Unidad

5

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 10

Threads

En este Capítulo

- Threads o subprocesos
- Las Tres Partes de un Thread
- Creación
- Comienzo de un Thread
- Schedule de un Thread
- Los estados de un thread
- Tiempos de ejecución de un thread
- Terminando un Thread
- Control Básico de Threads
- Thread en espera
- Interrupción de ejecución
- Usando la Palabra Clave lock o SyncLock (C# o VB)
- La Bandera para el Bloqueo de Objetos
- Diagrama de Estado de Sincronización de un Thread
- Interbloqueo (deadlock)
- Interacción de los subprocesos: la clase Monitor
- Modelo de control de la sincronización
- Ejemplo de interacción de threads

Threads o subprocesos

El subprocesamiento permite al programa de C# o VB efectuar procesamientos simultáneos de modo que se puedan realizar varias operaciones a la vez. Por ejemplo, puede utilizar el subprocesamiento para supervisar los datos introducidos por el usuario, realizar tareas en segundo plano y controlar entradas de dispositivos periféricos de manera simultánea.

Los subprocesos tienen las siguientes propiedades:

- Los subprocesos permiten al programa realizar procesamientos simultáneos.
- El espacio de nombres System.Threading de .NET Framework facilita el uso de subprocesos.
- Los subprocesos comparten los recursos de la aplicación.

De forma predeterminada, un programa de C# o VB tiene un subproceso, el thread principal que es a través del cual comienza la ejecución de un programa. Sin embargo, se pueden crear subprocesos auxiliares para ejecutar código en paralelo con el subproceso primario. Estos subprocesos se denominan a menudo subprocesos o threads de trabajo.

Los subprocesos de trabajo se pueden utilizar para realizar tareas que llevan mucho tiempo o en las que este es crucial, sin mantener bloqueado el subproceso primario mientras la realiza. Por ejemplo, los subprocesos de trabajo se utilizan a menudo en aplicaciones de servidor para atender las solicitudes entrantes sin esperar a que la solicitud anterior se haya completado. Los threads de trabajo también se utilizan para realizar tareas "en segundo plano" en aplicaciones de escritorio a fin de que el subproceso principal, que controla los elementos de la interfaz de usuario, siga respondiendo a las acciones del usuario.

Los threads resuelven los problemas de rendimiento y capacidad de respuesta, pero también pueden originar problemas en materia de uso compartido de recursos, como interbloqueos y condiciones de carrera (cuando dos threads compiten por obtener un recurso). Es mejor utilizar varios subprocesos para tareas que requieran recursos diferentes, como tratamiento de archivos y conexiones de red. Asignar varios threads a un único recurso probablemente produciría problemas de sincronización, y el bloqueo frecuente de los subprocesos puede determinar que se pierda la ventaja de procesamiento obtenida al procesarlo de esta manera.

Una estrategia común es utilizar subprocesos para realizar trabajos en tareas que llevan mucho tiempo o en las que el tiempo es crucial y que no requieran muchos de los recursos utilizados por otros threads. Cuando un recurso es necesario por varios subprocesos, se debe preservar la ejecución de los mismos para que no se corrompa la información manejada por uno con la acción del otro. Es estos casos, el espacio de nombres System.Threading proporciona clases para sincronizar los subprocesos. Estas clases incluyen Mutex, Monitor, Interlocked, AutoResetEvent y ManualResetEvent.

En este capítulo sólo se tratará la clase Monitor que permite sincronizar threads para memoria manejada por el CLR.

Se puede utilizar algunas de estas clases, o todas ellas, para sincronizar las actividades de varios subprocesos, pero los lenguajes C# y Visual Basic proporcionan una parte de la compatibilidad con los subprocesos dentro del mismo lenguaje. Por ejemplo, las instrucciones `lock` ó `SyncLock` (C# ó VB) proporcionan características de sincronización a través del uso implícito de Monitor.

Las Tres Partes de un Thread

Todo programa cuando comienza, se ejecuta sobre un thread principal o "main thread", el cual determina la secuencia de ejecución de las sentencias del programa.

.Net posee la habilidad de ejecutar más de un thread dentro de un mismo programa. Esto es, puede crear threads adicionales cuando se lo indique en el código y su creación o manejo es parte del Framework de .Net.

La pregunta fundamental, en caso de tener más de una secuencia de procesamiento en un mismo programa, es ¿cómo se comporta los elementos del programa en esta situación?

Como se mencionó anteriormente, todo programa respeta la estructura básica que muestra la siguiente figura:

Dónde:

- **Zona de Texto**: es el lugar donde se almacenan todas las instrucciones de programa
- **Memoria Estática**: es donde se almacenan las variables globales (según el lenguaje) y las estáticas.
- **Stack**: es el lugar donde se almacenan los parámetros y variables locales de un método. Se agranda o reduce dinámicamente en cada invocación a un método
- **Heap**: es la memoria asignada a un programa que no se encuentra en uso actualmente pero que se puede pedir dinámicamente.

Es claro que los valores que gobiernan la instrucción a ejecutar deberá ser propia de cada thread, ya que cada uno puede estar en ejecutándose en una sentencia diferente de una misma parte del programa.

Por otro lado, el stack también será propio de cada thread por el mismo motivo, ya que cada uno de ellos puede estar en métodos distintos y la secuencia de llamadas puede estar en diferentes etapas.

Sin embargo, la memoria estática y el heap son comunes a todos ellos porque son accesibles desde todo el programa, particularmente, desde cada thread. Esto genera un problema que se analizará posteriormente que tiene que ver con la concurrencia a memoria compartida.

En conclusión, cada thread para ser único necesita manejar diferentes stack, instrucciones a ejecutar en el código y tiempo de procesador para hacerlo. Por lo tanto, se resume que cada thread posee como si fueran propios y sólo le pertenecieran:

- Tiempo de CPU
- Datos
- Código

Creación

Conociendo conceptualmente las necesidades de un thread para ejecutarse, el próximo paso es ver como se crea uno.

Ejemplo

C#

```
namespace creacion
{
    class ThreadHola
    {
        public void EjecutarThread()
        {
            int i;
            i = 0;
            while (true)
            {
                Console.WriteLine("Hola desde EjecutarThread " + i++);
                if (i == 50)
                {
                    break;
                }
            }
        }
    }
}

using System;
using System.Threading;

namespace creacion
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            int i = 0;
            ThreadHola r = new ThreadHola();
            Thread t = new Thread(new ThreadStart(r.EjecutarThread));
            t.Start();
            while (true)
            {
                Console.WriteLine("Hola desde Main " + i++);
                if (i == 50)
                {
                    break;
                }
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class ThreadHola
    Public Sub EjecutarThread()
        Dim i As Integer = 0

        While True
            i += 1
            Console.WriteLine("Hola desde EjecutarThread " + i.ToString)
            If i = 50 Then
                Exit While
            End If
        End While
    End Sub
End Class

Imports System.Threading

Module Module1

    Sub Main()
        Dim i As Integer = 0
        Dim r As New ThreadHola()
        Dim t As Thread = New Thread(New ThreadStart(AddressOf r.EjecutarThread))
        t.Start()
        While True
            i += 1
            Console.WriteLine("Hola desde Main " + i.ToString)
            If i = 50 Then
                Exit While
            End If
        End While
        Console.ReadKey()
    End Sub
End Module
```

Si se ejecuta reiteradamente este programa, se notará que no genera la misma salida siempre. La razón de esto es como se le otorga a cada thread el permiso de ejecución, es decir, existe un componente llamado Scheduler en el sistema operativo que es el encargado de otorgar tiempo de ejecución a los procesos y subprocesos. La manera en que se otorgan estos tiempos de ejecución está determinada por un algoritmo interno que analiza constantemente los procesos y subprocesos en ejecución en base a varios elementos, como la prioridad y el uso de memoria entre otros y determina quién se ejecuta en qué momento. Por el momento, lo único relevante a tener en cuenta es ***los threads NO se ejecutan por defecto en forma secuencial uno detrás del otro, sino que se ejecutan el orden y el tiempo que determina el Scheduler***.

La mecánica de creación de los threads del ejemplo es simple:

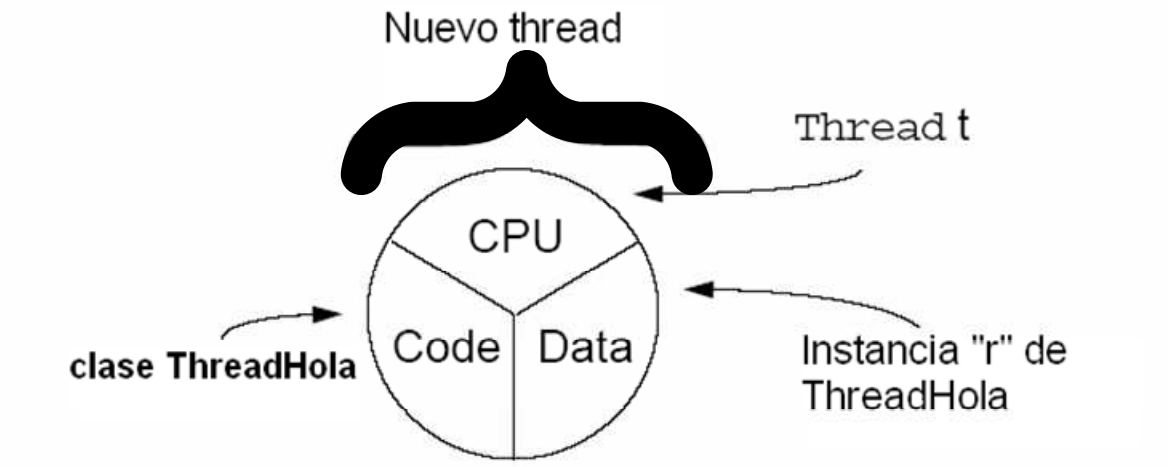
1. Se crea una nueva instancia del delegado ThreadStart al que se le asigna la dirección del método en cual comenzará la ejecución del thread. Dicho método no debe retornar valores ni definir parámetros, pero puede ser estático.
2. Se pasa la referencia del delegado como argumento del constructor de la clase Thread
3. En algún punto del código se inicia el thread invocando al método Start.
4. Opcionalmente los pasos 1 y 2 se pueden ejecutar en una misma línea de código

Cuando se trabaja en un ambiente multithreading con objetos, se pueden crear múltiples threads de dos formas:

- Multiple threads de la misma instancia de un objeto
- Threads que comparten el mismo código pero en objetos diferentes

La segunda situación es cuando se crean varias instancias de un mismo tipo de objeto y cada cual se invoca en threads diferentes.

