a páginas web, `ftp:` para acceder a un servidor de archivos, `telnet:` para conectarse de forma remota a una máquina, `mailto:` para enviar un mensaje de correo electrónico o `nntp:` para acceder a un servidor de noticias, por poner algunos ejemplos. Por tanto, para abrir una página de Internet desde nuestra aplicación, podemos usar directamente su URL en la llamada al método `Process.Start()`:

```
System.Diagnostics.Process.Start("http://csharp.ikor.org/");
```

Cuando llamamos a este método usando un nombre de un fichero o una URL, en realidad lo que hacemos es ejecutar el proceso asociado a abrir ese fichero tal como aparezca en el Registro de Windows. Podemos ejecutar la aplicación `regedit.exe` para ver el contenido del Registro de Windows en nuestra máquina. Dentro del apartado*Mi PC/HKEY_CLASSES_ROOT* aparecen los distintos tipos de archivo registrados. Al registrar un tipo de archivo, aparte de indicar el programa con el que ha de abrirse, también se puede especificar cómo han de realizarse otras operaciones con el archivo. Es lo que se conoce con el nombre de "verbos del shell", algunos de los cuales se reproducen a continuación:

Verbo	Descripción
`open`	Abrir un fichero (ejecutar una aplicación, abrir un documento o explorar una carpeta)
`edit`	Editar un documento
`print`	Imprimir un documento
`explore`	Explorar una carpeta
`find`	Iniciar una búsqueda en la carpeta indicada

El verbo `open` corresponde a la acción que se realiza cuando hacemos click con el ratón sobre el nombre de un fichero, si bien se pueden definir cuantas acciones deseemos. Para efectuar esas acciones, lo único que tenemos que hacer es recurrir de nuevo a la clase `ProcessStartInfo`:

```
using System.Diagnostics;
...

ProcessStartInfo info = new ProcessStartInfo();

info.FileName = "Ade.jpg";
info.WorkingDirectory = "f://";
info.Verb = "edit"; // vs. "open" || "print"

Process.Start(info);
```

Si nos interesase crear nuestro propio tipo de archivos, podemos hacer que el usuario no tenga que abrir nuestra aplicación para después acceder al fichero con el que quiera trabajar. Windows nos ofrece la posibilidad de utilizar su interfaz orientada a documentos. Sólo tenemos que crear entradas correspondientes en el Registro, lo que podemos hacer desde nuestra propia aplicación por medio de las clases `Microsoft.Win32.Registry` y `Microsoft.Win32.RegistryKey`, que son las que nos permiten manipular el Registro del sistema.

Finalización de procesos

Cuando se utiliza el método `Process.Start()` para iniciar la ejecución de un proceso, también se debería llamar al método `Process.Close()` para liberar todos los recursos asociados al objeto de tipo `Process`. En los ejemplos del apartado anterior, podríamos liberar los recursos asociados al proceso lanzado siempre y cuando antes nos asegurásemos de que la ejecución del proceso ha finalizado. Para comprobar que la ejecución de un proceso ha finalizado disponemos de varias alternativas, como veremos a continuación.

Si no nos importase que nuestra aplicación detuviese su ejecución hasta que el proceso lanzado finalizase por completo, nos bastaría con llamar al método `WaitForExit`:

```
Process process = Process.Start(...);

process.WaitForExit();
process.Close();
```

No obstante, no parece muy buena idea detener la ejecución de nuestra aplicación de forma indefinida. El subproceso podría "colgarse"; esto es, entrar en un bucle infinito o se quedarse esperando la ocurrencia de algún tipo de evento de forma indefinida. Si esto ocurriese, nuestra aplicación dejaría de responder. Para evitar esta posibilidad, podemos limitar la espera a un período de tiempo máximo. Utilizando de nuevo el método `WaitForExit`, lo único que tenemos que hacer es incluir un parámetro en el que indicamos el número de milisegundos que estamos dispuestos a esperar.

Ejecución invisible de comandos MS-DOS

```
Process proceso = new Process()

string  salida = Application.StartupPath + "/output.txt";
string  path   = System.Environment.GetFolderPath
                    (Environment.SpecialFolder.System);

proceso.StartInfo.FileName = "cmd.exe";
proceso.StartInfo.Arguments =
  "/C dir \""+path+"\" >> \"" + salida +"\" && exit";

proceso.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
proceso.StartInfo.CreateNoWindow = true;
proceso.Start()

proceso.WaitForExit(1000);   // 1 segundo de timeout

if (!proceso.HasExited()) {
   proceso.Kill();           // Finalizamos el proceso
} else {
   ...
}
```

El ejemplo muestra cómo podemos lanzar una secuencia de comandos MS-DOS desde nuestra aplicación. Si el proceso no termina en el tiempo máximo que le indicamos, lo que comprobamos llamando al método `HasExited()`, entonces podemos obligar la finalización de la ejecución del proceso con una llamada al método `Kill()`.

El método `HasExited()` también puede emplearse para comprobar la finalización de la ejecución de un proceso sin tener que bloquear nuestra aplicación con una llamada a

`WaitForExit()`, pero tenemos que ser nosotros los que periódicamente sondeemos el estado del proceso para comprobar su terminación.

Existe una tercera opción para comprobar la finalización de la ejecución de un proceso: aprovechar el evento `Exited` incluido en la clase `Process`. De esta forma, no tenemos que preocuparnos de bloquear la ejecución de nuestra aplicación mientras esperamos ni de sondear periódicamente el estado del proceso. El evento `Exited` se produce automáticamente cuando termina la ejecución del proceso, por lo que podemos escribir un manejador para este evento que realice las tareas oportunas en el momento de la finalización de la ejecución del proceso. En el siguiente ejemplo, el método `ExitHandler` se limita mostrar un mensaje en el que figura el momento en el que terminó la ejecución del proceso:

```
Process proceso = new Process()
...
proceso.Exited += new EventHandler(ExitHandler);
proceso.Start()
...

public void ExitHandler(object sender, EventArgs e)
{
  Process proceso = (Process) sender;

  MessageBox.Show ( "PID: " + proceso.Id
                  + System.Environment.NewLine
                  + "Hora: " + proceso.ExitTime
                  + System.Environment.NewLine
                  + "Código: " + proceso.ExitCode );

  proceso.Close();
}
```

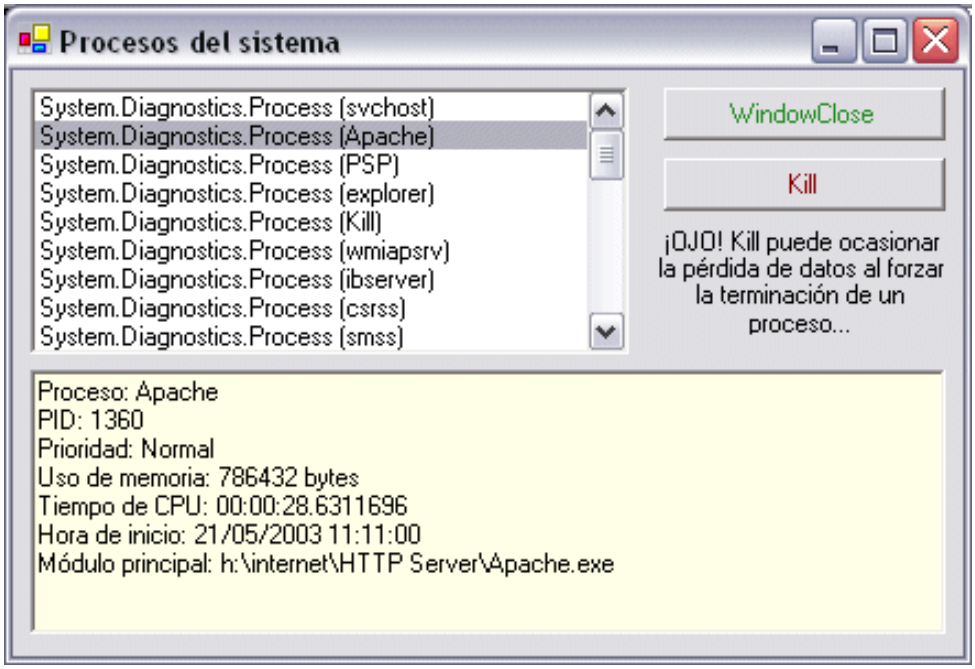
La clase `Process` nos permite provocar explícitamente la terminación de un proceso utilizando el método `Kill()` o el método `CloseMainWindow()`. Este último es el recomendado para una aplicación Windows, pues equivale a que el usuario de la aplicación cierre la ventana principal de ésta de la forma usual. Es decir, `CloseMainWindow()` genera el mismo evento que se produce cuando el propio usuario cierra la ventana de la aplicación. Este evento, a su vez, puede ocasionar que la aplicación realice algunas tareas de finalización que con `Kill()` nunca se efectuarían, como podría ser darle al usuario la oportunidad de guardar un fichero que ha sido modificado (o guardarlo de forma automática). En otras palabras, `Kill()` finaliza el proceso "por las bravas" y puede ocasionar la pérdida de datos que no hayan sido previamente guardados.

Monitorización de procesos

Como un lector observador ya habrá notado, la clase `Process` no sólo permite controlar la ejecución de un proceso, sino que también nos ofrece la posibilidad de acceder a información

relativa al proceso. Esta información es la misma que podemos obtener del Administrador de Tareas de Windows.

Tal como se muestra en la imagen, en este apartado construiremos un sencillo monitor de procesos en el que podamos ver datos relativos a los distintos procesos que se estén ejecutando en nuestra máquina.



Monitor de procesos

Nuestro monitor será una sencilla aplicación Windows con un único formulario. Sólo tenemos que crear la aplicación y añadir los componentes que se listan en la siguiente tabla:

Componente	Tipo	Propiedad	Valor
formProcess	Form	Text	Procesos del sistema
listProcesses	ListBox	ScrollAlwaysVisible	true
textInfo	TextBox	Multiline	true
		BackColor	Info
buttonClose	Button	Text	WindowClose
		ForeColor	ForestGreen

Componente	Tipo	Propiedad	Valor
buttonKill	Button	Text	Kill
		ForeColor	DarkRed
labelWarning	Label	TextAlign	MiddleCenter
		Text	¡OJO! Kill puede ocasionar la pérdida de datos al forzar la terminación de un proceso

Al comenzar la ejecución de la aplicación, rellenamos la lista `listProcesses` con los procesos que se estén ejecutando. Para ello, incluimos lo siguiente como respuesta al evento `FormLoad` de nuestro formulario:

```
Process[] procesos = Process.GetProcesses();

listProcesses.Items.AddRange(procesos);
```

Cuando el usuario selecciona un proceso concreto de la lista, hemos de mostrar la información detallada del proceso seleccionado en la caja de texto `textInfo`. Simplemente, definimos la respuesta de nuestra aplicación al evento `SelectedIndexChanged` de la lista `listProcesses`:

```
Process selected = (Process) listProcesses.SelectedItem;

textInfo.Clear();

if (selected!=null)
   textInfo.Lines = new string[] {
     "Proceso: " + selected.ProcessName,
     "PID: " + selected.Id,
         // "Máquina: " + selected.MachineName,
     "Prioridad: " + selected.PriorityClass,
     "Uso de memoria: " + selected.WorkingSet + " bytes",
         // selected.PagedMemorySize
         // selected.VirtualMemorySize
     "Tiempo de CPU: " + selected.TotalProcessorTime,
     "Hora de inicio: " + selected.StartTime,
     "Módulo principal: " + selected.MainModule.FileName
         // selected.MainWindowTitle
   };
```

Como se puede ver, la clase `Process` incluye numerosas propiedades que nos permiten controlar el estado y los recursos utilizados por un proceso.

