

Threads o Hilos

En .NET, cuando se lanza una aplicación se crea un proceso y dentro de este proceso un hilo de ejecución o thread para el método `Main`. Es posible, a la vez que se ejecuta el método `Main`, que la aplicación lance internamente nuevos hilos de ejecución en los que se ejecute el código de algún método.

Un ejemplo muy actual de utilización de threads es una aplicación de tipo servidor Web, como el IIS, el Apache Web Server, etc... Un servidor Web es una aplicación que, al ser lanzada y creado para ella un proceso y un hilo para el método `Main`, espera a que llegue una petición `http` de un cliente al puerto que escucha (el 80 generalmente) y cuando llega hace dos cosas:

- Crea un nuevo hilo de ejecución o thread en el que atiende la petición.
- Vuelve inmediatamente (en el hilo principal) a escuchar el puerto para atender nuevas peticiones (mientras se atiende la petición en el nuevo hilo).

Cada vez que llegue una nueva petición se repetirá el mismo proceso, de modo que se crearán nuevos hilos, pudiendo haber n hilos en ejecución simultáneamente, uno principal y $n-1$ atendiendo peticiones `http` de los clientes. Es de rigor puntualizar que al decir “simultáneamente” se piensa en condiciones ideales (un procesador por hilo), hablando en tales condiciones de concurrencia real de varios hilos o threads.

En equipos con menos procesadores que hilos o threads lanzados en un momento dado, se habla de concurrencia aparente, ya que todos los hilos no pueden estar ejecutándose a la vez. No obstante, no se ha de pensar que tener un solo procesador hace inútil lanzar más de un hilo o thread simultáneamente. Ni mucho menos, el 100% del tiempo de ejecución de un hilo no está ocupado el procesador (interacción con el usuario, entrada/salida, acceso a memoria), de modo que otro hilo puede aprovechar sus tiempos muertos. También es cierto que un exceso de threads o hilos resulta negativo, ya que se puede perder más tiempo “saltando” de un thread a otro (a esta operación se la llama “conmutación de contexto” e implica salvar y recuperar datos y registros de memoria) que en la ejecución real.

En Windows, aunque sólo se disponga de un procesador, se permite ejecutar varios hilos simultáneamente (concurrencia aparente). Lo que se hace es ofrecer un tiempo determinado de ejecución (`time slice` o “rodaja de tiempo”) a cada hilo (realmente son milisegundos). Cuando ese tiempo finaliza, Windows retoma el control y se lo cede a otro thread. De este modo se ofrece al usuario la ilusión de tener varias aplicaciones en ejecución simultáneamente y también se optimiza el uso de los recursos. A este modo de organizar la ejecución de varios threads se le llama `preemptive multitasking` (“multitarea preemptiva”)

En realidad, el sistema operativo y cualquier hilo que se lance ya son dos hilos, con lo cual la ejecución en Windows es siempre multihilo.

La clase Thread.

Esta clase pertenece al namespace `System.Threading`. Para crear un thread sólo hay que crear una instancia de esta clase. Sus métodos más importantes son:

- `start`: lanza el thread a ejecución.
- `suspend`: detiene momentáneamente la ejecución del thread.
- `resume`: activa el thread suspendido, es decir, lo vuelve a poner en ejecución.
- `abort`: aborta o para de modo inmediato la ejecución del thread.
- `join`: detiene el thread donde se invoca hasta que el thread para el que se le invoca termina.

Las propiedades más interesantes de la clase `Thread` son:

- `Name`: permite darle un nombre a un thread que lo distinga del resto.
- `CurrentThread`: contiene una referencia al thread que está actualmente en ejecución.

Ejecución de un thread.

Un thread no es más que un bloque de código vacío por defecto que es posible lanzar a ejecución de modo simultáneo a otros threads.

Para que el método `start` de la clase `Thread` lance un thread que ejecute un código real, ha de recibir como parámetro una referencia o delegate de tipo `ThreadStart` al punto de entrada del código real. `ThreadStart` es un delegate que se utiliza para referenciar el punto de entrada de un código que se desea sea el punto de entrada de un thread. Su definición es:

```
public delegate void ThreadStart();
```

Para ilustrar lo comentado, supóngase que se desea crear una aplicación desde la que se lance un thread que referencie una función que muestre 10 veces un mensaje ("Hola, soy el thread") en la consola. La aplicación lanzará también la función desde el thread principal de la aplicación (`Main`).