Un gráfico que demuestra esquemáticamente como trabaja el nuevo thread creado en el programa de ejemplo es el siguiente:



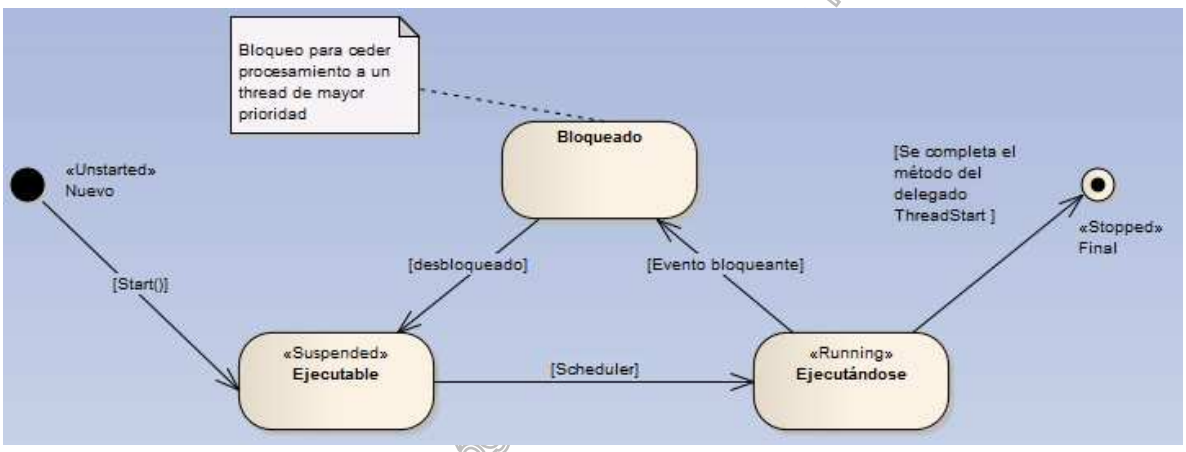
Comienzo de un Thread

Cuando se crea un thread no comienza automáticamente sino que se lo tiene que comenzar explícitamente con el método Start().

El llamado a este método coloca a la CPU definida para el thread en un estado de posible ejecución para ser autorizada a comenzar por el Scheduler en el sistema operativo que se esté ejecutando.

Schedule de un Thread

La situación descrita anteriormente se puede analizar mejor en un diagrama de estados, el cual refleja el cambio de estado del objeto cuando pasa de un estado "a ejecutar" (Runnable) a un estado en ejecución por designación de tiempo de CPU que realiza el Scheduler. El diagrama de estado es el siguiente:



Los estados de un thread

El diagrama de estados anterior deja en claro que un thread puede tener varios estados diferentes (en el diagrama los estados están nombrados como estereotipos) según el Scheduler vaya determinando el cambio de los mismos. El Framework de .Net define una enumeración para todos los posibles estados de un thread llamada ThreadState. Esta define un conjunto de todos los estados de ejecución posibles para los subprocesos. Una vez que se crea un subproceso, este siempre se encuentra en al menos uno de los estados hasta que termina. Los subprocesos creados dentro de Common Language Runtime se encuentran inicialmente en estado Unstarted, mientras que los subprocesos externos que entran en el motor en tiempo de ejecución ya se encuentran en estado Running .Para que un subproceso Unstarted pase al estado Running, hay que llamar a Start. No todas las combinaciones de valores de ThreadState son válidas; por ejemplo, un subproceso no puede encontrarse en estado Aborted y estado Unstarted a la vez.

Nombre de miembro	Descripción

Nombre de miembro	Descripción
Running	El subproceso se ha iniciado, no está bloqueado y no existe una excepción ThreadAbortException pendiente.
StopRequested	Se ha solicitado que el subproceso se detenga. Esto último sólo se refiere al uso interno.
SuspendRequested	Se ha solicitado que el subproceso se suspenda.
Background	El subproceso está ejecutándose como subproceso en segundo plano, por oposición a un subproceso en primer plano. Para controlar este estado, hay que establecer la propiedad Thread.IsBackground.
Unstarted	No se ha invocado al método Thread.Start en el subproceso.
Stopped	El subproceso se ha detenido.
WaitSleepJoin	Subproceso bloqueado. Este podría ser el resultado de llamar a Thread.Sleep o Thread.Join, de solicitar un bloqueo (por ejemplo, llamando a Monitor.Enter o Monitor.Wait) o de esperar en un objeto de sincronización de subprocesos como ManualResetEvent.
Suspended	El subproceso se ha suspendido.
AbortRequested	Se ha invocado al método Thread.Abort en el subproceso, pero el subproceso aún no ha recibido la excepción System.Threading.ThreadAbortException pendiente que intentará finalizarlo.
Aborted	El estado del subproceso incluye AbortRequested y el subproceso está ahora inactivo, pero su estado no ha cambiado todavía a Stopped.

Tiempos de ejecución de un thread

Los threads deben ser pensados en el diseño de un programa como atemporales y asincrónicos. Cada vez que se quiera manejar tiempos con un thread se debe forzar la operación, esto implica invocar métodos especiales para cambiar su funcionamiento atemporal y asíncrono.

Una de estas funciones es Sleep, la cual "duerme" al thread por un período de tiempo igual al que se establece en su argumento medido en milisegundos. El thread entra en un estado de espera de ejecución pero **retiene todos los bloqueos y recursos previos a la ejecución del método**.

El método Sleep es un miembro estático de la clase Thread. El método Thread.Sleep() toma como argumento la cantidad de tiempo (en milisegundos) que se desee detener un thread.

Un ejemplo de esto se puede apreciar en el siguiente código.

Ejemplo

C#

```
using System;
using System.Threading;

namespace dormir
{
```

```
    class Tiempos
    {
        public void EjecutarThread()
        {
            int i;
            i = 0;
            // Pone a dormir el thread 2 segundos
            Thread.Sleep(2000);
            while (true)
            {
                Console.WriteLine("Hola desde EjecutarThread " + i++);
                if (i == 50)
                {
                    break;
                }
            }
        }
    }
}


using System;
using System.Threading;

namespace dormir
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Tiempos r = new Tiempos();
            Thread t = new Thread(new ThreadStart(r.EjecutarThread));
            t.Start();
            while (true)
            {
                Console.WriteLine("Hola desde Main " + i++);
                if (i == 50)
                {
                    break;
                }
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System.Threading

Public Class Tiempos
    Public Sub EjecutarThread()
        Dim i As Integer = 0
        ' Pone a dormir el thread 2 segundos
        Thread.Sleep(2000)
```

```
        While True
            i += 1
            Console.WriteLine("Hola desde EjecutarThread " + i.ToString)
            If i = 50 Then
                Exit While
            End If
        End While
    End Sub
End Class

Imports System.Threading

Module Module1

    Sub Main()
        Dim i As Integer = 0
        Dim r As New Tiempos()
        Dim t As Thread = New Thread(New ThreadStart(AddressOf r.EjecutarThread))
        t.Start()
        While True
            i += 1
            Console.WriteLine("Hola desde Main " + i.ToString)
            If i = 50 Then
                Exit While
            End If
        End While
        Console.ReadKey()
    End Sub
End Module
```

Terminando un Thread

Un thread puede ser forzado a terminar con sólo provocar el fin del método en el que se inició su ejecución. Cualquiera sea la forma en que este método finalice, el thread termina su ejecución. Esto puede ser un beneficio o una desventaja si se accede a variables de instancia que intervengan en el procesamiento del método o cualquier otro tipo de suceso que interrumpa la ejecución del método que comenzó el thread, como una excepción. Por lo tanto, se debe utilizar con precaución.

Ejemplo

C#

```
namespace finalizar
{
    class ThreadConFinalizacion
    {
        private bool terminar = false;

        public void EjecutarThread()
        {
            int i;
            i = 0;
            while (!terminar)
```

```
            {
                Console.WriteLine("Hola desde EjecutarThread " + i++);
            }
        }

        public void FinEjecucion()
        {
            terminar = true;
        }
    }
}

using System.Threading;

namespace finalizar
{
    public class ControladorThread
    {
        private ThreadConFinalizacion r = new ThreadConFinalizacion();
        private Thread t;

        public ControladorThread()
        {
            t = new Thread(new ThreadStart(r.EjecutarThread));
        }

        public void IniciarThread()
        {
            t.Start();
        }
        public void FinalizarThread()
        {
            //Usa la instancia especifica de ThreadConFinalizacion
            r.FinEjecucion();
        }

    }
}

using System;
using System.Threading;

namespace finalizar
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            ControladorThread ct = new ControladorThread();
            ct.IniciarThread();
            while (i < 150) Console.WriteLine("Ciclo en Main: " + i++);
            ct.FinalizarThread();
            Console.ReadKey();
        }
```

```
    }
}
```

VB

```
Imports System.Threading

Public Class ThreadConFinalizacion
    Private terminar As Boolean = False

    Public Sub EjecutarThread()
        Dim i As Integer = 0

        While Not terminar
            i += 1
            Console.WriteLine("Hola desde EjecutarThread " + i.ToString)
        End While
    End Sub

    Public Sub FinEjecucion()
        terminar = True
    End Sub
End Class


Imports System.Threading

Public Class ControladorThread
    Private r As ThreadConFinalizacion = New ThreadConFinalizacion()
    Private t As Thread

    Public Sub New()
        t = New Thread(New ThreadStart(AddressOf r.EjecutarThread))
    End Sub

    Public Sub IniciarThread()
        t.Start()
    End Sub

    Public Sub FinalizarThread()
        'Usa la instancia específica de ThreadConFinalizacion
        r.FinEjecucion()
    End Sub
End Class

Module Module1

    Sub Main()
        Dim i As Integer = 0
        Dim ct As New ControladorThread()
        ct.IniciarThread()

        While i < 150
            i += 1
            Console.WriteLine("Ciclo en Main: " + i.ToString)
        End While
```

```
        ct.FinalizarThread()
        Console.ReadKey()
    End Sub
End Module
```

En este ejemplo, a diferencia del anterior, la finalización de la ejecución del thread está condicionada a la ejecución del método FinalizarThread dentro del código que aquel que lo creó, por lo tanto el ciclo que se ejecuta dentro del método EjecutarThread lo hará tantas veces como se lo permita el Scheduler, ya que este es quien asigna los tiempos de CPU. Cuando se termina el ciclo dentro de Main se invoca a FinalizarThread, que a la vez invoca a FinEjecucion que cambia el valor de la variable booleana que controla el ciclo dentro de EjecutarThread provocando la finalización del método y por lo tanto, del thread.