Finalmente, implementamos el código necesario para forzar la finalización de un proceso

usando `Kill()` o `CloseMainWindow()` en función del botón que pulse el usuario:

```
// Fuerza la finalización del proceso
// (puede provocar la pérdida de datos)

private void buttonKill_Click(object sender, System.EventArgs e)
{
  Process selected = (Process) listProcesses.SelectedItem;

  if (selected!=null)
     selected.Kill();
}

// Solicita la terminación de un proceso
// (como si el usuario solicitase cerrar la aplicación)

private void buttonClose_Click(object sender, System.EventArgs e)
{
  Process selected = (Process) listProcesses.SelectedItem;

  if (selected!=null) {

     if (selected.Id==Process.GetCurrentProcess().Id)
         MessageBox.Show("Ha decidido finalizar la aplicación actual");

     selected.CloseMainWindow();
  }
}
```

Este pequeño ejemplo nos ha servido para mostrar la información que se puede obtener de un proceso en ejecución a través de la clase `Process`. Como vimos en los apartados anteriores, `Process` es también la clase utilizada para controlar la ejecución de los procesos del sistema. En el apartado siguiente, para completar nuestro recorrido por el manejo de procesos en la plataforma .NET, veremos una última aplicación de la clase `Process`: la redirección de los flujos de entrada y de salida estándar de un proceso.

Nota

En el cuadro de herramientas de Visual Studio .NET, dentro del apartado "Componentes", podemos encontrar un componente denominado `Process` que nos permite, si así lo deseamos, trabajar con objetos de la clase `System.Diagnostics.Process` de forma "visual", entre comillas porque tampoco se puede decir que se automatice demasiado el uso del componente `Process` al trabajar con él dentro del diseñador de formularios.

Operaciones de E/S

En ciertas ocasiones, nos puede interesar que la entrada de un proceso provenga directamente de la salida de otro proceso diferente. Como se mencionó al motivar el uso de hebras y procesos, enviar la salida de un proceso a un fichero y, después, leer ese fichero no siempre es la opción más eficiente. Los sistemas operativos suelen proporcionar un mecanismo mejor para conectar unos procesos con otros a través de sus canales estándares de entrada y salida de datos: los *"pipes"*.

Un programa estándar, tanto en Windows como en UNIX, posee por defecto tres canales de entrada/salida: la entrada estándar `StdIn` (asociada usualmente al teclado del ordenador), la salida estándar `StdOut` y un tercer canal, denominado `StdErr`, que también es de salida y se emplea, por convención, para mostrar mensajes de error.

Como no podía ser de otra forma, `System.Diagnostics.Process` nos permite redireccionar los tres canales de E/S estándar (`StdIn`, `StdOut` y `StdErr`). Sólo tenemos que volver a usar la clase auxiliar `ProcessStartInfo`. Fijando a `true` las propiedades `RedirectStandardInput`, `RedirectStandardOutput` y `RedirectStandardError`, podemos acceder a las propiedades `StandardInput`, `StandardOutput` y `StandardError` del proceso. Estas últimas son objetos de tipo `StreamReader` (la entrada estándar) o `StreamWriter` (las dos salidas), los mismos que se utilizan en la plataforma .NET para trabajar con ficheros.

Sólo tenemos que tener en cuenta un último detalle para poder redireccionar la entrada o la salida de un proceso en C#. Como ya mencionamos antes, `Process.Start()` utiliza por defecto la función `ShellExec` del API Win32. Esta función delega en el shell de Windows la decisión de qué hacer para acceder un fichero. En el caso de un fichero ejecutable, lo que hará será ejecutarlo. No obstante, cuando utilicemos redirecciones, la propiedad `ProcessStartInfo.UseShellExecute` debe estar puesta a `false` para crear directamente el proceso a partir del fichero ejecutable que hayamos indicado en la propiedad `FileName` y que las redirecciones se hagan efectivas.

Por ejemplo, podemos ejecutar comandos MS-DOS redireccionando la entrada y las salidas del proceso `cmd.exe`, el que se utiliza para acceder al "símbolo del sistema" en Windows. En concreto, el siguiente fragmento de código nos muestra un mensaje con un listado del contenido del directorio raíz de la unidad `C:\`:

```
Process proceso = new Process();

proceso.StartInfo.FileName = "cmd.exe";
proceso.StartInfo.UseShellExecute = false;
proceso.StartInfo.CreateNoWindow = true;
proceso.StartInfo.RedirectStandardInput = true;
proceso.StartInfo.RedirectStandardOutput = true;
proceso.StartInfo.RedirectStandardError = true;
```

```
process.Start()

StreamWriter sIn = proceso.StandardInput;
StreamReader sOut = proceso.StandardOutput;
StreamReader sErr = proceso.StandardError;

sIn.AutoFlush = true;
sIn.Write("dir c:\" + System.Environment.NewLine);
sIn.Write("exit" + System.Environment.NewLine);

string output = sOut.ReadToEnd();

sIn.Close();
sOut.Close();
sErr.Close();
proceso.Close();

MessageBox.Show(output);
```

Obviamente, el mecanismo anterior sólo nos permite controlar las operaciones de E/S de las aplicaciones en modo consola típicas de MS-DOS. Las aplicaciones Windows no funcionan de la misma forma, sino que consisten, básicamente, en un proceso que va recibiendo los eventos generados por el sistema operativo y va realizando las operaciones oportunas en función del evento que se haya producido. Los eventos pueden ser de muchos tipos, desde mensajes para mover, maximizar o minimizar la ventana de la aplicación, hasta los mensajes que se producen cuando se pulsa un botón, se hace click con el ratón o se selecciona un elemento de una lista.

Cada vez que se pulsa una tecla, también se produce un evento, que la aplicación se encargará de interpretar en función del contexto en el que se produzca. Desde nuestras aplicaciones también se pueden generar eventos de forma artificial, de modo que podemos simular la pulsación de teclas por parte del usuario. Esto nos permite conseguir un efecto similar al que se consigue al redireccionar la entrada estándar de una aplicación en modo consola. Mediante la clase `SendKeys`, del espacio de nombres `System.Windows.Forms`, podemos automatizar la pulsación de teclas. A continuación, se muestra un pequeño ejemplo en el que el Bloc de Notas funciona "solo":

```
Process process = new Process();

process.StartInfo.FileName    = "notepad.exe";
process.StartInfo.WindowStyle = ProcessWindowStyle.Normal;
process.EnableRaisingEvents   = true;

process.Start();
process.WaitForInputIdle(1000); // 1 segundo de timeout

if (process.Responding)
   System.Windows.Forms.SendKeys.SendWait
   ("Texto enviado con System.Windows.Forms.SendKeys");
```

No se puede utilizar `SendKeys` hasta que no se haya creado y se visualice la ventana principal de la aplicación, por lo es necesaria la llamada a `Process.WaitForInputIdle()`. Este método espera hasta que la aplicación pueda aceptar entradas por parte del usuario y su funcionamiento es análogo al de `Process.WaitForExit()`.

Con un pequeño retardo entre pulsación y pulsación, podría simularse a una persona tecleando algo. Este retardo lo podríamos conseguir con al método `Thread.Sleep()`, aunque el control de la ejecución de una hebra es algo que dejaremos para el siguiente apartado.

Acceso a recursos nativos de Windows

Con `SendKeys` se puede simular la pulsación de cualquier combinación de teclas. No obstante, `SendKeys` se limita a enviar eventos de pulsación de teclas a la aplicación que esté activa en cada momento. Y no existe ninguna función estándar en .NET que nos permita establecer la aplicación activa, pero sí en el API nativo de Windows.

El API de Windows incluye una gran cantidad de funciones (no métodos), del orden de miles. En el caso que nos ocupa, podemos recurrir a las funciones `FindWindow` y `SetForegroundWindow` del API Win32 para establecer la aplicación activa. Como estas funciones no forman parte de la biblioteca de clases .NET, tenemos que acceder a ellas usando los servicios de interoperabilidad con COM. Estos servicios son los que nos permiten acceder a recursos nativos del sistema operativo y se pueden encontrar en el espacio de nombres `System.Runtime.InteropServices`.

Para poder usar la función `SetForegroundWindow`, por ejemplo, tendremos que escribir algo parecido a lo siguiente:

```
using System.Runtime.InteropServices;
...
[DllImport("User32",EntryPoint="SetForegroundWindow")]
private static extern bool SetForegroundWindow(System.IntPtr hWnd);
```

Una vez hecho esto, ya podemos acceder a la función `SetForegroundWindow` como si de un método más se tratase:

```
Process proceso = ...

if (proceso.Responding) {
   SetForegroundWindow(proceso.MainWindowHandle);
   SendKeys.SendWait("Sorpresa!!!");
   SetForegroundWindow(this.Handle);
}
```

Y ya podemos hacer prácticamente cualquier cosa con una aplicación Windows, controlándola desde nuestros propios programas.

Hebras

Para dotar de cierto paralelismo a nuestras aplicaciones, no es necesario dividirlas en procesos completamente independientes. Dentro de un proceso, podemos tener varias hebras ejecutándose concurrentemente. Con las hebras se gana algo de eficiencia, al resultar más eficientes tanto los cambios de contexto como la comunicación entre hebras. No obstante, al compartir el mismo espacio de memoria, deberemos ser cuidadosos al acceder a recursos compartidos, para lo que tendremos que utilizar los mecanismos de sincronización que veremos más adelante.

En la plataforma .NET, las clases incluidas en el espacio de nombres `System.Threading`, junto con la palabra reservada `lock` en el lenguaje de programación C#, nos proporcionan toda la infraestructura necesaria para crear aplicaciones multihebra. De hecho, todas las aplicaciones .NET son en realidad aplicaciones multihebra. El recolector de basura no es más que una hebra que se ejecuta concurrentemente con la hebra principal de la aplicación y va liberando la memoria que ya no se usa.

La clase Thread

En este apartado veremos cómo se crean hebras y se trabaja con ellas en la plataforma .NET. En esta plataforma, una hebra se representa mediante un objeto de la clase `Thread`, que forma parte del espacio de nombres `System.Threading`. En este espacio de nombres también se encuentra el delegado sin parámetros `ThreadStart`, que se emplea para especificar el punto de entrada a la hebra (algo así como la función `main` de una hebra). Este delegado se le pasa como parámetro al constructor de la clase `Thread` para crear una hebra.