El código necesario para crearlo es:

```
using System;
using System.Threading;

namespace Threads1
{
    class PruebaThread
    {
        static void Main(string[] args)
        {
            Thread miThread = new Thread(new ThreadStart(MiFun));
            miThread.Start();
            MiFun();
        }

        public static void MiFun()
        {

```

```

        System.Console.WriteLine("Hola, soy el thread");
        for (int i=0; i<=10; i++)
            System.Console.WriteLine ("Iteración:" + i);
    }
}

```

Al llamar a `miThread.Start` se lanza un nuevo thread con lo que hay dos threads en ejecución, el principal y `miThread`. Si se ejecuta este ejemplo, el resultado se reproduce en la figura 10.1:

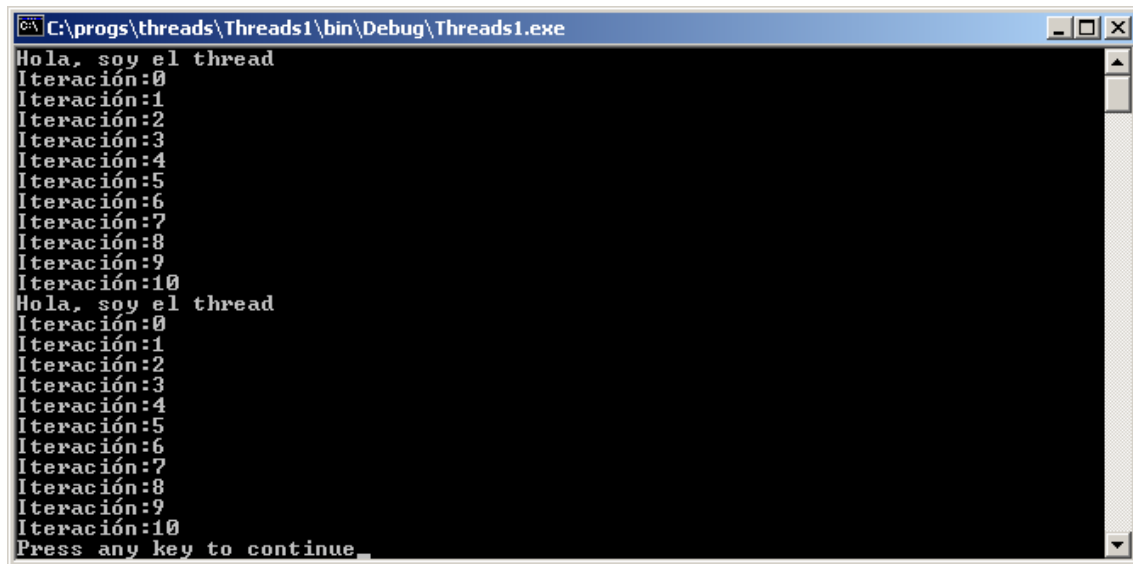


Figura 10.1

Este resultado provoca una pequeña desilusión, ya que parece que ambos threads se ejecutan de modo secuencial y además no hay modo de distinguir cuál es el principal y cuál el secundario.

Para conseguir un ejemplo más ilustrativo pueden utilizarse las propiedades `Name` y `CurrentThread`. Con la propiedad `Name` puede darse nombre a ambos threads y con la propiedad `CurrentThread` puede obtenerse el thread principal para darle nombre. Lo mejor es verlo con un ejemplo:

```

using System;
using System.Threading;

namespace Threads1
{
    class PruebaThread
    {
        static void Main(string[] args)
        {
            //Dar un nombre al thread principal para distinguirlo
            //del secundario cuando se muestre su información por
            //consola
            Thread.CurrentThread.Name = "Principal";
            Thread miThread = new Thread(new ThreadStart(MiFun));
            miThread.Name = "ThreadMiFun";
            miThread.Start();
        }
    }
}

```

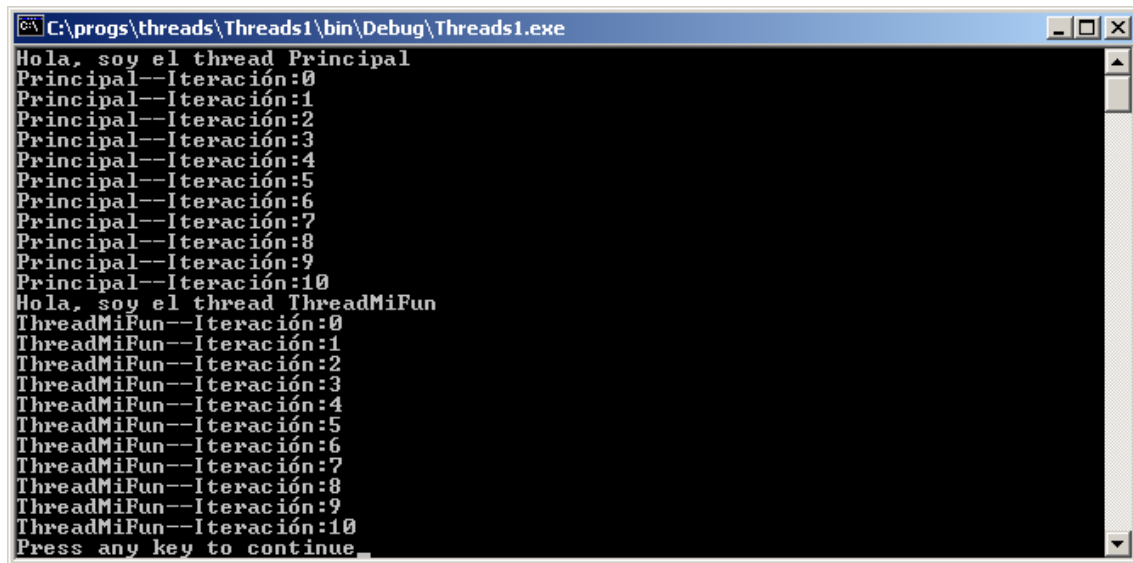
```

        MiFun();
    }

    public static void MiFun()
    {
        System.Console.WriteLine("Hola, soy el thread");
        for (int i=0; i<=10; i++)
            System.Console.WriteLine
                (Thread.CurrentThread.Name.ToString() + "--
                Iteración:" + i);
    }
}