Control Básico de Threads

Existen varios constructores, propiedades y métodos para el control de threads. Dar una explicación del threads excede los objetivos de la explicación ya que se podría escribir un libro sólo de este tema. Sin embargo, conociendo algunas propiedades y métodos básicos se cubren la mayoría de las necesidades de programación multithreads típica. Entre ellos, algunos de los más importantes son:

Miembro	Descripción
Name	Una propiedad de tipo cadena que se utiliza para obtener o establecer el nombre descriptivo de la instancia del thread.
Priority	Una propiedad del tipo System.Threading.ThreadPriority. Esta propiedad se usa para obtener o establecer el valor que indica la prioridad planificada por el Scheduler para este thread. La prioridad puede ser cualquier instancia de la enumeración ThreadPriority que incluye AboveNormal, BelowNormal, Normal, Highest y Lowest.
CurrentThread	Obtiene el subproceso actualmente en ejecución.
IsAlive	Una propiedad booleana que indica si el thread está vivo (en ejecución) o ha terminado.
ThreadState	Una propiedad del tipo System.Threading.ThreadState. Esta propiedad se utiliza para obtener el valor que contiene el estado del thread. El valor devuelto por esta propiedad es una instancia de la enumeración ThreadState que incluye Aborted, AbortRequested, Suspended, Stopped, Unstarted, Running, WaitSleepJoin, etc.
Start()	Comienza la ejecución del thread.
Abort()	Permite al subproceso actual detener la ejecución de un subproceso de forma permanente. El método lanza la excepción ThreadAbortException en el subproceso de ejecución.
Join()	Hace que el subproceso actual espere a que otro hilo llegue hasta el final.

La prioridad de un thread

El Scheduler determina los threads a ejecutar dependiendo de su prioridad. Cuando un thread tiene mayor prioridad, se ejecuta primero. Si un thread de mayor prioridad necesita tiempo de CPU y en ese momento se está ejecutando uno de menor prioridad, el Scheduler interrumpe la ejecución de este último para otorgarle el tiempo de CPU al de mayor prioridad.

Las prioridades están dividas en un sistema operativo según el lugar al cual pertenece el proceso. En Windows existe una división básica para la asignación de prioridades:

- Threads de usuarios, los cuales además gestionan prioridades para esta clasificación
- Threads del sistema operativo, los cuales tienen un sistema de prioridades al igual que el anterior, pero siempre son de mayor prioridad que los threads de usuario

De esta manera, todo thread del sistema operativo interrumpe siempre la ejecución de un proceso en modo usuario, pero sólo interrumpe los procesos o subprocesos del sistema operativo si tiene mayor prioridad.

Queda por analizar qué pasa cuando dos threads tienen la misma prioridad. En el caso de Windows cuando se da esta situación, el sistema operativo trabaja por "time slicing" lo que significa, más o menos, lapso de tiempo. El concepto es el siguiente: *si dos threads son del mismo tipo y tienen la misma prioridad, el sistema operativo otorga tiempos lo más equitativamente posible a ambos, intercambiando la ejecución entre uno u otro hasta que terminen*.

Por lo tanto, los threads que el programador maneja son siempre en modo usuario y se clasifican según una serie de prioridades cuyos valores típicos se establecen en la enumeración ThreadPriority.

ThreadPriority define el conjunto de todos los valores posibles para la prioridad de un subproceso. Las prioridades de los subproceso especifican la prioridad relativa de un subproceso frente a otro.

Cada subproceso tiene asignada una prioridad. Los subprocesos creados en tiempo de ejecución tienen asignada inicialmente la prioridad Normal, mientras que los subprocesos creados fuera del tiempo de ejecución conservan la prioridad declarada al momento de iniciarlos.

Para obtener y establecer la prioridad de un subproceso, hay que obtener acceso a la propiedad Priority.

La ejecución de los subprocesos se planea en función de su prioridad. El algoritmo de programación utilizado para determinar el orden de ejecución de los subprocesos varía en función de cada sistema operativo y se lo ha mencionado anteriormente como el Scheduler. El sistema operativo también puede ajustar la prioridad del subproceso de forma dinámica, en relación con el desplazamiento del foco de la interfaz de usuario entre segundo plano y primer plano. La razón de esto es que Windows encamina los mensajes que recibe una interfaz gráfica en base a su foco, lo

cual determina que aplicación se considera de “foreground” (al frente) y cual de “background” (por detrás).

La prioridad de un subproceso no afecta al estado del subproceso. Se dice que un thread se “programa” para ejecución cuando el Scheduler determina cunado se ejecutará. Los elementos de la enumeración se muestran a continuación:

Nombre de miembro	Descripción
Lowest	El thread puede programarse después de los subprocesos que tengan cualquier otra prioridad.
BelowNormal	El thread puede programarse después de los subprocesos con prioridad Normal y antes que los subprocesos con prioridad Lowest.
Normal	El thread puede programarse después de los subprocesos con prioridad AboveNormal y antes que los subprocesos con prioridad BelowNormal. Los subprocesos tienen prioridad Normal de forma predeterminada.
AboveNormal	El thread puede programarse después de los subprocesos con prioridad Highest y antes que los subprocesos con prioridad Normal.
Highest	El thread puede programarse antes que los subprocesos que tengan cualquier otra prioridad.

Ejemplo

C#

```
using System;
using System.Threading;

namespace prioridad
{
    public class VerificadorDePrioridades
    {
        bool banderaDelCiclo;

        public VerificadorDePrioridades()
        {
            banderaDelCiclo = true;
        }

        public bool BanderaDelCiclo
        {
            set { banderaDelCiclo = value; }
        }

        public void EjecutarThread()
        {
            long contadorDelThread = 0;

            while (banderaDelCiclo)
```

```
            {
                contadorDelThread++;
            }
            Console.WriteLine();
            Console.WriteLine("El {0} con la prioridad {1,11}" +
                " tiene una cuenta de = {2,13}", Thread.CurrentThread.Name,
                Thread.CurrentThread.Priority.ToString(),
                contadorDelThread.ToString("N0"));
        }
    }
}

using System;
using System.Threading;

namespace prioridad
{
    class Program
    {
        static void Main(string[] args)
        {
            VerificadorDePrioridades verificador = new VerificadorDePrioridades();
            ThreadStart delegadoDelComienzo =
                new ThreadStart(verificador.EjecutarThread);

            Thread threadUno = new Thread(delegadoDelComienzo);
            threadUno.Name = "Thread Uno";
            Thread threadDos = new Thread(delegadoDelComienzo);
            threadDos.Name = "Thread Dos";
            Console.WriteLine("Espere alrededor de 10 segundos por favor.");

            threadDos.Priority = ThreadPriority.BelowNormal;
            threadUno.Start();
            threadDos.Start();

            // Permitir contar por alrededor de 10 segundos.
            Thread.Sleep(10000);
            verificador.BanderaDelCiclo = false;
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System.Threading

Public Class VerificadorDePrioridades

    Dim _banderaDelCiclo As Boolean

    Sub New()
        _banderaDelCiclo = True
    End Sub

    WriteOnly Property BanderaDelCiclo As Boolean
```

```
        Set(value As Boolean)
            _banderaDelCiclo = value
        End Set
    End Property

    Sub EjecutarThread()
        Dim contadorDelThread As Long = 0

        While _banderaDelCiclo
            contadorDelThread += 1
        End While

        Console.WriteLine("El {0} con la prioridad {1,11}" & _
            " tiene una cuenta de = {2,13}", Thread.CurrentThread.Name, _
            Thread.CurrentThread.Priority.ToString(), _
            contadorDelThread.ToString("N0"))
    End Sub
End Class

Imports System.Threading

Module Module1

    Sub Main()
        Dim verificador As New VerificadorDePrioridades()

        Dim threadUno As Thread = _
            New Thread(AddressOf verificador.EjecutarThread)
        threadUno.Name = "Thread Uno"
        Dim threadDos As Thread = _
            New Thread(AddressOf verificador.EjecutarThread)
        threadDos.Name = "Thread Dos"
        Console.WriteLine("Espere alrededor de 10 segundos por favor.")

        threadDos.Priority = ThreadPriority.BelowNormal
        threadUno.Start()
        threadDos.Start()

        ' Permitir contar por alrededor de 10 segundos.
        Thread.Sleep(10000)
        verificador.BanderaDelCiclo = False
        Console.ReadKey()
    End Sub
End Module
```


Nota: los ejemplos como el anterior se presentan con fines didácticos, lo cual implica que el comportamiento de los threads para procesos cortos, que realizan una cuenta acumulada en un contador como el anterior, puede parecer que trabaja diferente a lo explicado. La razón de esto pueden ser muchas causas. Por ejemplo, para citar una de las posibles, en un sistema que tiene múltiples núcleos el sistema operativo trata de usarlos todos y éstos tienen un thread de sincronización independiente del sistema operativo, con lo cual un proceso de menor prioridad parece tener mayor prioridad a la esperada según el núcleo en el cual se ejecuta. Sin embargo, cuando los procesos trabajan a lo largo del tiempo, este comportamiento errático entre

procesadores se compensa y se puede observar el definido teóricamente.

Thread en espera

Existen mecanismos para bloquear la ejecución de un thread de forma temporal. Luego, es posible reanudarla como si nada hubiese ocurrido. El thread da la apariencia de haber ejecutado una instrucción con mucha lentitud.

Método Thread.Sleep()

El método Sleep proporciona una forma de detener un thread durante un periodo de tiempo. Tener presente que el thread no necesariamente reanuda su ejecución en el momento en que finaliza el periodo de espera. La razón es que puede haber otro thread en ejecución en ese momento y no se detendrá a menos que se produzca una de las siguientes situaciones:

- El thread que se reactiva tiene mayor prioridad.
- El thread en ejecución se bloquea por alguna otra razón.

Método Join

El método Join hace que el thread actual espere hasta que finalice el thread en el que se ha realizado la llamada a Join.

Por ejemplo, se puede suponer que hay que generar múltiples threads para hacer un determinado trabajo y continuar con el siguiente paso sólo después de que todos ellos completos. Hay que especificarle al thread que espere. El punto clave es utilizar el método Thread.Join().

Mediante Join se crea una especie de relación "padre – hijo" tal que el padre no puede terminar hasta que todos sus hijos (los método que invocaron a Join) no terminen.

Para comprender el efecto producido, se ejecuta un programa que crea threads desde otro independiente, sin efectuar ninguna unión. Los distintos subprocesos se ejecutan dependiendo de como el Scheduler determine su comienzo sin ninguna restricción particular respecto de cuándo comenzar cada uno.