Para comenzar la ejecución de la hebra, se ha de utilizar el método `Start()` de la clase `Thread`. Este método inicia la ejecución concurrente del delegado que sirve de punto de entrada de la hebra. Como este delegado no tiene parámetros, el estado inicial de la hebra hay que establecerlo previamente. Para establecer su estado se pueden utilizar las propiedades y métodos del objeto en que se aloje el delegado. El siguiente fragmento de código muestra cómo se implementa todo este proceso en C#:

```
// Creación del objeto que representa a la hebra
Tarea tarea = new Tarea();
// Inicialización de los parámetros de la hebra
tarea.Param = 1234;
```

```
// Establecimiento del punto de entrada de la hebra (delegado)
ThreadStart threadStart = new ThreadStart(tarea.Ejecutar);
// Creación de la hebra
Thread thread = new Thread(threadStart);
// Ejecución de la hebra
thread.Start();
...
// Espera a la finalización de la hebra
thread.Join();
```

El fragmento de código anterior presupone que el objeto que encapsula la hebra, de tipo `Tarea` tiene una propiedad `Param` que se utiliza como parámetro en la ejecución de la hebra y un método sin parámetros `Ejecutar` que sirve de punto de entrada de la hebra.

Al final del fragmento de código anterior incluimos una llamada al método `Join()` de la hebra. Este método detiene la ejecución de la hebra actual hasta que finalice la ejecución de la hebra que se lanzó con la llamada a `Start()`. En la práctica, no obstante, `Join()` no se utiliza demasiado precisamente porque bloquea la ejecución de la hebra actual y anula las ventajas de tener varias hebras ejecutándose en paralelo. Generalmente, sólo usaremos `Join()` cuando queramos finalizar la ejecución de nuestra aplicación. En este caso, antes de terminar la ejecución de la hebra principal deberemos, de algún modo, finalizar la ejecución de las hebras que hemos lanzado desde la hebra principal de la aplicación.

Establecimiento de prioridades

Como se mencionó en la primera parte de este capítulo, en ocasiones resulta aconsejable asignarles distintas prioridades a las hebras de nuestra aplicación. Antes de llamar al método `Start()`, podemos establecer la prioridad de una hebra mediante la propiedad `Priority` de la clase `Thread`. Esta propiedad es del tipo definido por la enumeración `ThreadPriority` y puede tomar cinco valores diferentes:

Nivel de prioridad	Descripción
`Highest`	Prioridad máxima
`AboveNormal`	Prioridad superior a la habitual
`Normal`	Prioridad por defecto
`BelowNormal`	Prioridad inferior a la habitual
`Lowest`	Prioridad mínima

Los valores anteriores nos permiten indicarle al sistema operativo qué hebras han de gozar de acceso preferente a la CPU del sistema. Hemos de tener cuidado al utilizar esta propiedad porque afecta significativamente a la forma en que se efectúan las tareas concurrentemente. Si nuestra aplicación tiene una hebra principal que se encarga de recibir los eventos de la interfaz de usuario y otra auxiliar que hace algún tipo de cálculo, lo normal es que la hebra auxiliar tenga menos prioridad que la hebra principal, con el único objetivo de que nuestra aplicación responda de la manera más inmediata posible a las acciones que realice el usuario.

Prioridades de los procesos en Windows

También se pueden establecer prioridades para los distintos procesos de nuestra aplicación. En este caso, se debe recurrir a la propiedad `PriorityClass` de la clase `System.Diagnostics.Process`. Esta propiedad tiene que tomar uno de los valores definidos por la enumeración `ProcessPriorityClass`, que es distinta a la enumeración `ThreadPriority`:

Clase de prioridad	Prioridad base
`Idle`	4
`Normal`	8
`High`	13
`RealTime`	24

La prioridad de las hebras de un proceso será relativa a la clase de prioridad del proceso:

- Las hebras de un procesos de prioridad máxima, de clase `RealTime`, tiene preferencia incluso sobre algunos procesos importantes del sistema. Si un procesos con prioridad `RealTime` tarda algo de tiempo en realizar su tarea, el resto del sistema puede dejar de responder adecuadamente.

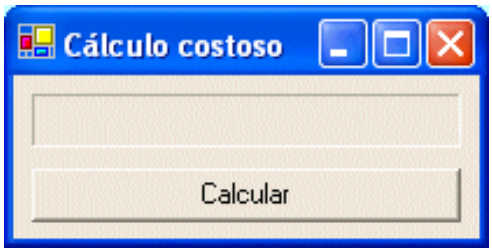
- En el polo opuesto, un proceso de prioridad `Idle` sólo accederá a la CPU cuando el sistema esté completamente inactivo, igual que el salvapantallas.

En la práctica, la asignación de prioridades a los procesos del sistema es algo más compleja porque el sistema operativo puede aumentar temporalmente la prioridad de un proceso bajo determinadas circunstancias. Por ejemplo, si ponemos a `true` la propiedad `PriorityBoostEnabled`, que por defecto es `false`, el sistema operativo le dará mayor prioridad al proceso cuando la ventana principal del mismo tenga el foco de Windows (esto es, esté en primer plano). En cualquier momento, no obstante, podemos conocer cuál era la prioridad inicial asignada a un proceso mediante la propiedad `BasePriority`, que es un número entero de sólo lectura.

Un pequeño ejemplo

Supongamos que nuestra aplicación, por el motivo que sea, ha de realizar una tarea costosa en términos de tiempo. La tarea en cuestión puede consistir en un complejo cálculo matemático, una larga simulación, la realización de una copia de seguridad, el acceso a un recurso a través de la red o cualquier otra cosa que requiera su tiempo. Pero a nosotros nos interesa que, mientras esa tarea se está efectuando, nuestra aplicación no deje de responder a las órdenes del usuario. Aquí es donde entra el juego el uso de hebras.

Si nuestro programa es una aplicación Windows, tendremos un formulario como el siguiente, en el cual aparece un botón `Button` y una barra de progreso `ProgressBar` que sirva para indicarle al usuario el avance en la realización de la tarea:



Siempre es buena idea proporcionarle al usuario pistas visuales en las que se ve cómo avanza la realización de la tarea. Sin embargo, el uso de otro tipo de pistas, como sonidos, no siempre es tan buena idea. Afortunadamente, el componente `System.Windows.Forms.ProgressBar` un mecanismo adecuado para mantener visible el estado de la aplicación. Hemos de tener en cuenta que la "visibilidad" del estado de la aplicación es un factor importante en la usabilidad de la misma. Además, esta propiedad no es exclusiva de las aplicaciones. En un proyecto de desarrollo de software, por ejemplo, mantener su estado visible es esencial para poder gestionarlo correctamente.

Volvamos ahora a nuestra aplicación. La tarea pendiente de realización, usualmente, se podrá descomponer en una serie de iteraciones que nos servirán para mantener la visibilidad de su estado. Esas iteraciones pueden consistir, por ejemplo, en realizar un paso en una simulación, transmitir un fragmento de los datos que se han de archivar o procesar un subconjunto de los registros con los que se ha de trabajar. Idealmente, la función que realice la tarea tendrá un parámetro proporcional al esfuerzo requerido:

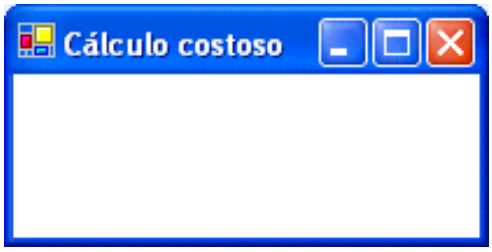
```
private void Calcular (int esfuerzo)
{
  int hecho = 0;

  MostrarProgreso (0, esfuerzo);
```

```
  while (hecho<esfuerzo) {
    ...
    hecho++;
    MostrarProgreso(hecho, esfuerzo);
  }
}
```

Los avances realizados se mostrarán periódicamente en la interfaz de usuario por medio de la barra de progreso, de forma que el usuario pueda hacerse una idea de cuánto tiempo falta para terminar la realización de la tarea:

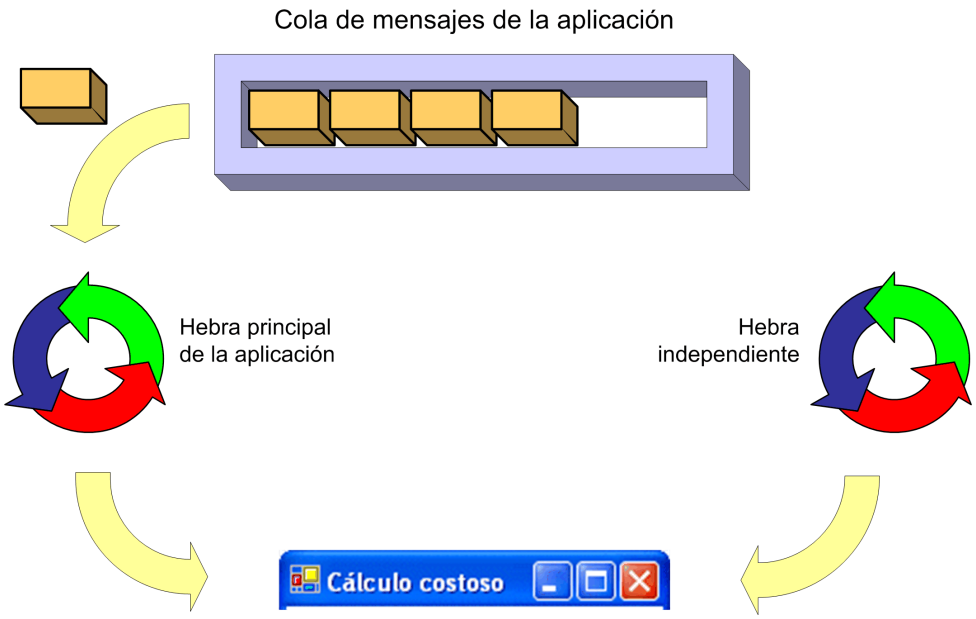
```
void MostrarProgreso (int actual, int total)
{
  progressBar.Maximum = total;
  progressBar.Value   = actual;
}
```

La forma ingenua de implementar nuestra aplicación sería hacer que, al pulsar el botón "Calcular", se invocase a la función `Calcular` que acabamos de definir. Sin embargo, esto lo único que conseguiría es que nuestra aplicación se quedase bloqueada mientras dure la operación. Al cambiar a otra aplicación y luego volver a la nuestra, nos podríamos encontrar una desagradable sorpresa:



Una imagen como esta es muy común, por otro lado, en demasiadas aplicaciones. La ventana "en blanco" se debe a que la aplicación Windows consiste, inicialmente, en una hebra que va recibiendo una secuencia de eventos del sistema operativo. Cuando se produce un evento, se llama al manejador correspondiente de forma síncrona. Hasta que no se vuelva de esa llamada, no se pasará a tratar el siguiente evento. Si la hebra principal de la aplicación se mantiene ocupada en realizar una tarea larga, como respuesta al evento que se produce al pulsar el botón "Calcular", esta hebra, como es lógico, no puede atender a ningún otro evento de los que se puedan producir. En el caso de arriba, el evento `Paint` del formulario, que solicita refrescar la imagen del formulario en pantalla, se quedará en cola esperando que termine la tarea asociada a la pulsación del botón, con lo que la interfaz de la aplicación se queda "congelada".