```

El resultado en este caso será:



```

C:\progs\threads\Threads1\bin\Debug\Threads1.exe
Hola, soy el thread Principal
Principal--Iteración:0
Principal--Iteración:1
Principal--Iteración:2
Principal--Iteración:3
Principal--Iteración:4
Principal--Iteración:5
Principal--Iteración:6
Principal--Iteración:7
Principal--Iteración:8
Principal--Iteración:9
Principal--Iteración:10
Hola, soy el thread ThreadMiFun
ThreadMiFun--Iteración:0
ThreadMiFun--Iteración:1
ThreadMiFun--Iteración:2
ThreadMiFun--Iteración:3
ThreadMiFun--Iteración:4
ThreadMiFun--Iteración:5
ThreadMiFun--Iteración:6
ThreadMiFun--Iteración:7
ThreadMiFun--Iteración:8
ThreadMiFun--Iteración:9
ThreadMiFun--Iteración:10
Press any key to continue

```

Figura 10.2

Ahora, por lo menos, se sabe en cada línea qué thread la ha generado. Lo curioso es que el orden es el inverso del esperado y sigue siendo secuencial.

El orden de comienzo de los threads no tiene porqué ser el mismo en el que se lancen las funciones, depende de la prioridad de los threads y otros factores. El que sea secuencial se debe a que la ejecución lleva poco tiempo. Si se inicializa el contador “i” de la función `MiFun` a 1000, en lugar de a 10 podrá comprobarse cómo la ejecución no es secuencial, sino que cada hilo recibe una “rodaja de tiempo” (figura 10.3).

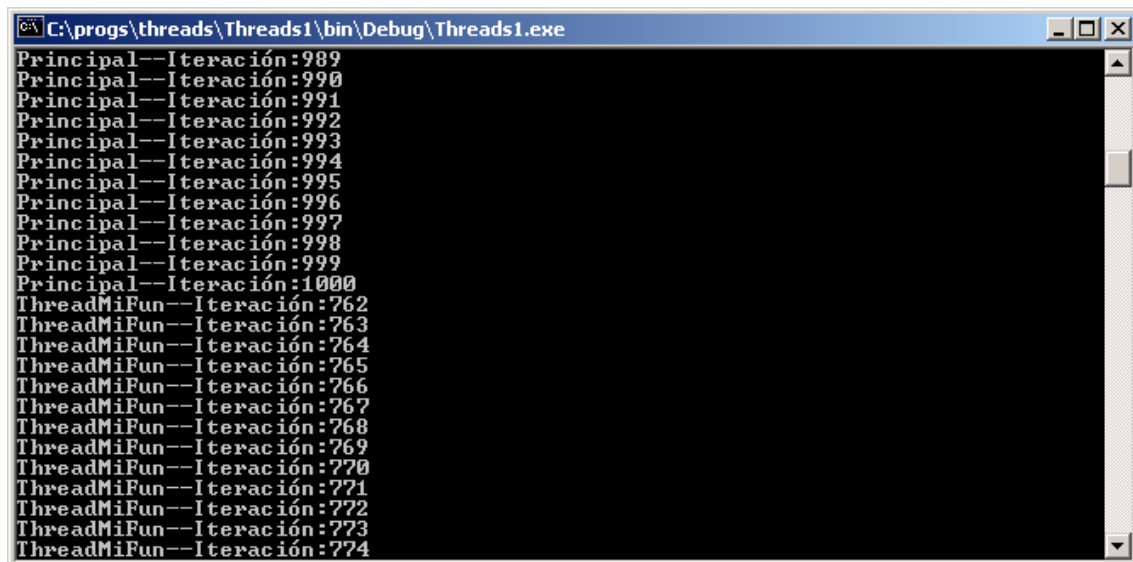


Figura 10.3

Parada y activación de un Thread.

Una vez un thread ha sido lanzado puede ser suspendido (`suspend`), reactivado (`resume`) o abortado (`abort`).

- `suspend` y `resume`: `suspend` suspende la ejecución de un thread momentáneamente. Un thread suspendido puede ser reactivado llamando a `resume`. Si no se utilizan bien pueden causar situaciones de bloqueo no recuperables.
- `abort`: lanza una excepción `ThreadAbortException` en el thread. El tratamiento por defecto para esta excepción es finalizar la ejecución del thread. No obstante si se captura la excepción, se puede programar el tratamiento que se desee.