Nota: las salidas de los programas que se presentan a continuación pueden variar dependiendo tanto de la vez que se corra el programa como de la plataforma en la cual se ejecute. Sin embargo el concepto se mantiene y con simples ajustes sobre la cantidad o tipo de impresiones realizadas se puede verificar igualmente.

Ejemplo

C#

```
using System;
using System.Threading;

namespace uniones
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Thread threadUno = new Thread(new ThreadStart(Metodo1));
            Thread threadDos = new Thread(new ThreadStart(Metodo2));
            threadUno.Start();
            threadDos.Start();
            threadUno.Join();
            Console.WriteLine("Finalizando Main");
            Console.ReadKey();
        }

        public static void Metodo1()
        {
            for (int i = 1; i <= 5; i++)
            {
                Console.WriteLine("El thread 1 escribe: {0}", i);
            }
        }

        public static void Metodo2()
        {
            for (int i = 10; i >= 5; i--)
            {
                Console.WriteLine("El thread 2 escribe: {0}", i);
            }
        }
    }
}
```

VB

```
Imports System.Threading

Module Module1

    Sub Main()
        Dim threadUno As New Thread(New ThreadStart(AddressOf Metodo1))
        Dim threadDos As New Thread(New ThreadStart(AddressOf Metodo2))
        threadUno.Start()
        threadDos.Start()
        threadUno.Join()
        Console.WriteLine("Finalizando Main")
        Console.ReadKey()
    End Sub

    Public Sub Metodo1()
        For i As Integer = 1 To 5
            Console.WriteLine("El thread 1 escribe: {0}", i)
        Next
    End Sub

    Public Sub Metodo2()
        For i As Integer = 1 To 5
            Console.WriteLine("El thread 2 escribe: {0}", i)
        Next
```

```
    End Sub
End Module
```

La salida que se obtiene es:

```
El thread 1 escribe: 1
El thread 1 escribe: 2
El thread 1 escribe: 3
El thread 1 escribe: 4
El thread 1 escribe: 5
El thread 2 escribe: 10
El thread 2 escribe: 9
El thread 2 escribe: 8
El thread 2 escribe: 7
El thread 2 escribe: 6
El thread 2 escribe: 5
Finalizando Main
```

Se puede clarificar más aún como Join genera uniones entre los threads (uno espera a que termine al que se unió) con un ejemplo simple dándole nombres a los threads mediante la propiedad Name y usando también la propiedad CurrentThread para obtener el nombre asignado del thread actual.

Ejemplo

C#

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace actual
{
    class VerificarThreads
    {
        private Queue<String> nombresDeThreads = new Queue<String>();
        private int cantidadThreads = 10;

        public void VerificaThread()
        {
            Thread.CurrentThread.Name = "VerificaThread";

            // Impresión para guiar donde se realiza la salida
            Thread[] threads = new Thread[cantidadThreads];
            Console.WriteLine("Esto se imprime antes");
            Console.WriteLine("Thread actual que crea subprocesos: " +
              Thread.CurrentThread.Name);
          for (int i = 0; i < threads.Length; i++)
            {
                threads[i] = new Thread(new ThreadStart(this.EjecutarThread));
                threads[i].Name = "Thread " + i;
```

```
                threads[i].Start();
              //  threads[i].Join();
            }
            Console.WriteLine("Fin del Thread : " + Thread.CurrentThread.Name);
        }

        public void EjecutarThread()
        {
            nombresDeThreads.Enqueue("Agregado: " + Thread.CurrentThread.Name);
            Console.WriteLine("Fin del Thread " + Thread.CurrentThread.Name);
        }

        public Queue<String> NombresDeThreads
        {
            get
            {
                return nombresDeThreads;
            }
        }
    }
}

using System;
using System.Threading;

namespace actual
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread.CurrentThread.Name = "Main";
            // Impresión para guiar donde se realiza la salida
            Console.WriteLine("Thread actual" + Thread.CurrentThread.Name);
            VerificarThreads test = new VerificarThreads();
            Thread t = new Thread(new ThreadStart(test.VerificaThread));
            t.Start();
           // t.Join();
            if (test.NombresDeThreads.Count == 0)
                Console.WriteLine("No hay nombres en la cola");
            // Impresión de todo los nombres de threads encolados
            foreach (String str in test.NombresDeThreads)
                Console.WriteLine("Nombre de Thread en cola: " +str);
            // Impresión para guiar donde se realiza la salida
            Console.WriteLine("Esto se imprime al final");
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System.Threading

Public Class VerificarThreads
    Private _nombresDeThreads As New List(Of String)()
```

```
    Private cantidadThreads As Integer = 10

    Public Sub VerificaThread()
        Dim threads(cantidadThreads) As Thread
        Thread.CurrentThread.Name = "VerificaThread"
        ' Impresión para guiar donde se realiza la salida
        Console.WriteLine("Esto se imprime antes")
        Console.WriteLine("Thread actual que crea subprocesos: " _
                          + Thread.CurrentThread.Name)
        For i As Integer = 0 To threads.Length - 2
            threads(i) = New Thread(New ThreadStart(AddressOf Me.EjecutarThread))
            threads(i).Name = "Thread " + i.ToString
            threads(i).Start()
            'threads(i).Join()
        Next
        Console.WriteLine("Fin del Thread : " + Thread.CurrentThread.Name)
    End Sub

    Public Sub EjecutarThread()
        _nombresDeThreads.Add(Thread.CurrentThread.Name)
        Console.WriteLine(Thread.CurrentThread.Name)
    End Sub
    Public ReadOnly Property NombresDeThreads() As List(Of String)
        Get
            Return _nombresDeThreads
        End Get
    End Property
End Class

Imports System.Threading

Module Module1

    Sub Main()
        Dim cantidadThreads As Integer = 10
        Thread.CurrentThread.Name = "Main"
        ' Impresión para guiar donde se realiza la salida
        Console.WriteLine("Thread actual " + Thread.CurrentThread.Name)
        Dim test As New VerificarThreads()
        Dim t As Thread = New Thread(New ThreadStart(AddressOf test.VerificaThread))

        t.Start()
        't.Join()
        ' Impresión de todo los nombres de threads encolados
        If test.NombresDeThreads.Count = 0 Then
            Console.WriteLine("No hay nombres en la cola")
        End If

        For Each str As String In test.NombresDeThreads
            Console.WriteLine(str)
        Next
        ' Impresión para guiar donde se realiza la salida
        Console.WriteLine("Esto se imprime al final")
        Console.ReadKey()
    End Sub
```

```
End Module
```

En el código se puede apreciar que no se ejecuta ningún método Join porque fueron marcados con comentarios, tanto en el método Main como en VerificaThread. La salida que se obtiene es:

```
Thread actualMain
No hay nombres en la cola
Esto se imprime al final
Esto se imprime antes
Thread actual que crea subprocesos: VerificaThread
Fin del Thread Thread 0
Fin del Thread Thread 1
Fin del Thread Thread 2
Fin del Thread Thread 3
Fin del Thread Thread 4
Fin del Thread Thread 5
Fin del Thread Thread 6
Fin del Thread Thread 7
Fin del Thread : VerificaThread
Fin del Thread Thread 8
Fin del Thread Thread 9
```

Otro punto importante es que el objeto de tipo Queue está vacío, o sea, ningún nombre de thread se le agregó, eso quiere decir que la salida se produce en main antes que cualquier thread se ejecute y por lo tanto se ejecutan todas las instrucciones. Sin embargo, al ser el proceso principal, no puede terminar hasta que todos los subprocesos creados a partir de él terminen.

Se puede apreciar en las líneas señaladas por la flecha gris (que se produce en Main) como en la flecha negra (que se produce en VerificaThread) que las salidas se producen antes que terminen los subprocesos que crearon. La razón de esto es nuecamente que el Scheduler decide que threads de igual prioridad ejecutará primero y ***no tiene porque ser secuencial respecto de la creación de los mismos***.

A continuación se saca el comentario del Join que se encuentra en Main y se analiza cómo afecta la salida.

Ejemplo

C#

```
using System;
using System.Threading;

namespace actual
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Thread.CurrentThread.Name = "Main";
            // Impresión para guiar donde se realiza la salida
            Console.WriteLine("Thread actual" + Thread.CurrentThread.Name);
            VerificarThreads test = new VerificarThreads();
            Thread t = new Thread(new ThreadStart(test.VerificaThread));
            t.Start();
            t.Join();
            if (test.NombresDeThreads.Count == 0)
                Console.WriteLine("No hay nombres en la cola");
            // Impresión de todo los nombres de threads encolados
            foreach (String str in test.NombresDeThreads)
                Console.WriteLine("Nombre de Thread en cola: " +str);
            // Impresión para guiar donde se realiza la salida
            Console.WriteLine("Esto se imprime al final");
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System.Threading

Module Module1

    Sub Main()
        Dim cantidadThreads As Integer = 10
        Thread.CurrentThread.Name = "Main"
        ' Impresión para guiar donde se realiza la salida
        Console.WriteLine("Thread actual " + Thread.CurrentThread.Name)
        Dim test As New VerificarThreads()
        Dim t As Thread = New Thread(New ThreadStart(AddressOf test.VerificaThread))

        t.Start()
        t.Join()
        ' Impresión de todo los nombres de threads encolados
        If test.NombresDeThreads.Count = 0 Then
            Console.WriteLine("No hay nombres en la cola")
        End If

        For Each str As String In test.NombresDeThreads
            Console.WriteLine(str)
        Next
        ' Impresión para guiar donde se realiza la salida
        Console.WriteLine("Esto se imprime al final")
        Console.ReadKey()
    End Sub
End Module
```


La salida que se obtiene ahora es:

```
Esto se imprime antes
Thread actual que crea subprocesos: VerificaThread
```

```
Fin del Thread Thread 0
Fin del Thread Thread 2
Fin del Thread Thread 1
Fin del Thread Thread 3
Fin del Thread Thread 5
Fin del Thread Thread 4
Fin del Thread Thread 7
Fin del Thread Thread 6
Fin del Thread : VerificaThread
Fin del Thread Thread 8
Fin del Thread Thread 9
Nombre de Thread en cola: Agregado: Thread 0
Nombre de Thread en cola: Agregado: Thread 2
Nombre de Thread en cola: Agregado: Thread 1
Nombre de Thread en cola: Agregado: Thread 3
Nombre de Thread en cola: Agregado: Thread 5
Nombre de Thread en cola: Agregado: Thread 4
Nombre de Thread en cola: Agregado: Thread 7
Nombre de Thread en cola: Agregado: Thread 6
Nombre de Thread en cola: Agregado: Thread 8
Nombre de Thread en cola: Agregado: Thread 9
Esto se imprime al final
```

Lo primero que se destaca es que la impresión ha cambiado mucho. Por ejemplo, no se imprime más la salida