Cuando la respuesta a un evento se pueda realizar en un tiempo no apreciable para nosotros, menos de una décima de segundo aproximadamente, podemos llamar de forma síncrona al método que se encargue de realizar la acción oportuna. Cuando no, la solución a nuestro problema pasa inexorablemente, pues, por crear una hebra independiente que se ejecute en paralelo y no interfiera en el comportamiento habitual de la interfaz de nuestra aplicación.



Con la hebra auxiliar, la aplicación puede seguir procesando los mensajes que recibe provenientes del sistema operativo (o de otras aplicaciones).

Para crear una hebra, necesitaremos un delegado sin parámetros `ThreadStart`, que será el responsable de llamar al método `Calcular`. En el manejador del evento que lanza la hebra, se obtienen los parámetros que sean necesarios y se crear la hebra a partir del delegado `ThreadStart`, para el cual tenemos que crear un método auxiliar sin parámetros:

```
private void buttonCalc_Click(object sender, System.EventArgs e)
{
  ThreadStart start  = new ThreadStart(CalcularNoParams);
  Thread      thread = new Thread(start);

  thread.Start();
}

private void CalcularNoParams () {
  Calcular(1000); // ... o lo que resulte adecuado
}
```

Con esta pequeña modificación de la aplicación inicial, hemos conseguido que nuestra aplicación se convierta en una aplicación multihebra y su interfaz gráfica funcione correctamente. Al menos, aparentemente.

Usando una hebra independiente, nuestra aplicación se comporta como cabría esperar de ella.

Ejecución asíncrona de delegados

En el ejemplo con el que cerramos el apartado anterior, tuvimos que implementar un método auxiliar sin parámetros únicamente para poder lanzar la hebra. C# ofrece un mecanismo que suele ser más cómodo para obtener el mismo resultado: crear un delegado y ejecutarlo de forma asíncrona.

Un delegado, recordemos, viene a ser algo parecido a un puntero a función en lenguajes como C. Para sustituir nuestro método auxiliar por un delegado, lo primero que tenemos que hacer es declararlo. Antes, en el método auxiliar inicializábamos los parámetros con los que se iba a ejecutar la hebra. Ahora, podemos declarar directamente un delegado con parámetros usando la palabra reservada `delegate`:

```
delegate void SimulationDelegate (int steps);
```

Una vez declarado el delegado como tipo, podemos crear una instancia de ese tipo a partir del método `Calcular`, el que se encarga de realizar la tarea para la cual vamos a crear una hebra auxiliar:

```
SimulationDelegate delegado = new SimulationDelegate(Calcular);
```

El delegado, como tal, lo podemos usar directamente para invocar al método que encapsula (igual que se invocan los manejadores de eventos usados en los formularios, los cuales también se registran como delegados para responder a los eventos correspondientes). Por ejemplo, dado el delegado anterior, podríamos llamar a la función `Calcular` a través del delegado:

```
  delegado (1000);
```

Esto, obviamente, no soluciona nuestro problema inicial, ya que la invocación al método `Calcular` se sigue realizando de forma síncrona.

No obstante, los delegados declarados con `delegate`, heredan en realidad de la clase `System.MulticastDelegate`. En la plataforma .NET, la clase `MulticastDelegate` implementa tres métodos:

```
class SimulationDelegate : MulticastDelegate
{
  public void Invoke(int steps);
  public void BeginInvoke
              (int steps, AsyncCallback callback, object state);
  public void EndInvoke(IAsyncResult result);
}
```

Cuando un delegado se usa directamente, como hicimos antes, en realidad se está llamando al método síncrono `Invoke`, que es el que se encarga de llamar a la función concreta con la que se instancia el delegado (`Calcular`, en este caso).

Los otros dos métodos del delegado, `BeginInvoke` y `EndInvoke`, son los que permiten invocar al delegado de forma asíncrona. Llamarlo de forma asíncrona equivale a crear una hebra auxiliar y comenzar la ejecución del delegado en esa hebra auxiliar. Por tanto, podemos realizar tareas en hebras independientes con sólo crear un delegado y llamar a su método `BeginInvoke`:

```
delegate void SimulationDelegate (int steps);

private void buttonCalc_Click(object sender, System.EventArgs e)
{
  SimulationDelegate delegado = new SimulationDelegate(Calcular);

  delegado.BeginInvoke(1000, null, null);
}
```

En principio, todo parece funcionar correctamente, aunque aún nos quedan algunos detalles por pulir, como veremos a continuación.

Hebras en aplicaciones Windows

En el ejemplo anterior, y gracias posiblemente a la implementación de nuestro sistema operativo, todo funcionó correctamente, al menos en apariencia. Sin embargo, violamos un principio básico de la programación con hebras. Accedimos de forma concurrente a un recurso compartido desde dos hebras diferentes sin coordinar el acceso a ese recurso compartido. En este caso, el recurso compartido era el formulario y sus controles. Como veremos cuando de describir cómo se trabaja con hebras en C#, existen mecanismos de sincronización que sirven de protección para acceder a recursos compartidos desde distintas hebras o procesos.

En el caso de las aplicaciones Windows, cuando el recurso compartido es una ventana, por convención se sigue la siguiente regla: una ventana se manipula únicamente desde la hebra que la creó. Esta hebra es la que se encarga de ir recibiendo eventos e ir respondiendo a esos eventos de la forma adecuada. De esta forma, el acceso a la ventana se serializa y evitamos tener que utilizar mecanismos de sincronización más complejos.

> Todos los controles de Windows, que derivan de la clase base `System.Windows.Forms.Control`, incluyen una propiedad llamada `InvokeRequired` que nos indica si estamos accediendo al control desde una hebra distinta a aquélla en la que se creó. Es decir, para manipular de forma segura las propiedades de un control, se debe verificar la siguiente aserción:
>
> ```
> using System.Diagnostics;
> ...
> Debug.Assert(control.InvokeRequired == false);
> ```

En este sentido, la documentación lo deja bastante claro: Sólo hay cuatro métodos de un control que se pueden llamar de forma segura desde cualquier hebra. Estos métodos son `Invoke`, `BeginInvoke`, `EndInvoke` y `CreateGraphics`. Cualquier otro método ha de llamarse a través de uno de los anteriores.

En la práctica, para modificar las propiedades de un control desde una hebra distinta a aquélla que lo creó, lo único que tenemos que hacer es volver a utilizar delegados. En el ejemplo de antes, la hebra auxiliar delegará en la hebra principal de la aplicación la tarea de actualizar el estado de la barra de progreso. Para ello, lo único que tenemos que hacer es aprovechar el método `Invoke` del control, que sirve para llamar de forma síncrona a un delegado desde la

hebra propietaria del control:

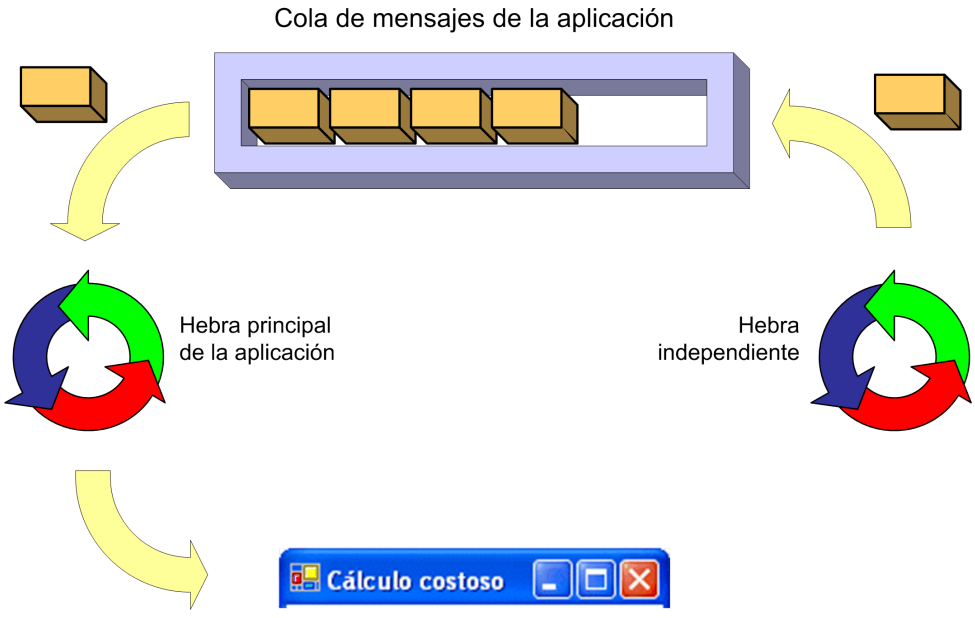
En primer lugar, declaramos un delegado para la función que hasta ahora accedía al formulario desde la hebra auxiliar, la función `MostrarProgreso`:

```
delegate void ProgressDelegate (int actual, int total);
```

A continuación, sustituimos las llamadas a `MostrarProgreso` por el siguiente fragmento de código:

```
ProgressDelegate progress = new ProgressDelegate(MostrarProgreso);
...
this.Invoke (progress, new object[]{actual, total} );
```

El uso de `Invoke` nos garantiza que accedemos de forma segura a los controles de nuestra ventana en una aplicación multihebra. La hebra principal crea una hebra encargada de realizar una tarea computacionalmente costosa, la hebra auxiliar, y ésta le pasa el control a la hebra principal cada vez que necesita actuar sobre los controles de la interfaz, de la que es responsable la hebra principal de la aplicación.



Delegando en la hebra principal a la hora de actualizar la interfaz, se evita el problema del acceso concurrente a un recurso compartido desde dos hebras diferentes

Tener que llamar al método `Invoke` del control cada vez que queremos garantizar el correcto funcionamiento de nuestra aplicación multihebra resulta bastante incómodo y, además, es bastante fácil que se nos olvide hacerlo en alguna ocasión. Por tanto, no es mala idea que sea la propia función a la que llamamos la que se encargue de asegurar su correcta ejecución en una aplicación multihebra, llamando ella misma al método `Invoke`.

En el ejemplo anterior, el método `MostrarProgreso` lo podemos implementar de la siguiente forma:

```
void MostrarProgreso (int actual, int total)
{
  if ( progress.InvokeRequired == false ) {
     // Hebra correcta
     progress.Maximum = total;
     progress.Value   = actual;
  } else {
    // Llamar a la función desde la hebra adecuada
    ProgressDelegate progress = new ProgressDelegate(MostrarProgreso);
    this.Invoke (progress, new object[] { actual, total} );
  }
}
```

De esta forma, podremos llamar al método `MostrarProgreso` de la forma habitual sin tener que preocuparnos del hecho de encontrarnos en una aplicación multihebra.