Supóngase que se desea crear una aplicación que lance dos funciones (`MiFun` y `MiFun2`) en sendos threads. `MiFun` muestra números del 1 al 10 y `MiFun2` letras de la 'a' a la 'z'. Se desea también que se ejecute primero `MiFun` pero sólo hasta mostrar los 5 primeros números y quedar suspendida, tras lo cual debe ejecutarse `MiFun2` hasta acabar y luego debe reactivar a `MiFun`. El código para hacer esto es (se resaltan en **negrita** los puntos clave):

```
using System;
using System.Threading;

namespace Threads2
{
    class PararActivarThread
    {
        static Thread miThread = null;
        static void Main(string[] args)
        {
            miThread = new Thread(new ThreadStart(MiFun));
            miThread.Name = "ThreadMiFun";

            Thread      miThread2      =      new      Thread(new
            ThreadStart(MiFun2));
        }
    }
}
```

```

        miThread2.Name = "ThreadMiFun2";

        miThread.Start();
        miThread2.Start();
    }

    public static void MiFun()
    {
        System.Console.WriteLine("Hola, soy el thread " +
            Thread.CurrentThread.Name.ToString());
        for (int i=0; i<=10; i++)
        {
            System.Console.WriteLine
                (Thread.CurrentThread.Name.ToString() + "--
                Iteración:" + i);
            if (i==5)
            {
                Thread.CurrentThread.Suspend();
            }
        }
    }

    public static void MiFun2()
    {
        System.Console.WriteLine("Hola, soy el thread " +
            Thread.CurrentThread.Name.ToString());
        for (char c='a'; c<='z'; c++)
        {
            System.Console.WriteLine
                (Thread.CurrentThread.Name.ToString() + "--
                Iteración:" + c);
        }
        miThread.Resume();
    }
}

```

El resultado será:

```

C:\progs\threads\Threads2\bin\Debug\Threads2.exe
ThreadMiFun2--Iteración:h
ThreadMiFun2--Iteración:i
ThreadMiFun2--Iteración:j
ThreadMiFun2--Iteración:k
ThreadMiFun2--Iteración:l
ThreadMiFun2--Iteración:m
ThreadMiFun2--Iteración:n
ThreadMiFun2--Iteración:o
ThreadMiFun2--Iteración:p
ThreadMiFun2--Iteración:q
ThreadMiFun2--Iteración:r
ThreadMiFun2--Iteración:s
ThreadMiFun2--Iteración:t
ThreadMiFun2--Iteración:u
ThreadMiFun2--Iteración:v
ThreadMiFun2--Iteración:w
ThreadMiFun2--Iteración:x
ThreadMiFun2--Iteración:y
ThreadMiFun2--Iteración:z
ThreadMiFun--Iteración:6
ThreadMiFun--Iteración:7
ThreadMiFun--Iteración:8
ThreadMiFun--Iteración:9
ThreadMiFun--Iteración:10
Press any key to continue

```

Figura 10.4

Join.

El método `Join` pertenece a la clase `Thread` y lo que hace es detener la ejecución del thread donde se invoca hasta que el thread para el que se invoca termina.

Por ejemplo:

```
using System;
using System.Threading;

namespace Threads1
{
    class PruebaThread
    {
        static void Main(string[] args)
        {
            //Dar un nombre al thread principal para distinguirlo
            //del secundario cuando se muestre su información por
            //consola
            Thread.CurrentThread.Name = "Principal";
            Thread miThread = new Thread(new ThreadStart(MiFun));
            miThread.Name = "ThreadMiFun";
            miThread.Start();
            miThread.Join();
            MiFun();
        }

        public static void MiFun()
        ...
        ...
    }
}
```

La llamada a `Join` del ejemplo hace que el hilo principal detenga su ejecución hasta que acabe `miThread`. El efecto de esta llamada a `Join` es que la llamada a `MiFun` desde el hilo principal no se ejecute hasta que acabe `miThread`, con lo que en la consola se verá primero la salida correspondiente a `miThread`. En realidad, este ejemplo no tiene sentido, ya que fuerza a una ejecución secuencial de los threads, independientemente de que el sistema pueda o no ejecutarlos concurrentemente.

Para observar un ejemplo con sentido se propone un cambio en la clase `PararActivarThread`. El cambio consiste en mostrar un mensaje cuando acaben de ejecutarse los threads `miThread` y `miThread2`, indicando el mensaje “Han finalizado los threads”. En un principio, el código lógico es:

```
using System;
using System.Threading;

namespace Threads2
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class PararActivarThread
    {
        static Thread miThread = null;
        static void Main(string[] args)
        {
            //Dar un nombre al thread principal para distinguirlo
            //del secundario
            //cuando se muestre su información por consola
        }
    }
}
```

```

        //Thread.CurrentThread.Name = "Principal";
        miThread = new Thread(new ThreadStart(MiFun));
        miThread.Name = "ThreadMiFun";
        Thread miThread2 = new Thread(new
        ThreadStart(MiFun2));
        miThread2.Name = "ThreadMiFun2";

        miThread.Start();
        miThread2.Start();
        Console.WriteLine("Han finalizado los threads");
    }
    ...
    ...