```
No hay nombres en la cola
```

Lo cual significa, cono se corrobora en la misma salida, que la cola contiene elementos, más aún, contiene TODOS los nombres de threads creados. Esto sólo es posible si el thread Main ESPERA a que termine la ejecución del thread VerificaThread, lo cual implica que al menos se debe crear todos los threads, aunque no necesariamente ejecutarlos antes de terminar VerificaThread, como se puede apreciar en la salida marcada por la flecha gris.

Es el momento de corroborar lo dicho. Si esto es verdad, se puede hacer que VerificaThread espere a que se ejecuten todos los threads que crea efectuando una unión con todos ellos. Por lo tanto se saca el comentario del método Join en el ciclo que crea los threads en VerificaThread.

Ejemplo

C#

```
        public void VerificaThread()
        {
            Thread.CurrentThread.Name = "VerificaThread";

            // Impresión para guiar donde se realiza la salida
```

```
            Thread[] threads = new Thread[cantidadThreads];
            Console.WriteLine("Esto se imprime antes");
            Console.WriteLine("Thread actual que crea subprocesos: " +
              Thread.CurrentThread.Name);
          for (int i = 0; i < threads.Length; i++)
            {
                threads[i] = new Thread(new ThreadStart(this.EjecutarThread));
                threads[i].Name = "Thread " + i;
                threads[i].Start();
                threads[i].Join();
            }
            Console.WriteLine("Fin del Thread : " + Thread.CurrentThread.Name);
        }
```

VB

```
Imports System.Threading

Public Class VerificarThreads
    Private _nombresDeThreads As New List(Of String)()
    Private cantidadThreads As Integer = 10

    Public Sub VerificaThread()
        Dim threads(cantidadThreads) As Thread
        Thread.CurrentThread.Name = "VerificaThread"
        ' Impresión para guiar donde se realiza la salida
        Console.WriteLine("Esto se imprime antes")
        Console.WriteLine("Thread actual que crea subprocesos: " _
                          + Thread.CurrentThread.Name)
        For i As Integer = 0 To threads.Length - 2
            threads(i) = New Thread(New ThreadStart(AddressOf Me.EjecutarThread))
            threads(i).Name = "Thread " + i.ToString
            threads(i).Start()
            threads(i).Join()
        Next
        Console.WriteLine("Fin del Thread : " + Thread.CurrentThread.Name)
    End Sub

    Public Sub EjecutarThread()
        _nombresDeThreads.Add(Thread.CurrentThread.Name)
        Console.WriteLine(Thread.CurrentThread.Name)
    End Sub
    Public ReadOnly Property NombresDeThreads() As List(Of String)
        Get
            Return _nombresDeThreads
        End Get
    End Property
End Class
```


La salida producida ahora es:

```
Esto se imprime antes
Thread actual que crea subprocesos: VerificaThread
Fin del Thread Thread 0
Fin del Thread Thread 1
```

```
Fin del Thread Thread 2
Fin del Thread Thread 3
Fin del Thread Thread 4
Fin del Thread Thread 5
Fin del Thread Thread 6
Fin del Thread Thread 7
Fin del Thread Thread 8
Fin del Thread Thread 9
Fin del Thread : VerificaThread
Nombre de Thread en cola: Agregado: Thread 0
Nombre de Thread en cola: Agregado: Thread 1
Nombre de Thread en cola: Agregado: Thread 2
Nombre de Thread en cola: Agregado: Thread 3
Nombre de Thread en cola: Agregado: Thread 4
Nombre de Thread en cola: Agregado: Thread 5
Nombre de Thread en cola: Agregado: Thread 6
Nombre de Thread en cola: Agregado: Thread 7
Nombre de Thread en cola: Agregado: Thread 8
Nombre de Thread en cola: Agregado: Thread 9
Esto se imprime al final
```

Como se puede apreciar en las líneas marcadas por la flecha gris, cada thread sobre el cual se ejecutó un Join espera al método al cual se unió a que este termine.

Puntos importantes a tener en cuenta:

- El Join no asegura secuencialidad, sino que el thread sobre el que se ejecuta el Join espera al que se unió a que termine.
- Lo único que se puede asegurar con un Join es que el subprocesos al que se unió termine totalmente, es decir, si se crean subprocesos a partir de él, estos deben terminar también.
- La única jerarquía que se puede establecer para determinar la finalización de los threads es de padre a hijo, de manera que usando Join todo padre (el que crea el thread) debe esperar obligatoriamente que termine su hijo.
- El método Join sincroniza SOLO LA EJECUCIÓN, nunca los datos en memoria.

Interrupción de ejecución

Se puede interrumpir un subproceso en espera llamando a Thread.Interrupt para aquel que esté bloqueado (por ejemplo, usando el método Thread.Sleep), lo cual produce una excepción del tipo ThreadInterruptedException, que saca al subproceso de la llamada que lo bloquea. En el caso que dicho subproceso no se encuentre bloqueado al momento de llamar a Thread.Interrupt, cuando se produce la primera situación que lo bloquee actuará como si se pidiese la interrupción en ese momento.

El subproceso debe detectar la excepción ThreadInterruptedException y hacer lo que sea necesario para seguir funcionando. Si el subproceso pasa por alto la excepción, el motor en tiempo de ejecución detecta la excepción y detiene el subproceso. Si el subproceso nunca se bloquea, no se produce la excepción y, por lo tanto, el subproceso podría finalizar sin interrumpirse en ningún momento.

El ejemplo a continuación muestra cuando se intenta interrumpir un thread que aún no está bloqueado. Cuando éste intenta bloquearse mediante el método Sleep, se interrumpe dicho bloqueo y se lanza una excepción del tipo ThreadInterruptedException.

Ejemplo

C#

```
using System;
using System.Threading;

namespace interrupciones
{
    class SinDormir
    {
        bool _aDormir = false;

        public bool ADormir
        {
            set { _aDormir = value; }
        }

        public SinDormir() { }

        public void EjecutarThread()
        {
            Console.WriteLine(
              "El thread nuevoThread está ejecutando EjecutarThread().");
            while (!_aDormir)
            {
                // Usar SpinWait en lugar de Sleep para demostrar el
                // efecto de llamar a Interrupt en un thread en ejecución
                // y asegurarse que no se llamó a Sleep todavía.
                Thread.SpinWait(10000000);
            }
            try
            {
                Console.WriteLine("El thread nuevoThread se va a dormir.");

                // Cuando nuevoThread se va a dormir es despertado
                // inmediatamente por una ThreadInterruptedException.
                Thread.Sleep(Timeout.Infinite);
            }
            catch (ThreadInterruptedException e)
            {
                Console.WriteLine("El thread nuevoThread no puede dormir - " +
```

```
                    "fue interrumpido por el thread de Main.");
                Console.WriteLine("El mensaje de excepción es: {0}", e.Message);
            }
        }
    }
}

using System;
using System.Threading;

namespace interrupciones
{
    class Program
    {
        static void Main(string[] args)
        {
            SinDormir despierto = new SinDormir();
            Thread nuevoThread =
                new Thread(new ThreadStart(despierto.EjecutarThread));
            nuevoThread.Start();

            // La siguiente línea causa que se lanze una excepción
            // en EjecutarThread si está actualmente bloqueado o se
            // bloquea en el futuro
            nuevoThread.Interrupt();
            Console.WriteLine(
              "El thread de Main llama a Interrupt en nuevoThread.");

            // Indicarle a nuevoThread que duerma.
            despierto.ADormir = true;

            // Esperar que termine nuevoThread.
            nuevoThread.Join();
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System.Threading

Public Class SinDormir

    Dim _aDormir As Boolean = False

    WriteOnly Property ADormir As Boolean
        Set(value As Boolean)
            _aDormir = value
        End Set
    End Property

    Sub New()
    End Sub

    Sub EjecutarThread()
```

```
        Console.WriteLine("El thread nuevoThread está ejecutando EjecutarThread().")
        While Not _aDormir

            ' Usar SpinWait en lugar de Sleep para demostrar el
            ' efecto de llamar a Interrupt en un thread en ejecución
            ' y asegurarse que no se llamó a Sleep todavía.
            Thread.SpinWait(10000000)
        End While
        Try
            Console.WriteLine("El thread nuevoThread se va a dormir.")

            ' Cuando nuevoThread se va a dormir es despertado
            ' inmediatamente por una ThreadInterruptedException.
Thread.Sleep(Timeout.Infinite)
        Catch e As ThreadInterruptedException
            Console.WriteLine("El thread nuevoThread no puede dormir - " +
                 "fue interrumpido por el thread de Main.")
            Console.WriteLine("El mensaje de excepción es: {0}", e.Message)
        End Try
    End Sub

End Class

Imports System.Threading

Module Module1

    Sub Main()
        Dim despierto As New SinDormir()
        Dim nuevoThread As New Thread(AddressOf despierto.EjecutarThread)
        nuevoThread.Start()

        ' La siguiente línea causa que se lanze una excepción
        ' en EjecutarThread si esta actualmente bloqueado o se
        ' bloquea en el futuro       nuevoThread.Interrupt()
        Console.WriteLine("El thread de Main llama a Interrupt en nuevoThread.")