Con la llamada síncrona a `Invoke`, la hebra auxiliar queda detenida mientras la hebra principal llama al delegado encargado de actualizar la interfaz de usuario, igual que antes la hebra principal quedaba bloqueada cuando no usábamos una hebra auxiliar. Para evitar este bloqueo temporal, podemos recurrir al método `BeginInvoke`, que hace lo mismo que `Invoke` pero de forma asíncrona.+

En el ejemplo que venimos utilizando, como el método `MostrarProgreso` no devuelve ningún valor, podemos sustituir directamente la llamada síncrona con `Invoke` por una llamada asíncrona con `BeginInvoke`, sin más complicaciones:

```
BeginInvoke (progress, new object[] {actual, total} );
```

Si el método devolviese algún valor, entonces habría que recurrir al uso de `EndInvoke`, si bien este no es el caso porque lo único que hace el método es actualizar la interfaz de usuario desde el lugar adecuado para ello: la hebra principal de la aplicación.

Siempre que podamos, el uso de `BeginInvoke` es preferible al de `Invoke` porque el método `BeginInvoke` evita que se puedan producir bloqueos. Al final de este capítulo veremos otro ejemplo del uso de llamadas asíncronas en la plataforma .NET, si bien antes nos

falta por ver cómo se puede controlar la ejecución de las hebras y cómo se emplean mecanismos de sincronización para solucionar el problema del acceso a recursos compartidos en situaciones más generales.

Control de la ejecución de una hebra

Como dijimos hace ya algunas páginas, es aconsejable que el usuario mantenga siempre el control de la aplicación (un principio básico del diseño de interfaces de usuario). Cuando se ha de realizar una tarea larga, se utilizan hebras para que la aplicación siga respondiendo a las órdenes del usuario. Y una de esas órdenes puede ser, precisamente, la de cancelar la ejecución de la tarea para la cual creamos una hebra auxiliar. En otras palabras, una vez que el usuario ordena la ejecución de una tarea computacionalmente costosa, sería deseable que también pudiese rectificar y anular la orden dada.

Para que el usuario mantenga el control absoluto sobre la ejecución de las distintas hebras de nuestra aplicación, hemos de incluir en la interfaz de usuario algún mecanismo que le permita al usuario cancelar la ejecución de la hebra auxiliar. El mismo botón que se utilizó para lanzar la hebra puede servirnos para cancelar la operación, si bien también se podría crear una ventana de diálogo independiente con la misma funcionalidad, al estilo del Explorador de Windows.

Aparte de lo anterior, para que cuando el usuario decida cancelar la operación en curso, la detención de la operación se haga efectiva, la hebra que controla la interfaz de usuario debe comunicarle a la hebra auxiliar que detenga su ejecución. Y esta última debe, a su vez, comprobar periódicamente si el usuario desea terminar la ejecución de la tarea que tiene asignada la hebra auxiliar. Para faciliar la comunicación entre las hebras, podemos definir una variable de estado:

```
enum Estado { Activo, Inactivo, Cancelando };
Estado estado = Estado.Inactivo;
```

La hebra estará en estado `Inactivo` cuando no esté ejecutándose, `Activo` cuando esté ejecutándose y `Cancelando` cuando esté ejecutándose y el usuario haya solicitado la finalización de su ejecución.

> Un programador con experiencia en el desarrollo de aplicaciones multihebra con herramientas de Borland como Delphi o C++Builder se dará cuenta de que lo que vamos a hacer no es más que implementar el mecanismo que suministra la clase `TThread` con el método `Terminate()` y la propiedad `Terminated`.

Si optamos por controlar la ejecución de la hebra desde un único botón de nuestra interfaz de usuario, este botón variará su comportamiento en función del estado actual de la hebra.

```
private void buttonCalc_Click(object sender, System.EventArgs e)
{
  SimulationDelegate delegado;

  switch (estado) {

    case Estado.Inactivo:             // Comenzar el cálculo
         estado = Estado.Activo
         buttonCalc.Text = "Cancelar";
         delegado = new SimulationDelegate(Calcular);
         delegado.BeginInvoke(1000, null, null);
         break;

    case Estado.Activo:              // Cancelar operación
         estado = Estado.Cancelando;
         buttonCalc.Enabled = false;
         break;

    case Estado.Cancelando:          // No debería suceder nunca...
         Debug.Assert(false);
         break;
  }
}
```

Obsérvese que, mientras la hebra no detenga su ejecución, tenemos que deshabilitar el botón con el fin de evitar que, por error, se cree una nueva hebra en el intervalo de tiempo que pasa desde que el usuario pulsa el botón "Cancelar" hasta que la hebra realmente se detiene.

Para que la hebra detenga su ejecución, ésta ha de comprobar periódicamente que su estado es distinto de `Cancelando`, el estado al que llega cuando el usuario le indica que quiere finalizar su ejecución. En este caso, el estado de la hebra es una variable compartida entre las dos hebras. En general, deberemos tener especial cuidado con el acceso a recursos compartidos y sincronizar el acceso a esos recursos utilizando mecanismos de sincronización como los que veremos al final de este capítulo. En este caso, no obstante, no resulta necesario porque sólo usamos ese recurso compartido como indicador. Una hebra establece su valor y la otra se limita a sondear periódicamente dicho valor. La hebra principal establece el estado de la hebra auxiliar y la hebra auxiliar simplemente comprueba su estado para decidir si ha de finalizar su ejecución o no.

```
delegate void EndProgressDelegate ();

private void CalcularPi (int precision)
{
  EndProgressDelegate endProgress =
            new EndProgressDelegate(EndProgress);
  ...
```

```
      try {

        while ( (estado!=Estado.Cancelando) && ...) {
              ...
        }

      } finally {
        this.Invoke(endProgress);
      }
    }

    private void EndProgress ()
    {
      estado = Estado.Inactivo;
      buttonCalc.Text = "Calcular";
      buttonCalc.Enabled = true;
    }
```

Aparte de ir mostrando periódicamente el progreso de su ejecución, algo que deberíamos seguir haciendo tal como indicamos en el apartado anterior, la hebra auxiliar ahora también es responsable de volver a habilitar el botón `buttonCalc` cuando termina su ejecución.

¡Ojo!

Cuando varias hebras acceden simultáneamente a algún recurso compartido es necesario implementar algún tipo de mecanismo de exclusión mutua. En el desarrollo de aplicaciones multihebra siempre debemos garantizar la exclusión mutua en el acceso a recursos compartidos. En el caso anterior, la aplicación funciona porque sólo una de las hebras modifica el valor de la variable compartida y, además, lo restablece sólo después de que la otra hebra haya comprobado el valor de la variable compartida y haya decidido finalizar su ejecución (cuando se llama al delegado `EndProgress`).

En situaciones como ésta, en las que prescindimos de mecanismos de sincronización, deberemos estudiar las posibles condiciones de carrera que se pueden producir. Si usamos mecanismos de sincronización, entonces tendremos que estudiar la posibilidad de que se produzcan bloqueos. Estos problemas los analizaremos con más detenimiento en este mismo capítulo, si bien antes nos detendremos un momento en ver otras formas de controlar la ejecución de las hebras.

Interrupción de la ejecución de una hebra

En ocasiones, puede que nos interese controlar la ejecución de una hebra desde fuera. En tal situación, disponemos básicamente de dos opciones:

- Detener temporalmente la ejecución de la hebra para luego proseguir su ejecución desde el punto en el que se hubiese detenido.

- Terminar la ejecución de la hebra definitivamente, posiblemente porque no responda "voluntariamente" a las peticiones del usuario que está intentando finalizar su ejecución.

A su vez, para cada una de esas opciones tenemos dos mecanismos diferentes para lograr nuestros objetivos, como veremos en los párrafos siguientes.

Detención temporal de la ejecución de una hebra

Si lo único que queremos es detener momentáneamente la ejecución de una hebra de nuestra aplicación para después dejarla proseguir su camino, podemos emplear:

- El método `Sleep()`, que detiene la hebra actual durante un número determinado de milisegundos.

- El método `Suspend()`, que detiene de forma indefinida la ejecución de la hebra (hasta el momento en que se llame al método `Resume()`).

Tanto uno como otro nos sirven para detener temporalmente una hebra y hacerlo de forma que no consuma tiempo de CPU, quedando ésta libre para poder ejecutar otras hebras. `Sleep()` se suele utilizar con frecuencia, por ejemplo, cuando se quiere introducir un retardo artificial entre dos operaciones, como sucede cuando queremos "animar" una parte de nuestra aplicación. El ojo humano sólo detecta cambios si éstos se mantienen durante decenas de milisegundos, por lo que, si queremos que el usuario pueda seguir visualmente la ejecución de una tarea, tendremos que introducir pequeños retardos entre las operaciones y hacer que éstas se ejecuten a nuestra velocidad, no a la de la máquina.

Finalización obligatoria de la ejecución de una hebra

Otra situación en la que estaríamos interesados en controlar la ejecución de una hebra es cuando la hebra "no nos hace caso". En este caso, que procuraremos evitar en todo momento, la solución suele pasar por llamar al método `Abort()`. Esta solución es drástica, pues interrumpe inmediatamente la ejecución de la hebra, esté donde esté, y genera una excepción de tipo `ThreadAbortException`.

El problema de `Abort()` es que la interrupción brusca de la ejecución de la hebra puede dejar en estado inconsistente los datos con los que la hebra estuviese trabajando. Afortunadamente, existe una solución menos radical que consiste en utilizar el método `Interrupt()`.

El método `Interrupt()` también sirve para finalizar la ejecución de una hebra, pero dicha finalización, que involucra la generación de una excepción de tipo `ThreadInterruptedException`, sólo se realizará en puntos predeterminados. De esta forma, se asegura que los datos con los que trabaja la hebra verificarán sus invariantes; es decir, siempre tendrán valores válidos. La excepción `ThreadInterruptedException`, junto con la consiguiente terminación de la ejecución de la hebra, sólo se producirá cuando la hebra esté detenida esperando algo. Esto incluye cuando la hebra está detenida temporalmente en una llamada a `Sleep()`, cuando la hebra está detenida esperando la finalización de otra hebra en una llamada a `Join()` y cuando la hebra está esperando que se libere un recurso compartido bloqueado por otra hebra. Esta última situación se produce cuando la hebra llama a `Monitor.Wait()`, un mecanismo de sincronización que estudiaremos en el siguiente apartado de este capítulo.

> Si al llamar a `Interrupt()`, la hebra no está detenida, la llamada queda registrada y la excepción se producirá automáticamente la próxima vez que se detenga la hebra. Si queremos asegurarnos de que la hebra terminará su ejecución rápidamente, podemos introducir llamadas artificiales del tipo `Thread.Sleep(0)` para darle a la hebra la oportunidad de terminar su ejecución.

Mecanismos de sincronización

El código ejecutado por una hebra debe tener en cuenta la posible existencia de otras hebras que se ejecuten concurrentemente, especialmente si esa hebra comparte determinados recursos con otras hebras. Dado que las hebras se ejecutan en el mismo espacio de memoria, dentro del proceso del que son "subprocesos", hay que tener especial cuidado para evitar que dos hebras accedan a un recurso compartido al mismo tiempo. Este recurso compartido, usualmente, será un objeto al que las distintas hebras acceden través de un miembro estático de alguna clase (el equivalente a las variables globales en orientación a objetos) o de algún otro punto de acceso común.