```

Si se ejecuta la aplicación con este cambio, el resultado será:

```

C:\progs\threads\Threads2\bin\Debug\Threads2.exe
Han finalizado los threads
Hola, soy el thread ThreadMiFun
ThreadMiFun--Iteración:0
ThreadMiFun--Iteración:1
ThreadMiFun--Iteración:2
ThreadMiFun--Iteración:3
ThreadMiFun--Iteración:4
ThreadMiFun--Iteración:5
Hola, soy el thread ThreadMiFun2
ThreadMiFun2--Iteración:a
ThreadMiFun2--Iteración:b
ThreadMiFun2--Iteración:c
ThreadMiFun2--Iteración:d
ThreadMiFun2--Iteración:e
ThreadMiFun2--Iteración:f
ThreadMiFun2--Iteración:g
ThreadMiFun2--Iteración:h
ThreadMiFun2--Iteración:i
ThreadMiFun2--Iteración:j
ThreadMiFun2--Iteración:k
ThreadMiFun2--Iteración:l
ThreadMiFun2--Iteración:m
ThreadMiFun2--Iteración:n
ThreadMiFun2--Iteración:o
ThreadMiFun2--Iteración:p

```

Figura 10.5

Este no es el mensaje esperado ya que el mensaje “Han finalizado los threads” se muestra antes de que comiencen.

A pesar de no ser el mensaje esperado tiene mucho sentido. Si se observa de nuevo el código, teniendo en cuenta que se ejecuta en el `Main`, es decir, en el hilo principal:

```

miThread.Start();
miThread2.Start();
Console.WriteLine("Han finalizado los threads");

```

Se puede deducir que el hilo principal lanza `miThread`, después lanza `miThread2` e inmediatamente después sigue en ejecución, en la instrucción `Console.WriteLine`. Como el hilo principal tiene la más alta de las prioridades lo primero que se ejecuta es `Console.WriteLine`.

Llamando al método `Join` desde el hilo principal, para los hilos secundarios, se puede corregir esta situación.

```

miThread.Start();

```



```
miThread2.Start();
miThread.Join();
miThread2.Join();
Console.WriteLine("Han finalizado los threads");
```

En este código se indica que el thread principal se ha de detener hasta que `miThread` termine y hasta que `miThread2` termine (poner un solo `Join` esperaría por un solo thread y continuaría, aquí se desea continuar sólo cuando los dos threads hayan acabado). Si se ejecuta ahora la aplicación, el resultado será esperado (figura 10.6):

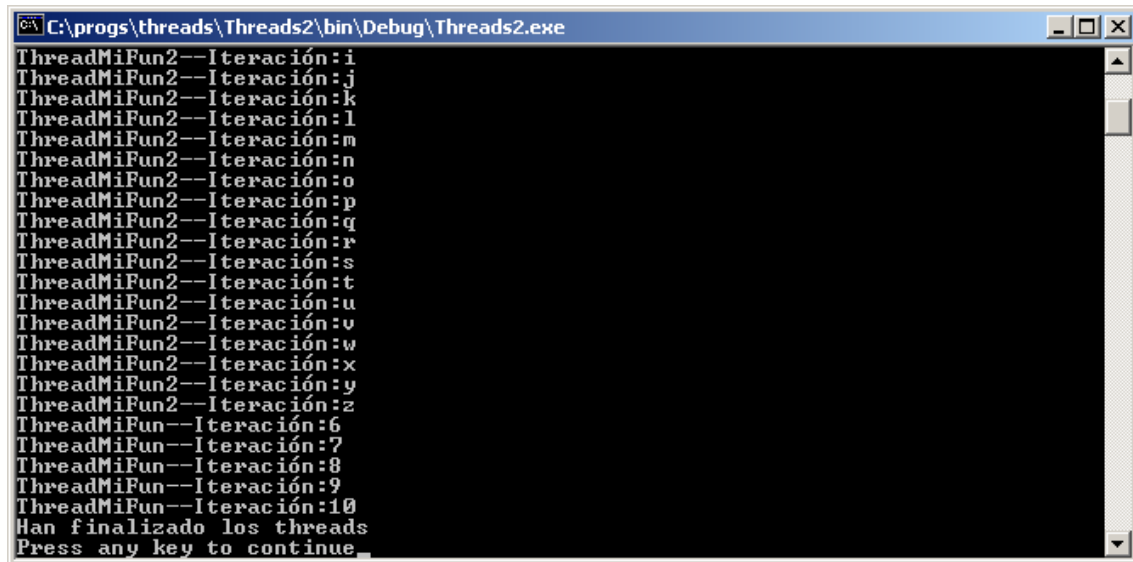


Figura 10.6

El método `Join` es realmente un método de sincronización (la sincronización se explica más adelante) y consiste en que quien lo invoca (en este caso el hilo principal) ejecuta el método `Wait` (ver la clase `Monitor`, más adelante), deteniéndose, y el thread sobre el que es invocado ejecuta el método `Pulse` (ver la clase `Monitor`, más adelante) cuando termina, permitiendo a quien se detuvo mediante un `Wait` reanudar su ejecución.