        ' Indicarle a nuevoThread que duerma.
        despierto.ADormir = True

        ' Esperar que termine nuevoThread.
        nuevoThread.Join()
        Console.ReadKey()
    End Sub
End Module
```


La salida del programa es la siguiente:

```
El thread de Main llama a Interrupt en nuevoThread.
El thread nuevoThread está ejecutando EjecutarThread().
El thread nuevoThread se va a dormir.
El thread nuevoThread no puede dormir - fue interrumpido por el thread de Main.
El mensaje de excepción es: Se interrumpió el estado de espera del subproceso.
```

Usando la Palabra Clave lock o SyncLock (C# o VB)

Todavía queda por tratar el tema de la memoria compartida. Existen situaciones en las cuales dos o más threads compiten por acceder a un espacio de memoria en común. Cuando esta situación puede provocar inconvenientes en el procesamiento normal de cada uno de ellos, se debe bloquear la memoria para que ninguno cause inconvenientes al otro.

Para exponer el problema, se plantea la siguiente situación en la cual existe una pila manejada como un vector y se utiliza una variable de instancia para controlar el elemento a acceder dentro de él. Si esta variable es incrementada o decrementada en los métodos de extracción o escritura más allá del valor esperado, puede ocasionar que la pila quede inconsistente.

Ejemplo

C#

```
using System;
using System.Threading;

namespace pilaDesincronizada
{
    class PilaSinSincronizar
    {
        private int indice = 0;
        private char[] datos = new char[20];

        public void Poner(char d)
        {
            datos[indice] = d;
            indice++;
            Console.WriteLine("Poner\t{0}\t Índice: {1}",
              Thread.CurrentThread.Name, indice);
        }

        public char Sacar()
        {
            indice--;
            Console.WriteLine("Sacar\t{0}\tValor: {1}\tÍndice: {2}",
              Thread.CurrentThread.Name, datos[indice], indice);
            return datos[indice];
        }
    }
}
```

VB

```
Imports System.Threading

Public Class PilaSinSincronizar
    Private indice As Integer = 0
    Private datos(20) As Char

    Public Sub Poner(d As Char)
        datos(indice) = d
```

```
        indice += 1
        Console.WriteLine("Poner" + vbTab + "{0}" _
                          + vbTab + "Índice: {1}", _
                          Thread.CurrentThread.Name, indice)
    End Sub

    Public Function Sacar() As Char
        indice -= 1
        Console.WriteLine("Sacar" + vbTab + "{0}" _
                          + vbTab + "Valor: {1}" + vbTab + _
                          "Índice: {2}", Thread.CurrentThread.Name, _
                          datos(indice), indice)
        Return datos(indice)
    End Function
End Class
```

Puede suceder que para un mismo objeto del tipo PilaSinSincronizar se invoque en threads independientes los métodos Poner y Sacar. Esto puede ocasionar que se incremente (o decremente) el valor del índice más allá de lo que debería hacerse.

Esto se puede ver al ejecutar el siguiente programa que utiliza la clase PilaSinSincronizar.

Ejemplo

C#

```
using System;
using System.Threading;

namespace pilaDesincronizada
{
    class Program
    {
        private static PilaSinSincronizar pss = new PilaSinSincronizar();

        static void Main(string[] args)
        {
            Thread[] threadsDesincronizadosPoner = new Thread[6];
            for (int i = 0; i < threadsDesincronizadosPoner.Length; i++)
            {
                threadsDesincronizadosPoner[i] = new Thread(
                    new ThreadStart(PonerDatosDesincronizados));
                threadsDesincronizadosPoner[i].Name = "Thread " + i;
                threadsDesincronizadosPoner[i].Start();
 //                 threadsDesincronizadosPoner[i].Join();
            }

            Thread[] threadsDesincronizadosSacar = new Thread[6];
            for (int i = 0; i < threadsDesincronizadosSacar.Length; i++)
            {
                threadsDesincronizadosSacar[i] = new Thread(
                    new ThreadStart(SacarDatosDesincronizados));
                threadsDesincronizadosSacar[i].Name = "Thread " + i;
                threadsDesincronizadosSacar[i].Start();
```

```
//                threadsDesincronizadosSacar[i].Join();
            }

            Console.ReadKey();
        }

        public static void PonerDatosDesincronizados()
        {
            char c;
            Random r = new Random();

            for (int i = 0; i < 3; i++)
            {
                c = (char)(r.Next(0, 26) + 'a');
                Console.WriteLine("El {0} intenta poner el valor {1}",
                    Thread.CurrentThread.Name, c);
                pss.Poner(c);
            }
        }

        public static void SacarDatosDesincronizados()
        {
            //Thread.Sleep(500);
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("El {0} saca el valor {1}",
                    Thread.CurrentThread.Name, pss.Sacar());
            }
        }

    }
}
```

VB

```
Imports System.Threading

Module Module1
    Private pss As PilaSinSincronizar = New PilaSinSincronizar()

    Sub Main()
        Dim threadsDesincronizadosPoner(6) As Thread

        For i As Integer = 0 To threadsDesincronizadosPoner.Length - 2
            threadsDesincronizadosPoner(i) = New Thread( _
              New ThreadStart(AddressOf PonerDatosDesincronizados))
            threadsDesincronizadosPoner(i).Name = "Thread " + i.ToString
            threadsDesincronizadosPoner(i).Start()
            'threadsDesincronizadosPoner(i).Join()
        Next

        Dim threadsDesincronizadosSacar(6) As Thread
        For i As Integer = 0 To threadsDesincronizadosSacar.Length - 2
            threadsDesincronizadosSacar(i) = New Thread( _
              New ThreadStart(AddressOf SacarDatosDesincronizados))
            threadsDesincronizadosSacar(i).Name = "Thread " + i.ToString
```

```
            threadsDesincronizadosSacar(i).Start()
            'threadsDesincronizadosSacar(i).Join()
        Next

        Console.ReadKey()
    End Sub

    Public Sub PonerDatosDesincronizados()
        Dim c As Char
        Dim r As New Random()

        For i As Integer = 0 To 2
            c = Convert.ToChar(Convert.ToInt32(Int(r.Next(0, 26)) + 97))
            Console.WriteLine("El {0} intenta poner el valor {1}", _
              Thread.CurrentThread.Name, i)
            pss.Poner(c)
        Next
    End Sub

    Public Sub SacarDatosDesincronizados()
        'Thread.Sleep(500)
        For i As Integer = 0 To 2
            Console.WriteLine("El {0} saca el valor {1}", _
              Thread.CurrentThread.Name, pss.Sacar())
        Next
    End Sub
End Module
```

Una posible salida al ejecutar el programa es la siguiente:

```
El Thread 0 intenta poner el valor m
El Thread 1 intenta poner el valor m
El Thread 2 intenta poner el valor m
Poner   Thread 2         Índice: 3
El Thread 2 intenta poner el valor c
Poner   Thread 2         Índice: 4
El Thread 2 intenta poner el valor g
Poner   Thread 2         Índice: 5
Poner   Thread 1         Índice: 2
El Thread 1 intenta poner el valor c
Poner   Thread 1         Índice: 6
El Thread 1 intenta poner el valor g
Poner   Thread 1         Índice: 7
El Thread 5 intenta poner el valor v
Poner   Thread 5         Índice: 8
El Thread 5 intenta poner el valor u
Poner   Thread 5         Índice: 9
El Thread 5 intenta poner el valor l
Poner   Thread 5         Índice: 10
El Thread 4 intenta poner el valor v
```

```
Poner   Thread 4         Índice: 11
El Thread 4 intenta poner el valor u
Poner   Thread 4         Índice: 12
El Thread 4 intenta poner el valor l
Poner   Thread 4         Índice: 13
El Thread 3 intenta poner el valor v
Poner   Thread 3         Índice: 14
El Thread 3 intenta poner el valor u
Poner   Thread 3         Índice: 15
El Thread 3 intenta poner el valor l
Poner   Thread 3         Índice: 16
Poner   Thread 0         Índice: 1
El Thread 0 intenta poner el valor c
Poner   Thread 0         Índice: 17
El Thread 0 intenta poner el valor g
Poner   Thread 0         Índice: 18
Sacar   Thread 0        Valor: g        Índice: 17
El Thread 0 saca el valor g
Sacar   Thread 0        Valor: c        Índice: 16
El Thread 0 saca el valor c
Sacar   Thread 0        Valor: l        Índice: 15
El Thread 0 saca el valor u
Sacar   Thread 1        Valor: u        Índice: 14
El Thread 1 saca el valor u
Sacar   Thread 1        Valor: v        Índice: 13
El Thread 1 saca el valor v
Sacar   Thread 1        Valor: u        Índice: 11
El Thread 1 saca el valor u
Sacar   Thread 3        Valor: l        Índice: 12
El Thread 3 saca el valor u
Sacar   Thread 3        Valor: v        Índice: 10
El Thread 3 saca el valor v
Sacar   Thread 3        Valor: l        Índice: 9
El Thread 3 saca el valor l
Sacar   Thread 4        Valor: u        Índice: 8
El Thread 4 saca el valor v
Sacar   Thread 4        Valor: g        Índice: 6
El Thread 4 saca el valor g
Sacar   Thread 4        Valor: c        Índice: 5
El Thread 4 saca el valor c
Sacar   Thread 2        Valor: v        Índice: 7
El Thread 2 saca el valor c
Sacar   Thread 2        Valor: c        Índice: 3
Sacar   Thread 5        Valor: g        Índice: 4
El Thread 5 saca el valor c
```