Si bien, idealmente, podría pensarse que las hebras deberían ser completamente independientes, en la práctica esto no es así. La ejecución de una hebra puede depender del resultado de las tareas que realicen otras hebras. De hecho, lo normal es que tenga que existir un mínimo de cooperación en la ejecución de las hebras y esa cooperación se realizará a través de recursos compartidos. En otras palabras, las distintas hebras de una aplicación han de coordinar su ejecución y los mecanismos de sincronización son necesarios para ello.

Nota

El modelo de sincronización utilizado en la plataforma .NET es completamente análogo al utilizado por el lenguaje de programación Java. Afortunadamente para el programador, que ya no tiene que hacer malabarismos con los distintos modelos de ejecución del modelo COM usado en Windows (salvo que tenga que utilizar los mecanismos de interoperabilidad con COM, claro está).

Acceso a recursos compartidos

Cuando varias hebras comparten el uso de un recurso, pueden ocurrir situaciones no deseadas si dos o más hebras acceden (o intentan acceder) al mismo recurso simultáneamente. Pensemos, por ejemplo, en lo que sucedería si una impresora en red no tuviese una cola de impresión. Lo impreso en papel podría ser desastroso si se mezclasen fragmentos de los documentos enviados por distintos usuarios. Lo mismo puede suceder cuando el recurso compartido es simplemente un fragmento de memoria en el que se almacenan datos con los que trabaja nuestra aplicación.

Más formalmente, los objetos compartidos por varias hebras han de verificar determinadas

propiedades, sus invariantes. Esos invariantes se mantienen cuando lo único que hacemos es consultar el estado del objeto (una simple operación de lectura), pero puede que no se verifiquen siempre cuando llamamos a un método que actualiza el estado del objeto (el equivalente orientado a objetos de una operación de escritura). En este último caso, se hace necesario que la hebra que actualiza el estado del objeto bloquee el acceso a ese objeto, al menos mientras dure la operación de actualización.

Cuando se bloquea el acceso al objeto, sólo una hebra puede manipularlo y así nos aseguramos de que ninguna otra hebra obtiene algo inconsistente al consultar el estado del objeto. En otras palabras, las aplicaciones multihebra evitan el acceso concurrente a recursos compartidos utilizando mecanismos de exclusión mutua. En realidad, sólo resulta necesario garantizar la exclusión mutua a aquellos objetos que pueden cambiar de estado.

Si no se bloquea el acceso a un objeto mientras se éste se modifica, el resultado de la ejecución de un programa dependerá de la forma en que se entrelacen las operaciones de las distintas hebras. Y, como ya sabemos, eso es algo que no se puede predecir y que dependerá de la situación, por lo que este tipo de errores son difíciles de detectar y de reproducir.

El mecanismo más común para garantizar la exclusión mutua consiste en el uso de **secciones críticas**. Una sección crítica delimita un fragmento de código al cual sólo puede acceder una hebra en cada momento. En el caso del lenguaje de programación C#, las secciones críticas las podemos implementar con monitores o con cerrojos, que son los dos mecanismos de exclusión mutua que veremos en los próximos dos apartados.

> Por un lado, hay que bloquear el acceso concurrente a objetos compartidos. Por otro lado, hay que tener cuidado de no bloquear innecesariamente la ejecución de otras hebras, con el objetivo de que la aplicación sea realmente concurrente y no simplemente secuencial.

Monitores

Como se acaba de mencionar, la exclusión mutua puede conseguirse utilizando distintos mecanismos que permiten definir secciones críticas, regiones de código a la que sólo puede encontrarse una hebra en cada momento, garantizando así la exclusión mutua en el acceso a recursos compartidos. En la plataforma .NET, la clase `System.Threading.Monitor` se utiliza para implementar uno de esos mecanismos: los monitores.

Los monitores proporcionan un modelo de coordinación similar a las secciones críticas que se pueden encontrar en el propio API del sistema operativo Windows. De hecho, su uso en .NET es muy similar al uso de secciones críticas en Win32:

Secciones críticas en C#

```
// Entrada en la sección crítica

Monitor.Enter(this);

try {

  // Acciones que requiran un acceso coordinado
  ...

} finally {

  // Salida de la sección crítica
  Monitor.Exit(this);
}
```

En este caso, hemos creado una sección crítica asociada al objeto `this`, de forma que, mientras una hebra esté ejecutando el fragmento de código incluido entre `Monitor.Enter(this)` y `Monitor.Exit(this)`, ninguna otra hebra podrá acceder a la sección crítica asociada a `this`. En el caso de intentar hacerlo, quedará detenida en `Monitor.Enter(this)` hasta que la hebra que esté ejecutando la sección crítica salga de ella. Obviamente, otras hebras podrán siempre acceder a secciones críticas que estén asociadas a otros objetos (distintos de `this`) mientras las secciones críticas a las que intenten acceder estén libres.

Obsérvese también cómo hemos envuelto la sección crítica en un bloque `try..finally` para garantizar que se sale de la sección crítica aun cuando se produzca una excepción mientras se ejecutan las sentencias incluidas en las sección crítica. Si no hiciésemos y saltase una excepción, ninguna otra hebra podría acceder a la sección crítica y, posiblemente, esto acabaría ocasionando un bloqueo de nuestra aplicación.

Aparte de poder definir secciones críticas, resulta conveniente disponer de algún mecanismo que permita bloquear la ejecución de una hebra hasta que se cumpla una condición. En algunas plataformas, este mecanismo de coordinación lo proporcionan las "variables condición". En otras, se dispone de semáforos. En la plataforma .NET, la clase `System.Threading.Monitor` incluye los métodos estáticos `Wait`, `Pulse` y `PulseAll` que se utilizan en la práctica con la misma finalidad que los semáforos o las variables condición.

Cuando una hebra llama al método `Monitor.Wait(obj)`, queda a la espera de que otra hebra invoque al método `Monitor.Pulse(obj)` sobre el mismo objeto `obj` para proseguir su ejecución. Cuando la hebra llama a `Monitor.Wait(obj)`, la hebra ha de tener acceso exclusivo al objeto `obj`. Esto es, la llamada a `Monitor.Wait(obj)` estará dentro de una sección crítica definida sobre el objeto `obj`. Si no, se produciría una excepción de tipo `SynchronizationLockException`.

En el momento en que la hebra llama a `Monitor.Wait(obj)`, la hebra desbloquea el acceso al objeto `obj` de forma que otras hebras puedan acceder a dicho objeto, a la vez que se queda esperando al objeto hasta que otra hebra llame a `Monitor.Pulse(obj)` o `Monitor.PulseAll(obj)` (o la ejecución de la hebra se interrumpa bruscamente con una llamada a `Interrupt()`).

La llamada a `Monitor.Pulse(obj)` no hace nada a no ser que haya una hebra esperando al objeto `obj`. En ese caso, la hebra recibe una notificación del cambio de estado y puede reanudar su ejecución en cuanto recupere el acceso en exclusiva al objeto `obj`. `Monitor.PulseAll(obj)`, como puede suponerse, notifica el cambio de estado a todas las hebras que estuviesen esperando al objeto `obj`.

Ahora bien, tras llamar a `Monitor.Pulse(obj)`, no se garantiza que la hebra que sale de su espera haya sido la siguiente en conseguir el acceso exclusivo al objeto `obj`, justo después de la llamada a `Monitor.Pulse(obj)`. Por tanto, habrá que volver a comprobar si se cumple o no la condición que condujo inicialmente a la situación de espera. Es decir, siempre que usemos `Monitor.Wait()`, lo haremos de la siguiente forma:

```
while (!condición)
      Monitor.Wait(obj);
```

Aunque el bucle `while` pueda parecer redundante, es completamente necesario para garantizar la corrección del programa en el uso de `Monitor.Wait()`

> Como sucede con casi todos los mecanismos de coordinación y comunicación que involucran una espera indefinida, existen variantes de `Monitor.Wait()` que incluyen un parámetro adicional que indica el tiempo máximo de espera. Si la espera no termina antes de el plazo determinado por ese *time-out*, entonces `Monitor.Wait()` devuelve `false` y la hebra reanuda su ejecución (en cuanto consiga acceso exclusivo al objeto correspondiente).

Cerrojos: La sentencia lock

A diferencia de lenguajes como C++, que no incorpora ningún mecanismo en la sintaxis del lenguaje para dotar de paralelismo a nuestros programas, lenguajes modernos como C# o Java incluyen palabras reservadas cuya misión consiste en facilitar el acceso a recursos compartidos en aplicaciones multihebra.

C++ no incluye ninguna palabra reservada con este fin no porque el desarrollo de aplicaciones multihebra sea algo reciente, sino porque el creador del lenguaje, Bjarne Stroustroup, defiende el uso de bibliotecas como medio para ampliar la capacidad de un lenguaje de programación. En el caso de C o C++, se suele utilizar la biblioteca de hebras POSIX, que forma parte de un estándar (Portable Operating System Interface, IEEE Std. 1003.1-2001).

En el caso del lenguaje C#, la sentencia `lock` nos permite implementar una sección crítica completamente equivalente al uso de la clase `Monitor`. En lenguajes como C# o Java, cada objeto implementa un cerrojo, que se puede usar como mecanismo de exclusión mutua. La sentencia `lock` tiene como parámetro un objeto, de tal forma que `lock(obj)` mantiene bloqueado el acceso al objeto `obj` mientras dura la ejecución de las sentencias incluidas dentro del ámbito de la sentencia `lock`. Su sintaxis, en realidad, es muy sencilla:

La sentencia lock

```
lock (this) {

  // Acciones que requiran un acceso coordinado
  // ...
}
```

El uso de `lock(obj)` es completamente equivalente al uso de `Monitor.Enter(obj)` y `Monitor.Exit(obj)`. Sólo una de las hebras tendrá acceso a las regiones de código delimitadas por `lock(obj)` (o `Monitor.Enter(obj)` y `Monitor.Exit(obj)`, en su caso). La ventaja que tiene el uso de `lock(obj)` es que nunca se nos olvida la llamada a `Monitor.Exit(obj)`. Las llaves que delimitan el ámbito de la sentencia `lock(obj)` sirven automáticamente para delimitar la sección crítica.

Llegado este punto, alguien puede pensar ¿para qué sirve entonces la clase `Monitor`? En realidad, la sentencia `lock` no es más que "azúcar sintáctica" que se añade al lenguaje para facilitar su uso. Además, la clase `Monitor` incorpora funciones que no están disponibles cuando se usa `lock`, como puede ser esperar con `Monitor.Wait()` o tantear a ver si se puede acceder de forma exclusiva a un objeto con `Monitor.TryEnter()` sin que la hebra se bloquee. En la práctica, se puede combinar el uso de la sentencia `lock` con las llamadas a los métodos estáticos definidos en la clase `Monitor`, tal como muestra el siguiente ejemplo.