De lo comentado se deduce un peligro. ¿Qué sucede si el thread para el que se ha llamado a `Join` no termina o no llega nunca a llamar a `Pulse`? En tal caso, el hilo desde el que se ha llamado a `Join` (desde el que se ha hecho la `Wait`) queda bloqueado indefinidamente. Para solucionar este problema se permite que cuando se llama a `Join` se fije un tiempo máximo de espera. Para ello se ha de pasar un parámetro de tipo entero a `Join` indicando el número de milisegundos que como máximo se puede esperar (`Join` admite también un parámetro de la clase `TimeSpan`). Por ejemplo:

```
miThread.Start();
miThread2.Start();
miThread.Join(100);
miThread2.Join(100);
Console.WriteLine("Han finalizado los threads");
```

El tiempo máximo que espera el thread principal por ambos threads secundarios es de 100 milisegundos.

Prioridades.

Cuando se lanzan varios threads para que se ejecuten simultáneamente, se les asignan las “rodajas de tiempo” en función del valor que tenga su propiedad `Priority`.

La propiedad `Priority` es una propiedad pública de la clase `Thread` (no es `static`). Realmente es una enumeración del tipo `ThreadPriority`, cuyos posibles valores son:

Valor	Descripción
<code>AboveNormal</code>	El thread tiene la prioridad por encima de la normal.
<code>BelowNormal</code>	El thread tiene la prioridad por debajo de la normal.
<code>Highest</code>	El thread tiene la prioridad más alta.
<code>Lowest</code>	El thread tiene la prioridad más baja.
<code>Normal</code>	El thread tiene la prioridad normal.

Tabla 10.1

Un ejemplo de la influencia que tiene la prioridad de un thread se puede observar en la aplicación en que se lanzaba `MiFun` dos veces, una en un thread secundario y otra en el principal. A pesar de lanzar primero el secundario y luego el principal se ejecutaba primero el principal y después el secundario (ya que el principal tiene la prioridad, por defecto, más alta que el secundario).

Pueden hacerse pruebas con las prioridades de los threads en el ejemplo anterior, tanto para consultarlas como para cambiarlas, observando el resultado. Por ejemplo, si se desea que la prioridad del segundo thread (`MiThread2`) esté por encima de lo normal, se ha de hacer:

```
miThread2.Priority = ThreadPriority.AboveNormal;
```

Sincronización.

La sincronización de threads consiste en asegurar que distintos threads acceden de modo coordinado a recursos compartidos.

El caso más sencillo es la sincronización del uso del procesador, lo cual se lleva a cabo por el sistema operativo mediante la “multitarea preemptiva” (“rodajas de tiempo”). Pero existen otras situaciones en las que la sincronización es necesaria y no la hace por defecto el sistema operativo.

lock.

La sentencia `lock` bloquea el acceso a un bloque de código, asegurando que sólo el thread que lo ha bloqueado tiene acceso a tal bloque.

Supóngase la siguiente clase:

```
class Cuenta
{
    int saldo;
```

```

Random num = new Random();

internal Cuenta(int saldo_inicial)
{
    saldo = saldo_inicial;
}

int Realizar_Transaccion(int cantidad)
{
    if (saldo >= cantidad)
    {
        //El método static sleep detiene la ejecución del
        //thread durante 5 mseg.
        Thread.Sleep(5);
        saldo = saldo - cantidad;
        System.Console.Write
            (Thread.CurrentThread.Name.ToString());
        System.Console.WriteLine("-- Saldo= " +
                                   saldo.ToString());

        return saldo;
    }
    else
    {
        //si el balance es menor que la
        //cantidad que se desea retirar
        //se deniega la transacción
        System.Console.Write
            (Thread.CurrentThread.Name.ToString());
        System.Console.WriteLine("-- Transacción denegada.");
        System.Console.WriteLine("-- El saldo sería= " +
                                   (saldo - cantidad));

        return 0;
    }
}

internal void Realizar_Transacciones()
{
    for (int i = 0; i < 5; i++)
    {
        //devuelve un número aleatorio entre -50 y 100
        Realizar_Transaccion(num.Next(-50, 100));
    }
}

```

El método más importante es `Realizar_Transaccion` que recibe como parámetro un valor `cantidad` y en caso de que sea menor que el `saldo`, se lo resta. Si la cantidad es mayor que el `saldo`, no se hace la resta y se deniega la transacción.

El método `Realizar_Transacciones` se ha creado para poder simular varias transacciones sobre la cuenta llamándolo sólo una vez.