```
Sacar   Thread 5        Valor: m        Índice: 2
El Thread 5 saca el valor m
Sacar   Thread 5        Valor: m        Índice: 1
El Thread 5 saca el valor m
El Thread 2 saca el valor c
Sacar   Thread 2        Valor: m        Índice: 0
El Thread 2 saca el valor m
```

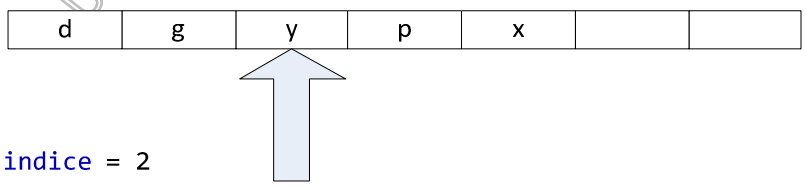
En un primer intento de ordenar la ejecución, se puede intentar poner a dormir los thread que van a sacar valores de la pila. Sin embargo, esto no garantiza que el valor del índice utilizado varíe según el thread que lo accede solamente, ya el método Sleep trabaja sobre el tiempo de ejecución y no sobre la memoria compartida. El razonamiento es análogo si se utiliza Join (se pueden comparar las salidas simplemente sacando los comentarios del programa), lo cual nuevamente esquiva el problema real el cuál es que un thread no debe poder acceder a la memoria que accede otro thread cuando este se está ejecutando. Esto se conoce en programación como ***problema de concurrencia de threads***.

Se necesita colocar una "bandera" para que un thread indique a cualquier otro que está trabajando sobre la variable que mientras el acceda a ella, ningún otro podrá hacerlo.

El comportamiento de este código exige que el valor de índice contenga el subíndice de vector de la siguiente celda vacía de la pila. El uso del mecanismo de decremento e incremento de la variable utilizada como índice genera esta información.

Suponiendo ahora que dos threads contienen una referencia a **una sola instancia** de esta clase. Uno de los threads introduce datos en la pila y el otro, de forma más o menos independiente, retira datos de la pila. En principio, los datos se agregan y suprimen correctamente. No obstante, existe un problema potencial.

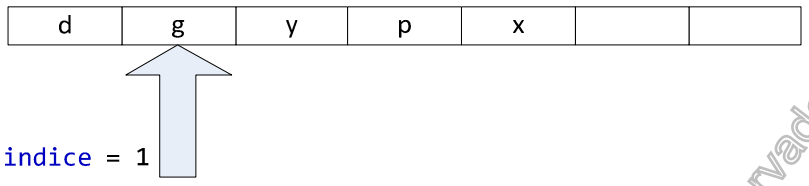
Si el thread "Thread 0" está agregando caracteres y el thread "Thread 1" los está suprimiendo y el thread "Thread 0" acaba de escribir un carácter, pero aún no ha incrementado el contador del índice. Entonces, por alguna razón, dicho thread es sustituido por otro que se apropia del derecho de ejecución. En este momento, el modelo de datos representado por el objeto deja de ser coherente.



Para mantener la coherencia, es necesario que sea indice = 3 o que el carácter no se haya agregado aún.

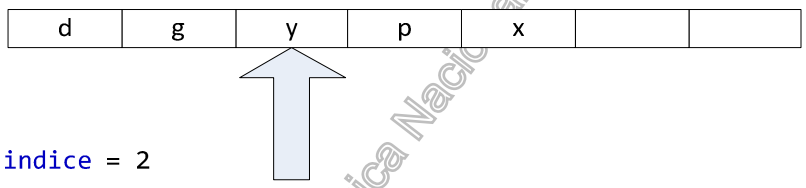
Si el “Thread 0” reanuda la ejecución, puede que no se produzcan errores, pero suponiendo que el “Thread 1” estaba esperando a suprimir un carácter mientras que el “Thread 0” está esperando a tener otra oportunidad de ejecutarse, “Thread 1” tiene la ocasión de suprimir un carácter.

Se produce una situación de falta de coherencia de los datos de entrada del método Poner y, sin embargo, dicho método procede a reducir el valor del índice.



En la práctica, esto hace que no se tenga en cuenta el carácter **y**. A continuación, devuelve el carácter **g**. Hasta ahora, el comportamiento ha sido como si la letra **y** no se hubiese agregado, por lo que es difícil detectar que hay un problema. Pero observar lo que ocurre cuando el thread original, “Thread 0”, reanuda la ejecución.

El “Thread 0” reinicia la ejecución en el punto donde la dejó, en el método Sacar, y procede a incrementar el valor del índice. Ahora tenemos lo siguiente:



Esta configuración implica que el carácter **g** es válido y que la celda que contiene la **y** es la siguiente celda vacía. En otras palabras, **g** se lee como si se hubiese colocado dos veces en la pila y la letra **y** no aparece nunca.

Esto es un ejemplo de un problema general que surge cuando hay varios threads que comparten los datos a los que acceden. Es preciso disponer de un mecanismo que garantice que esos datos compartidos mantengan la integridad antes de que cualquiera de los threads empiece a utilizarlos para una determinada tarea.

Una forma de hacerlo sería impedir que el thread “Thread 0” interrumpa su ejecución hasta que finalice la sección crítica de código. Este planteamiento es habitual en programación en código máquina de bajo nivel, pero suele ser inadecuado en sistemas multiusuario.

Otro enfoque, que es el utilizado por la tecnología de .Net, es proporcionar un mecanismo para tratar los datos con delicadeza. Este enfoque permite que un thread se trate atomizado y sea capaz de acceder a los datos con independencia de que lo interrumpan mientras está realizando el acceso.

El área de memoria bloqueada se la denomina “área crítica” y el bloqueo le indica a cualquier subproceso que dicha parte de la memoria no puede ser accedida hasta que el subproceso que retiene el bloqueo no la libere. Si durante la espera que mantiene un thread a la liberación del recurso se ejecuta el método Interrupt() sobre el mismo, se lanza una ThreadInterruptedException en el subproceso que está esperando para escribir en el área crítica.

Las áreas crítica de memoria se definen con la sentencia `lock` o `SyncLock` (C# o VB). La palabra clave `lock` o `SyncLock` (C# o VB) marca un bloque de instrucciones como un área crítica, para lo cual utiliza el bloqueo de exclusión mutua de un objeto, la ejecución de una instrucción y, posteriormente, la liberación del bloqueo. Se denomina “exclusión mutua de un objeto” cuando dos o más procesos no pueden acceder a los elementos de memoria que maneja dicha objeto. Los elementos de memoria están dados por los atributos manejados en el heap de dicho objeto. Cuando un subproceso bloquea un objeto, sólo él puede modificar los valores en las variables de instancia. Los demás threads deberán esperar a poder adquirir el bloqueo para realizar alguna operación sobre dicho objeto.

En general se debe evitar bloquear memoria definida con el modificador de acceso `public`. Las construcciones comunes `lock(this)` o `SyncLock Me` (C# o VB), `lock (typeof (MyType))` o `SyncLock Typeof (MyType)` (C# o VB) y `lock ("myLock")` o `SyncLock "myLock"` (C# o VB) incumplen esta recomendación en las siguientes situaciones:

- `lock(this)` o `SyncLock(Me)` (C# o VB), se convierte en un problema si la instancia es accesible públicamente (modificador de acceso `public`).
- `lock (typeof (MyType))` o `SyncLock Typeof (MyType)` (C# o VB) se convierte en un problema si `MyType` es accesible públicamente.
- `lock ("myLock")` o `SyncLock "myLock"` (C# o VB) se convierte en un problema puesto que cualquier otro código del proceso que utilice la misma cadena compartirá el mismo bloqueo.

El procedimiento recomendado es definir un objeto un atributo en un objeto como `private` o `Private` (C# o VB), para realizar el bloqueo o una variable que maneje una referencia a un objeto que sea `private static` o `Private Shared` (C# o VB), si se quiere acceder de diferentes instancias y se debe proteger los datos comunes a todas las instancias.

Para ejemplificar lo explicado, se puede suponer la siguiente situación:

- Existe una cuenta en la cual se pueden realizar retiros.
- Cada retiro se ejecuta en una operación independiente
- Todas las operaciones convergen al mismo objeto de tipo cuenta que almacena el balance de la cuenta.
- Si la transacción a realizar supero el monto disponible, se debe rechazar la transacción retornando un valor 0 en esa operación de retiro.
- Se debe finalizar las acciones una vez que se acaba el dinero a retirar (o lo que es igual a tener balance 0).

Analizando el problema, se puede deducir fácilmente que es necesario tener un objeto que sea del tipo de una abstracción llamada Cuenta, que tenga un atributo que determine el balance y un servicio que permita realizar retiros, de manera que los retiros se ejecuten cada uno en un thread independiente. El siguiente ejemplo muestra una posible resolución emulando los valores a retirar con un número aleatorio generado.

Ejemplo

C#

```
using System;

namespace bloqueos
{
    class Cuenta
    {
        private int balance;

        Random r = new Random();

        public Cuenta(int inicial)
        {
            balance = inicial;
        }

        int Retiro(int cantidad)
        {

            // Esta condición es siempre falsa salvo que
            // se comente la instrucción de bloqueo
            if (balance < 0)
            {
                throw new Exception("Balance Negativo");
            }

            // Si se comenta la próxima línea se puede
            // apreciar lo ocurrido cuando no se bloquea
            lock (this)
            {
                if (balance >= cantidad)
                {
                    Console.WriteLine("Balance antes de retirar:  " + balance);
                    Console.WriteLine("Cantidad a retirar      : -" + cantidad);
                    balance = balance - cantidad;
                    Console.WriteLine("Balance luego del retiro: " + balance);
                    return cantidad;
                }
                else
                {
                    return 0; // transacción rechazada
                }
            }
        }
```

```
        public void HacerTransacciones()
        {
            int aux = 0;
            int valor = 0;
            for (int i = 0; i < 50 && balance > 0; i++)
            {
                valor = r.Next(1, 100);
                aux = Retiro(valor);
                if (aux == 0)
                {
                    Console.WriteLine("Transacción rechazada para {0}", valor);
                }
            }
        }
    }
}

using System;
using System.Threading;

namespace bloqueos
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread[] threads = new Thread[10];
            Cuenta cuenta = new Cuenta(1000);
            for (int i = 0; i < 10; i++)
            {
                Thread t = new Thread(new ThreadStart(cuenta.HacerTransacciones));
                threads[i] = t;
            }
            for (int i = 0; i < 10; i++)
            {
                threads[i].Start();
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Cuenta
    Private balance As Integer
    Private r As New Random

    Public Sub New(inicial As Integer)
        balance = inicial
    End Sub

    Public Function Retiro(cantidad As Integer)
        ' Esta condición es siempre falsa salvo que
        ' se comente la instrucción de bloqueo
```

```
        If balance < 0 Then
            Throw New Exception("Balance Negativo")
        End If

        ' Si se comenta la próxima línea se puede
        ' apreciar lo ocurrido cuando no se bloquea
        SyncLock (Me)
            If balance >= cantidad Then
                Console.WriteLine("Balance antes de retirar:  " _
                                  + balance.ToString)
                Console.WriteLine("Cantidad a retirar      : -" _
                                  + cantidad.ToString)
                balance = balance - cantidad
                Console.WriteLine("Balance luego del retiro: " _
                                  + balance.ToString)
                Return cantidad
            Else
                Return 0 ' transacción rechazada
            End If
        End SyncLock
    End Function

    Public Sub HacerTransacciones()
        Dim aux As Integer = 0
        Dim valor As Integer = 0

        If balance > 0 Then
            For i As Integer = 0 To 48
                valor = r.Next(1, 100)
                aux = Retiro(valor)
                If aux = 0 Then
                    Console.WriteLine("Transacción rechazada para {0}", valor)
                End If
            Next
        End If

    End Sub
End Class

Imports System.Threading

Module Module1

    Sub Main()
        Dim threads(10) As Thread
        Dim cuenta As New Cuenta(1000)
        For i As Integer = 0 To 8
            Dim t As Thread = _
              New Thread(New ThreadStart(AddressOf cuenta.HacerTransacciones))
            threads(i) = t
        Next
        For i As Integer = 0 To 8
            threads(i).Start()
        Next
        Console.ReadKey()
```

```
    End Sub
End Module
```

Una *posible* salida parcial del programa es la que se muestra a continuación (por cuestiones de espacio y legibilidad se pone parcialmente la salida). Notar que el programa nunca descuenta de manera incorrecta y el balance siempre es consistente debido a que el área crítica de código está encerrada por la instrucción `lock` o `SyncLock` (C# o VB).