Lectores y escritores

El método `Monitor.PulseAll(obj)`, que puede parecer algo inútil a primera vista, resulta bastante práctico cuando se quiere permitir el acceso concurrente a un recurso compartido para leer pero se quiere garantizar el acceso exclusivo al recurso cuando se va a modificar. La lectura concurrente de los datos mejora el rendimiento de una aplicación multihebra y sin afectar a la corrección del programa. A continuación se muestra una posible implementación de un mecanismo que permita concurrencia entre lectores pero exija exclusión mutua a los escritores:

```
class RWLock
{
  int lectores = 0;

  public void AcquireExclusive()
  {
    lock (this) {
         while (lectores!=0) Monitor.Wait(this);
         lectores--;
    }
  }

  public void AcquireShared()
  {
    lock (this) {
         while (lectores<0) Monitor.Wait(this);
         lectores++;
    }
  }

  public void ReleaseExclusive()
  {
    lock (this) {
         lectores = 0;
         Monitor.PulseAll(this);
    }
  }

  public void ReleaseShared()
  {
    lock (this) {
         lectores--;
         if (lectores==0) Monitor.Pulse(this);
    }
  }
}
```

En el artículo de Birrell citado al final de este capítulo se puede encontrar más información acerca de este tipo de mecanismos de sincronización, incluyendo una discusión detallada de los problemas que pueden aparecer con una implementación como la mostrada.

NOTA: En la biblioteca estándar de la plataforma .NET, la clase `ReaderWriterLock`, incluida en el espacio de nombres `System.Threading`, proporciona la funcionalidad del cerrojo que acabamos de implementar.

La única regla que hemos de tener en cuenta para utilizar mecanismos de exclusión mutua es la siguiente: en una aplicación multihebra, el acceso a objetos compartidos cuyo estado puede variar ha de hacerse siempre de forma protegida. La protección la puede proporcionar directamente el cerrojo asociado al objeto cuyas variables de instancia se modifican. El acceso a los datos siempre se debe hacer desde la hebra que tenga el cerrojo del objeto para garantizar la exclusión mutua. El siguiente ejemplo muestra cómo se debe hacer:

```
public class Account
{
  decimal balance;

  public void Deposit(decimal amount)
  {
    lock (this) {
      balance += amount;
    }
  }

  public void Withdraw(decimal amount)
  {
    lock (this) {
      balance -= amount;
    }
  }
}
```

Obviamente, el lenguaje no pone ninguna restricción sobre el objeto que se utiliza como parámetro de la sentencia `lock`. De todas formas, para simplificar el mantenimiento de la aplicación se procura elegir el que resulte más obvio. Si se modifican variables de instancia de un objeto, se usa el cerrojo de ese objeto. Si se modifican variables estáticas de una clase, entonces se utiliza el cerrojo asociado al objeto que representa la clase (`lock(typeof(Account))`, por poner un ejemplo). De esta forma, se emplea siempre el cerrojo asociado al objeto cuyos datos privados se manipulan. Además, se evitan de esta forma situaciones como la siguiente:

```
lock (this) {
 i++;
}
...
lock (otro) {
 i--;
}
```

En este ejemplo, la variable `i` se manipula en dos secciones críticas diferentes, si bien dichas secciones no garantizan la exclusión mutua en el acceso a `i` porque están definidas utilizando los cerrojos de dos objetos diferentes. En programación concurrente, algo de disciplina y de rigor siempre resulta aconsejable, si no necesario.

Aunque el compilador de C# no lo hace, existen herramientas de análisis que permiten comprobar los cerrojos que se han adquirido para acceder a cada variable e incluso detectan si se utilizan conjuntos inconsistentes de cerrojos para acceder a variables concretas (como sucede en el ejemplo que acabamos de ver).

Obsérvese también que no hay que abusar del uso de secciones críticas. Cada cerrojo que se usa impide potencialmente el progreso de la ejecución de otras hebras de la aplicación. Las secciones críticas eliminan el paralelismo y, con él, muchas de las ventajas que nos condujeron a utilizar hebras en primer lugar. La aplicación puede carecer por completo de paralelismo y convertirse en una aplicación secuencial pese a que utilicemos hebras, con la consiguiente degradación del rendimiento de la aplicación.

Pero la ausencia de paralelismo no es, ni mucho menos, el peor problema que se nos puede presentar durante el desarrollo de una aplicación multihebra. El uso de mecanismos de sincronización como cerrojos o monitores puede conducir a otras situaciones poco deseables que debemos tener en cuenta:

- **Interbloqueos** [*deadlocks*]: Una hebra se bloquea cuando intenta bloquear el acceso a un objeto que ya está bloqueado. Si dicho objeto está bloqueado por una hebra que, a su vez, está bloqueada esperando que se libere un objeto bloqueado por la primera hebra, ninguna de las dos hebras avanzará. Ambas se quedarán esperando indefinidamente. El interbloqueo es consecuencia de que el orden de adquisición de los cerrojos no sea siempre el mismo y la forma más evidente de evitarlo es asegurar que los cerrojos se adquieran siempre en el mismo orden.

- **Inanición** [*starvation*]: Cuando una hebra nunca avanza porque siempre hay otras a las que se les da preferencia. Por ejemplo, en el código correspondiente al cerrojo `RWLock`, un escritor nunca obtendrá el cerrojo para modificar el recurso compartido mientras haya lectores. Los escritores "se morirán de inanición".

- **Inversión de prioridad**: El planificador de CPU, usualmente parte del sistema operativo, decide en cada momento qué hebra, de entre todas las no bloqueadas, ha de disponer de tiempo de CPU. Usualmente, esta decisión viene influida por la prioridad de las hebras (que se puede fijar mediante la propiedad `Priority` de la clase `Thread` en C#). Sin embargo, al usar cerrojos se puede dar el caso de que una hebra de prioridad alta no avance nunca, justo lo contrario de lo que su prioridad nos podría hacer pensar. Imaginemos que tenemos tres hebras H1, H2 y H3, de mayor a menor prioridad. H3 está ejecutándose y bloquea el acceso al objeto O. H2 adquiere la CPU al tener más prioridad que H3 y comienza un cálculo muy largo. Ahora, H1 le quita la CPU a H2 y, al intentar adquirir el cerrojo de O, queda bloqueada. Entonces, H2 pasa a disponer de la CPU para proseguir su largo cómputo. Mientras, H1 queda bloqueada porque H3 no llega a liberar el cerrojo de O. Mientras que H3 no disponga de algo de tiempo de CPU,

H1 no avanzará y H2, pese a tener menor prioridad que H1, sí que lo hará. El planificador de CPU puede solucionar el problema eventualmente si todas las hebras disponibles avanzan en su ejecución, aunque sea más lentamente cuando tienen menor prioridad. De ahí la importancia de darle una prioridad alta a las hebras cuyo tiempo de respuesta haya de ser bajo, y una prioridad baja a las hebras que realicen tareas muy largas.

Todos estos problemas ponen de manifiesto la dificultad que supone trabajar con aplicaciones multihebra que comparten datos. Hay que asegurarse de asignar los recursos cuidadosamente para minimizar las restricciones de acceso en exclusiva a los recursos compartidos. En muchas ocasiones, por ejemplo, se puede simplificar notablemente la implementación de las aplicaciones multihebra si hacemos que cada una de las hebras trabaje de forma independiente, incluso con copias distintas de los mismos datos si hace falta. Sin datos comunes, los mecanismos de sincronización se hacen innecesarios.

Los mecanismos de sincronización anteriores utilizan memoria compartida y son útiles puntualmente en sistemas centralizados. En un sistema distribuido, sin embargo, la comunicación y sincronización entre procesos habrá de realizarse mediante paso de mensajes, ya sea mediante denominación directa (haciendo referencia al proceso con el que deseamos comunicarnos o canal a través del cual realizaremos la comunicación) o a través de nombres globales (usando buzones, puertos o señales).

Otros mecanismos de sincronización

Las clases incluidas en el espacio de nombres `System.Threading` de la biblioteca de clases de la plataforma .NET proporciona una amplia gama de mecanismos de sincronización además de los monitores y cerrojos ya vistos. De hecho, algunas de ellas se corresponden con los mecanismos proporcionados por el núcleo de Windows, a los cuales tradicionalmente se accedía a través del API Win32.

Entre los más mecanismos de sincronización del sistema operativo que permiten garantizar el acceso exclusivo a un recurso compartido se encuentran las clases derivadas de `System.Threading.WaitHandle`:

- Los **eventos**, que no hay que confundir con los eventos que se producen en la interfaz de usuario y gobiernan el funcionamiento de las aplicaciones Windows, se utilizan para notificarle a una hebra que esté esperando que se ha producido algún tipo de evento. Por medio de un evento, se activa una señal que puede utilizarse para coordinar la ejecución de distintas hebras. Las clases `AutoResetEvent` y `ManualResetEvent` encapsulan los eventos Windows en la plataforma .NET. Mediante el método `Set()` asociado al evento se activa una señal a la que puede esperar una hebra con una simple llamada a `WaitOne()`. El método `Reset`, por su parte, desactiva la señal y es necesario invocarlo siempre cuando el evento es de tipo `ManualResetEvent`. Con

`AutoResetEvent`, la señal se desactiva sola en cuanto una hebra recibe la notificación del evento.

- Otra primitiva de sincronización la proporcionan las **variables de exclusión mutua**, implementadas en .NET por medio de la clase `Mutex`. Para solicitar el acceso exclusivo a un recurso compartido, se usa de nuevo el método `WaitOne()` asociado al mutex. Una vez utilizado el recurso compartido, ha de liberarse la variable de exclusión mutua con una llamada a `ReleaseMutex()` o `Close()`, para que otras hebras puedan acceder al recurso compartido. Una vez en posesión del mutex, se pueden realizar múltiples llamadas a `WaitOne()`, si bien luego habrá que llamar el mismo número de veces a `ReleaseMutex()` para liberar el cerrojo, algo que sucede automáticamente si termina la ejecución de la hebra propietaria del mismo.

```
Mutex mutex = new Mutex(false);

mutex.WaitOne();
...
mutex.Close();
```

Como siempre, al usar este tipo de mecanismos, las llamadas a `WaitOne()` pueden llevar un parámetro opcional que indique un tiempo máximo de espera, pasado el cual la hebra reanuda su ejecución aunque no se haya cumplido la condición por la que estaba esperando. De este modo, podemos asegurarnos de que nuestras aplicaciones no se quedarán indefinidamente a la espera de un suceso que puede no llegar a producirse.

Además, la clase base `WaitHandle` incluye un método llamado `WaitAll()` que se puede utilizar para esperar simultáneamente a todos los elementos de una matriz, ya sean éstos eventos o variables de exclusión mutua.

Como se puede apreciar, los eventos y las variables de exclusión mutua proporcionan la misma funcionalidad que los monitores y cerrojos vistos con anterioridad. Su principal ventaja reside en que se pueden utilizar para sincronizar hebras que estén ejecutándose en distintos espacios de memoria, convirtiéndose así en un auténtico mecanismo de comunicación entre procesos.