A continuación se muestra una clase `Prueba_Cuenta` que crea una instancia de la clase `Cuenta` y diez threads que realizan transacciones sobre la misma mediante el método `Realizar_Transacciones`.

```

class Prueba_Cuenta
{

```

```

static internal Thread[] threads = new Thread[10];

public static void Main()
{
    Cuenta c1 = new Cuenta (0);
    for (int i = 0; i < 10; i++)
    {
        Thread t = new Thread(new
            ThreadStart(c1.Realizar_Transacciones));
        threads[i] = t;
        threads[i].Name = "Thread-" + i.ToString();
    }
    for (int i = 0; i < 10; i++)
    {
        threads[i].Start();
    }
}

```

Si se crea una aplicación con las clases anteriores y se ejecuta, el resultado es el de la figura 10.7:

```

C:\progs\threads\Threads3\bin\Debug\Threads3.exe
Thread-2-- Transacción denegada.-- El saldo sería= -54
Thread-2-- Transacción denegada.-- El saldo sería= -159
Thread-2-- Transacción denegada.-- El saldo sería= -159
Thread-3-- Saldo= -92
Thread-3-- Transacción denegada.-- El saldo sería= -159
Thread-3-- Transacción denegada.-- El saldo sería= -143
Thread-5-- Saldo= -138
Thread-5-- Transacción denegada.-- El saldo sería= -171
Thread-7-- Saldo= -145
Thread-7-- Transacción denegada.-- El saldo sería= -172
Thread-7-- Transacción denegada.-- El saldo sería= -144
Thread-8-- Saldo= -216
Thread-8-- Transacción denegada.-- El saldo sería= -237
Thread-8-- Transacción denegada.-- El saldo sería= -234
Thread-8-- Transacción denegada.-- El saldo sería= -174
Thread-9-- Saldo= -290
Thread-9-- Transacción denegada.-- El saldo sería= -370
Thread-9-- Transacción denegada.-- El saldo sería= -361
Thread-9-- Transacción denegada.-- El saldo sería= -312
Thread-9-- Transacción denegada.-- El saldo sería= -351
Thread-9-- Transacción denegada.-- El saldo sería= -251
Thread-0-- Saldo= -359
Thread-0-- Transacción denegada.-- El saldo sería= -317
Thread-0-- Transacción denegada.-- El saldo sería= -350
Press any key to continue

```

Figura 10.7

Pero parece no tener sentido, el código no permite que la cuenta llegue a saldo negativo y no se han denegado todas las transacciones que lo dejan negativo.

La realidad es que sí tiene sentido. El código de la función `Realizar_Transaccion` es invocado por todos los threads en ejecución y por tanto compartido por todos ellos. Si en el momento en que un thread ha comprobado que el saldo (100 por ejemplo) es mayor que la cantidad pedida (80 por ejemplo), justo antes de que haga la resta, se le quita el control y se le da a otro, el nuevo thread creará que el saldo es el que hay (100) y si se le pide que reste 50 lo hará, quedando el saldo a 50. Cuando el thread interrumpido recupere el control, realizará la resta en la que se lo quitaron, restando 80 y dejando el saldo a -30.

Para resolver este problema, ningún thread debiera perder el control del procesador mientras ejecuta el método `Realizar_Transaccion`, es decir, mientras un thread ejecuta el método `Realizar_Transaccion` éste debe estar bloqueado a los demás threads, que esperarán para utilizarlo a que acabe el thread que lo bloquea, momento en el que será desbloqueado.

Esto que explicado es tan largo, se reduce a introducir el código del método `Realizar_Transaccion` en una sentencia `lock`.

```
int Realizar_Transaccion(int cantidad)
{
    //comentar este método para ver el problema de no usarlo
    lock(this)
    {
        if (saldo >= cantidad)
        {
            Thread.Sleep(5);
            saldo = saldo - cantidad;
            System.Console.Write
                (Thread.CurrentThread.Name.ToString());
            System.Console.WriteLine("-- Saldo= " +
                                    saldo.ToString());
            return saldo;
        }
        else
        {
            //si el balance es menor que la
            //cantidad que se desea retirar
            //se deniega la transacción
            System.Console.Write
                (Thread.CurrentThread.Name.ToString());
            System.Console.Write ("-- Transacción denegada.");
            System.Console.WriteLine ("-- El saldo sería= " +
                                    (saldo - cantidad));
            return 0;
        }
    }
}
```

El resultado de ejecutar la aplicación con los nuevos cambios se reproduce en la figura 10.8:

```

C:\progs\threads\Threads3\bin\Debug\Threads3.exe
Thread-4-- Transacción denegada.-- El saldo sería= -17
Thread-5-- Saldo= 10
Thread-6-- Saldo= 59
Thread-7-- Saldo= 10
Thread-8-- Saldo= 12
Thread-9-- Transacción denegada.-- El saldo sería= -59
Thread-1-- Transacción denegada.-- El saldo sería= -18
Thread-2-- Transacción denegada.-- El saldo sería= -50
Thread-3-- Transacción denegada.-- El saldo sería= -63
Thread-4-- Transacción denegada.-- El saldo sería= -42
Thread-5-- Transacción denegada.-- El saldo sería= -74
Thread-6-- Transacción denegada.-- El saldo sería= -73
Thread-7-- Saldo= 28
Thread-7-- Saldo= 2
Thread-8-- Saldo= 22
Thread-9-- Transacción denegada.-- El saldo sería= -51
Thread-1-- Saldo= 62
Thread-2-- Saldo= 9
Thread-3-- Transacción denegada.-- El saldo sería= -84
Thread-4-- Transacción denegada.-- El saldo sería= -69
Thread-5-- Transacción denegada.-- El saldo sería= -88
Thread-6-- Transacción denegada.-- El saldo sería= -4
Thread-9-- Transacción denegada.-- El saldo sería= -12
Thread-8-- Transacción denegada.-- El saldo sería= -66
Press any key to continue.

```

Figura 10.8

Como puede observarse, la cuenta no se queda con saldo negativo.

System.Threading.Monitor

La clase `Monitor` implementa el concepto de monitor de sincronización. Del mismo modo que existe esta clase existen otras como `Mutex`, etc...

No es el objetivo aquí profundizar en los conceptos del multihilo y de la sincronización. No obstante sí se comentará algún aspecto de la clase `Monitor`, ya que la sentencia `lock` equivale a utilizar un `Monitor` llamando a sus métodos `Enter` (al comienzo del bloque) y `Exit` (al final del bloque).

Los métodos más importantes de la clase `monitor` son:

- `Enter`: bloquea el bloque de código al que precede.
- `TryEnter`: es similar a `Enter` pero no bloquea o produce sólo un bloqueo temporal.
- `Exit`: libera el bloque.
- `Wait`: Detiene al thread que lo llama (muy importante, no bloquea el código a otros threads sino que detiene al thread llamador) dejándolo en espera de que otro thread le notifique que puede seguir mediante el método `Pulse`. También libera el bloqueo que haya hecho ese thread si es que lo hay.
- `Pulse`: notifica a un thread en la cola de espera (`Wait`) que puede continuar.
- `PulseAll`: realiza la notificación a todos los threads de la cola de espera.

A continuación, como ejemplo, se sustituye en el ejercicio anterior `lock` por las correspondientes llamadas a `Enter` y `Exit` en la clase `Monitor`.

```

int Realizar_Transaccion(int cantidad)
{
    //comentar este método para ver el problema de no usarlo

```

```

//lock(this)
Monitor.Enter(this);
{
    if (saldo >= cantidad)
    {
        Thread.Sleep(5);
        saldo = saldo - cantidad;
        System.Console.Write
            (Thread.CurrentThread.Name.ToString());
        System.Console.WriteLine ("-- Saldo= " +
                                   saldo.ToString());

        Monitor.Exit(this);
        return saldo;
    }
    else
    {
        //si el balance es menor que la
        //cantidad que se desea retirar
        //se deniega la transacción
        System.Console.Write
            (Thread.CurrentThread.Name.ToString());
        System.Console.Write ("-- Transacción denegada.");
        System.Console.WriteLine ("-- El saldo sería= " +
                                   (saldo - cantidad));

        Monitor.Exit(this);
        return 0;
    }
}

```

Se puede observar que las llamadas a `Exit` han de ser antes de `return` o no se ejecutarán.

Una apreciación que puede hacerse es que al poder decidir cuándo bloquear y cuándo desbloquear podría afinarse aún más, realizando el desbloqueo cuando ya se haya hecho la resta del saldo y la cantidad.

```

int Realizar_Transaccion(int cantidad)
{
    //comentar este método para ver el problema de no usarlo
    //lock(this)
    Monitor.Enter(this);
    {
        if (saldo >= cantidad)
        {
            Thread.Sleep(5);
            saldo = saldo - cantidad;
            Monitor.Exit(this);
            System.Console.Write
                (Thread.CurrentThread.Name.ToString());
            System.Console.WriteLine ("-- Saldo= " +
                                       saldo.ToString());
            return saldo;
        }
        else
        {
            //si el balance es menor que la
            //cantidad que se desea retirar
            //se deniega la transacción
            Monitor.Exit(this);

```

```
        System.Console.WriteLine
        (Thread.CurrentThread.Name.ToString());
        System.Console.WriteLine      ("--      Transacción
denegada.");
        System.Console.WriteLine ("-- El saldo sería= "
+ (saldo - cantidad));
        return 0;
    }
}
}
```

De este modo se optimiza el tiempo de ejecución, al no perder tiempo en bloqueos innecesarios.

Es muy importante utilizar bien la clase `Monitor` y nunca dejar un bloqueo sin desbloqueo posterior, ya que se puede llegar a una situación de deadlock o “abrazo mortal”, quedando la aplicación completa bloqueada sin posibilidad de continuar.

Para probar esto basta con eliminar la llamada a `Monitor.Exit(this)` de cualquiera de las ramas del `if`.