Aparte de las ya vistas, el espacio de nombres `System.Threading` incluye otras dos clases que se pueden emplear para coordinar la ejecución de tareas que se han de realizar concurrentemente:

- Se pueden usar temporizadores gracias a la clase `Timer`, que proporciona un mecanismo para ejecutar tareas periódicamente. El sistema operativo se encargará de llamar al método referenciado por un delegado de tipo `TimerCallback` cada vez que expire el temporizador. Nótese que el método llamado por el temporizador se ejecutará en una hebra independiente a la hebra en la que se creó

el temporizador, por las que deberemos ser cuidadosos si dicho método accede a datos compartidos con otras hebras.

- La clase `Interlocked` suministra un método un mecanismo de exclusión mutua a bajo nivel que protege a las aplicaciones multihebra de los errores que se pueden producir cuando se efectúa un cambio de contexto mientras se está actualizando el valor de una variable. Sus métodos estáticos `Increment()` y `Decrement()` sirven para incrementar o decrementar el valor de una variable entera de forma atómica, ya que incluso algo aparentemente tan simple no se ejecuta de forma atómica en el hardware del ordenador. Esta operación se realiza, en realidad, en tres pasos: cargar el dato en un registro del microprocesador, modificar el valor y volver a almacenarlo en memoria. Si queremos que los tres pasos se ejecuten como uno sólo, en vez de los operadores `++` y `--`, tendremos que usar los métodos estáticos `Increment()` y `Decrement()`. Esta clase proporciona, además, otros dos métodos estáticos que permiten establecer el valor de una variables de forma atómica, con `Exchange()`, y reemplazar el valor de una variable sólo si ésta es igual a un valor determinado con la función `CompareExchange()`, lo que puede ser útil para comprobar que la variable no ha sido modificada desde que se accedió a ella por última vez.

Como estas páginas demuestran, la gama de mecanismos a nuestra disposición para garantizar el acceso en exclusiva a un recurso compartido por varias hebras es bastante amplia. Muchos de estos mecanismos son equivalentes y sólo difieren en su implementación, por lo que su uso resulta indistinto en la mayor parte de las situaciones con las que nos podamos encontrar.

Operaciones asíncronas

Antes de pasar a estudiar distintos mecanismos de comunicación entre procesos, cerraremos este capítulo retomando la ejecución asíncrona de delegados en un contexto de especial interés: la **realización de operaciones asíncronas de entrada/salida**.

Como ya hemos visto, el uso de hebras resulta especialmente interesante cuando hay que realizar tareas lentas, como puede ser el acceso a dispositivos de E/S a través de llamadas síncronas a métodos de las clases incluidas en el espacio de nombres `System.IO`. Al llamar a un método de forma síncrona, la ejecución de nuestra hebra queda bloqueada hasta que el método finaliza su ejecución, algo particularmente inoportuno cuando se realiza una costosa operación de E/S. Si pudiésemos realizar esas llamadas de forma asíncrona, nuestra aplicación no tiene que detenerse mientras se realiza la operación de E/S. Como veremos a continuación, la plataforma .NET nos permite realizar operaciones asíncronas de E/S de forma completamente análoga a la ejecución asíncrona de delegados que vimos al hablar del uso de hebras en C#.

En primer lugar, lo que haremos será crear una estructura de datos auxiliar que usaremos para ir almacenando los datos que leamos de forma asíncrona. Esta estructura de datos auxiliar

puede ser tan sencilla como la definida por la siguiente clase:

Estructura de datos auxiliar

```
public class StateObject
{
  public byte[] bytes;

  public int size;

  public FileStream fs;
}
```

A continuación, lo que haremos será lanzar la operación de lectura de datos de forma síncrona, para lo cual recurrimos al método `BeginRead()` de la clase `System.IO.Stream`:

Inicio de la operación de lectura asíncrona:

```
...
StateObject   state    = new StateObject();
AsyncCallback callback = new AsyncCallback(CallbackFunction);

FileStream fs = new FileStream ( "fichero.txt",
                                 FileMode.Open,
                                 FileAccess.Read,
                                 FileShare.Read,
                                 1, true);
state.fs    = fs;
state.size  = fs.Length;
state.bytes = new byte[state.size];

fs.BeginRead (state.bytes, 0, state.size, callback, state);
...
```

Al tratarse de una operación de lectura, hemos de incluir un delegado `AsyncCallback` para que la operación de lectura "devuelva la llamada" una vez que se concluya la lectura de datos. Esa llamada, obviamente, se realizará en una hebra distinta a la hebra en la cual se invoca a `BeginRead()`. Si no fuese así, la operación bloquearía la hebra original y no sería asíncrona.

Para procesar los datos leídos, no tenemos que preocuparnos de crear una hebra independiente y que ésta espere a que termine la operación de lectura. La implementación de `System.IO.Stream` se encarga de hacerlo automáticamente por medio de una función *callback* ("de llamada devuelta"), que tendrá un aspecto similar al siguiente:

Procesamiento de los datos leídos

```
public static void CallbackFunction (IAsyncResult asyncResult)
{
  StateObject state = (StateObject) asyncResult.AsyncState;
  Stream      stream = state.fs;

  int bytesRead = stream.EndRead(asyncResult);

  if (bytesRead != state.size)
     throw new Exception("Sólo se han leído {0} bytes", bytesRead);

  stream.Close();

  // Operaciones con los datos (state.bytes)
  ...

  // Notificación de la finalización de la operación
  Monitor.Enter(WaitObject);
  Monitor.Pulse(WaitObject);
  Monitor.Exit(WaitObject);
}
```

Cuando no nos interesa obtener un resultado de la ejecución de la operación asíncrona, podemos simplemente iniciar la operación y olvidarnos de ella, como hicimos al ver por primera vez la ejecución asíncrona de delegados. Sin embargo, cuando esa operación ha de devolver un valor, las llamadas a `BeginInvoke()`, `BeginRead()` o `BeginWrite` devuelven un objeto de tipo `IAsyncResult` que se puede utilizar para acceder al resultado de la operación. Este objeto `IAsyncResult` se recibe como parámetro de la función *callback* y, en combinación con las operaciones `EndInvoke()`, `EndRead()` y `EndWrite`, se puede usar para acceder a los resultados de la operación asíncrona una vez que esta ha finalizado.

Dónde usar EndInvoke/EndRead/EndWrite

Obsérvese que, en el ejemplo, hemos utilizado la operación `EndRead()` de finalización de la operación asíncrona de lectura en una hebra independiente de la hebra que lanzó la operación con `BeginRead()`.

Si se utilizan operaciones asíncronas, la existencia de un único cerrojo puede producir el bloqueo de la aplicación. Una cadena de llamadas a métodos cruza el límite de una hebra cuando se utiliza `EndInvoke()`, `EndRead()` o `EndWrite`. Imaginemos una hebra H que adquiere el cerrojo L asociado a un recurso compartido. A continuación, esta hebra puede utilizar la ejecución asíncrona de delegados para iniciar una operación asíncrona A, que puede necesitar

Dónde usar EndInvoke/EndRead/EndWrite

acceder al recurso compartido bloqueado por L, por lo que la operación asíncrona queda bloqueada esperando que H libere L. Ahora bien, si H, a su vez, se bloquease esperando el resultado de la operación asíncrona (con una simple llamada a `EndInvoke()`, `EndRead()` o `EndWrite`), ni la hebra H ni la operación asíncrona A avanzarían, ya que A debería concluir antes de poder empezar. Interbloqueo.

De acuerdo con la situación anterior, las llamadas a `EndInvoke()`, `EndRead()` o `EndWrite` nunca deberían realizarse dentro de una sentencia `lock`. Desgraciadamente, eso es más fácil decirlo que hacerlo, pues no resulta fácil comprobar si una hebra, llegado determinado punto de su ejecución, posee algún cerrojo o no. Por tanto, siempre usaremos `EndInvoke()`, `EndRead()` o `EndWrite` desde el interior de la función *callback*, que se ejecuta de forma independiente a las demás hebras de nuestra aplicación.

Dadas las restricciones que impone el uso adecuado de la operación `EndRead`, para comprobar si la operación de lectura de datos ha finalizado, recurrimos a los métodos `Monitor.Wait()` y `Monitor.Pulse()`, que ya vimos en su momento. Usando estos métodos podemos comprobar fácilmente si la operación asíncrona ha finalizado su ejecución desde la hebra que la lanzó:

En espera de la finalización de la operación

```
Monitor.Enter(WaitObject);
Monitor.Wait(WaitObject);
Monitor.Exit(WaitObject);
```

Como podríamos esperar, la clase `System.IO.Stream` incluye también un par de métodos, `BeginWrite()` y `EndWrite()`, que permiten realizar operaciones de escritura de forma asíncrona. Su funcionamiento es completamente equivalente al de los métodos `BeginRead()` y `EndRead()` que acabamos de utilizar en esta sección

Obviamente, también podríamos haber realizado la operación asíncrona creando nosotros mismos una hebra independiente. Sin embargo, la invocación asíncrona de delegados proporcionada por la plataforma .NET simplifica la implementación de la aplicación. Es más, al usar delegados de esta forma se suele mejora el rendimiento de la aplicación. Al usar un *pool* de hebras compartidas para ejecutar de forma asíncrona los delegados, se evita que el número de hebras sea superior al que soporte el sistema.

Referencias

La programación con hebras no es un tema discutido con excesiva asiduidad, si bien pueden encontrarse bastantes artículos en Internet acerca del uso de hebras y procesos. A continuación se mencionan algunos que pueden resultar de interés:

- Andrew D. Birrell: *An Introduction to Programming with C# Threads*. Microsoft Corporation, 2003. (basado en un informe publicado por el mismo autor en 1989 cuando éste trabajaba para DEC [Digital Equipment Corporation], que ahora forma parte de HP tras ser adquirida por Compaq)

- Shawn Cicoria: *Proper Threading in Winforms .NET*. The Code Project (CodeProject.com), May 2003.

- Chris Sells: *Safe, Simple Multithreading in Windows Forms, Part 1*. MSDN, June 28, 2002.

- Chris Sells: *Safe, Simple Multithreading in Windows Forms, Part 2*. MSDN, September 2, 2002.

- Chris Sells: *Safe, Simple Multithreading in Windows Forms, Part 3*. MSDN, January 23, 2003.

- Ian Griffiths: *Give Your .NET-based Application a Fast and Responsive UI with Multiple Threads*. MSDN Magazine, February 2003

- Matthias Steinbart: *How to do Application Initialization while showing a SplashScreen*. The Code Project (CodeProject.com), January 2003.

- David Carmona: *Programming the Thread Pool in the .NET Framework.* MSDN, June 2002.

- Brian Goetz: *Concurrency made simple (sort of)*. IBM developerWorks, November 2002.

Muchos de estos artículos suelen estar enfocados para la resolución de problemas concretos con los que puede encontrarse un programador. Su utilidad radica precisamente en eso. Dado que estamos acostumbrados a pensar en programas secuenciales, normalmente no razonamos igual de bien acerca de procesos y hebras concurrentes. En estas situaciones, una rápida consulta en el Google puede ahorrarnos muchos quebraderos de cabeza. Un poco de ayuda siempre viene bien, aunque siempre hemos de mantener nuestro punto de vista crítico: ¿quién nos asegura que lo que encontremos en Internet funcione correctamente?