```
Balance antes de retirar:  1000
Cantidad a retirar      : -75
Balance luego del retiro: 925
Balance antes de retirar:  925
Cantidad a retirar      : -76
Balance luego del retiro: 849
Balance antes de retirar:  849
Cantidad a retirar      : -98
Balance luego del retiro: 751
Balance antes de retirar:  751
Cantidad a retirar      : -9
Balance luego del retiro: 742
Balance antes de retirar:  742
Cantidad a retirar      : -96
Balance luego del retiro: 646
Balance antes de retirar:  646
Cantidad a retirar      : -6
Balance luego del retiro: 640
Balance antes de retirar:  640
Cantidad a retirar      : -22
Balance luego del retiro: 618
Balance antes de retirar:  618
Cantidad a retirar      : -7
Balance luego del retiro: 611
Balance antes de retirar:  611
Cantidad a retirar      : -17
Balance luego del retiro: 594
Balance antes de retirar:  594
Cantidad a retirar      : -44
Balance luego del retiro: 550
Balance antes de retirar:  550
Cantidad a retirar      : -64
Balance luego del retiro: 486
Balance antes de retirar:  486
Cantidad a retirar      : -20
Balance luego del retiro: 466
Balance antes de retirar:  466
Cantidad a retirar      : -73
Balance luego del retiro: 393
Balance antes de retirar:  393
Cantidad a retirar      : -14
Balance luego del retiro: 379
Balance antes de retirar:  379
Cantidad a retirar      : -6
```

```
Balance luego del retiro: 373
Balance antes de retirar:  373
Cantidad a retirar      : -26
Balance luego del retiro: 347
Balance antes de retirar:  347
Cantidad a retirar      : -61
Balance luego del retiro: 286
Balance antes de retirar:  286
Cantidad a retirar      : -64
Balance luego del retiro: 222
Balance antes de retirar:  222
Cantidad a retirar      : -88
Balance luego del retiro: 134
Balance antes de retirar:  134
Cantidad a retirar      : -54
Balance luego del retiro: 80
Balance antes de retirar:  80
Cantidad a retirar      : -51
Balance luego del retiro: 29
Transacción rechazada para 45
Transacción rechazada para 78
Transacción rechazada para 38
Transacción rechazada para 75
Balance antes de retirar:  29
Cantidad a retirar      : -4
```

La Bandera para el Bloqueo de Objetos

El Framework de .Net maneja las situaciones de memoria compartida a través de banderas de bloqueo internas, las cuales sirven para que cualquier thread que quiera acceder a dicha variable de instancia deba consultar si la misma es accesible o está **bloqueada**.

Estas situaciones se dan sólo en programación multithreading y para manejarla el lenguaje incorpora la palabra reservada synchronized, la cual permite la interacción con la bandera de bloqueo.

Ejemplo

C#

```
using System.Text;
using System.Threading;

namespace pilaSincronizada
{
    class PilaSincronizada
    {
        private int indice = 0;
        private int[] datos = new int[20];

        public void Poner(int d)
        {
```

```
            lock (this)
            {
                datos[indice] = d;
                Console.WriteLine("Poner\t{0}\t Índice: {1}",
                    Thread.CurrentThread.Name, indice);
                indice++;
          }
        }

        public int Sacar()
        {
            lock (this)
            {
                indice--;
                Console.WriteLine("Sacar\t{0}\tValor: {1}\tÍndice: {2}",
                    Thread.CurrentThread.Name, datos[indice], indice);
                return datos[indice];
            }
        }
    }
}
```

VB

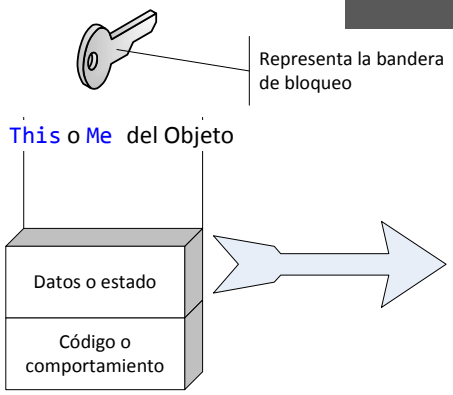
```
Imports System.Threading

Public Class PilaSincronizada
    Private indice As Integer = 0
    Private datos(20) As Integer

    Public Sub Poner(d As Integer)
        SyncLock Me
            datos(indice) = d
            Console.WriteLine("Poner" + vbTab + "{0}" _
                              + vbTab + "Índice: {1}", _
                              Thread.CurrentThread.Name, indice)
            indice += 1
        End SyncLock
    End Sub

    Public Function Sacar() As Integer
        SyncLock Me
            indice -= 1
            Console.WriteLine("Sacar" + vbTab + "{0}" _
                              + vbTab + "Valor: {1}" + vbTab + _
                              "Índice: {2}", Thread.CurrentThread.Name, _
                              datos(indice), indice)
            Return datos(indice)
        End SyncLock
    End Function
End Class
```

El siguiente gráfico muestra dicha situación:

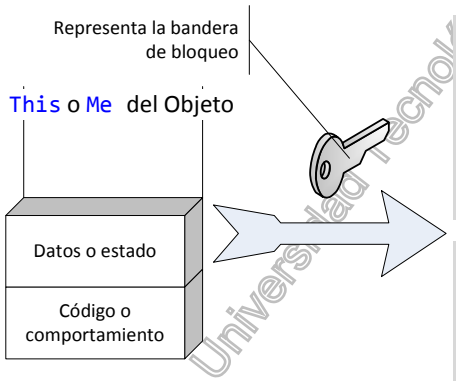


Thread antes de `lock(this)` o `SyncLock Me`

```
lock (this)
{
       datos[indice] = d;
       Console.WriteLine("Poner\t{0}\t Índice: {1}",
              Thread.CurrentThread.Name, indice);
       indice++;
}
```

```
SyncLock Me
       datos(indice) = d
       Console.WriteLine("Poner" + vbTab + "{0}" _
              + vbTab + "Índice: {1}", _
              Thread.CurrentThread.Name, indice)
       indice += 1
End SyncLock
```

El bloque de código asociado a `lock` o `SyncLock` (C# o VB) indica cuando el thread deberá consultar si la variable que pertenece a la memoria compartida puede ser accedida y, en caso de ser así, tendrá que bloquearla antes de ejecutar el código en el bloque. Esto le permite interactuar libremente con la variable con la seguridad que ningún otro thread modificará el valor en ella en tanto no se termine la ejecución del código definido en su bloque de sentencias. Para ello, cuando ingresa al bloque definido por `lock(this)` o `SyncLock Me` (C# o VB), intenta adquirir un bloqueo sobre las variables de instancia utilizadas dentro del bloque de sentencias asociado.



Thread después de `lock(this)` o `SyncLock Me`

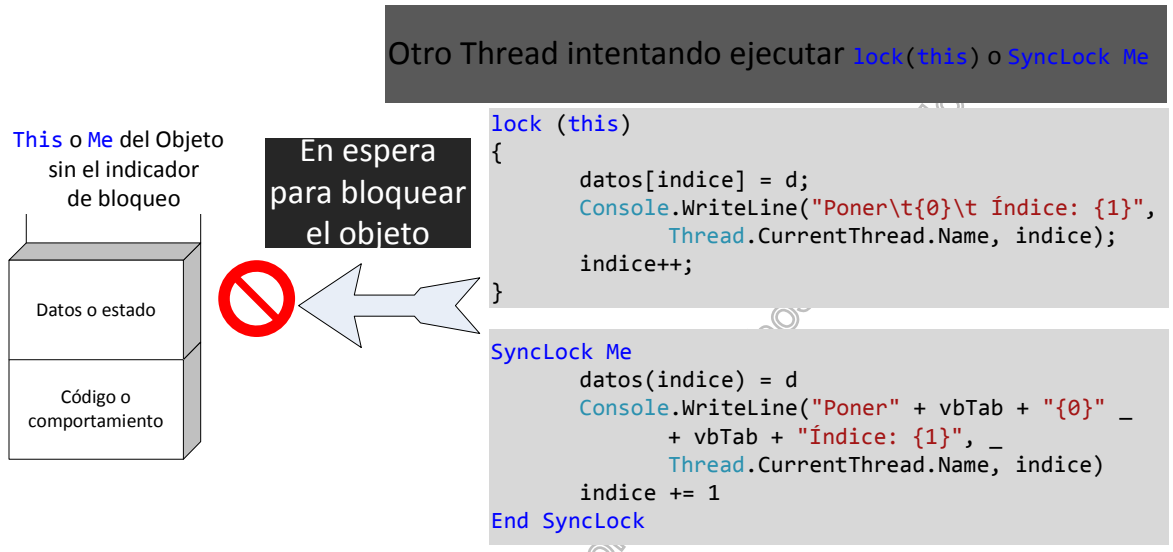
```
lock (this)
{
       datos[indice] = d;
       Console.WriteLine("Poner\t{0}\t Índice: {1}",
              Thread.CurrentThread.Name, indice);
       indice++;
}
```

```
SyncLock Me
       datos(indice) = d
       Console.WriteLine("Poner" + vbTab + "{0}" _
              + vbTab + "Índice: {1}", _
              Thread.CurrentThread.Name, indice)
       indice += 1
End SyncLock
```

Es importante tener en cuenta que esta acción se protege los datos. Si el método Sacar del objeto que posee los datos compartidos no está protegido por `lock(this)` o `SyncLock(Me)` (C# o VB) y otro thread hace una llamada a sacar, sigue existiendo el riesgo de perder la coherencia de los

datos. Todos los métodos que acceden a los datos deben sincronizarse con respecto al mismo indicador de bloqueo para que éste sea efectivo.

En la siguiente figura se observa lo que ocurre si el método Sacar está protegido y otro thread intenta ejecutar el método Sacar del objeto mientras el thread original retiene el indicador de bloqueo del objeto sincronizado.



Cuando el thread quiere ejecutar la primera sentencia dentro del bloque asociado a `lock(this)` o `SyncLock(Me)` (C# o VB), trata de conseguir el indicador de bloqueo que tiene el objeto actualmente en ejecución cuya referencia se encuentra en `this` o `Me` (C# o VB). Como no lo consigue, no puede continuar la ejecución. Seguidamente, el thread se une a un grupo de éstos, que se encuentran en las mismas condiciones en estado de espera, que están asociados al indicador de bloqueo de dicho objeto. Cuando el indicador retorna y queda disponible en el objeto, se entrega el bloqueo a uno de los threads que estén esperándolo y es ese thread quien continúa con la ejecución.

Liberación del indicador de bloqueo

Si un thread está esperando el indicador de bloqueo de un objeto, no puede reanudar su ejecución hasta que el indicador esté disponible. Por lo tanto, es importante que el thread que retiene el bloqueo devuelva el indicador cuando deja de necesitarlo.

En ese momento, el objeto que tiene la memoria compartida recobra el indicador de forma automática. Cuando el thread que retiene el bloqueo pasa el final del bloque de código de la sentencia `lock` o `SyncLock` (C# o VB) con la que se obtuvo el bloqueo, éste se libera. El Framework de .Net garantiza que el bloqueo siempre se devuelva de forma automática, incluso aunque sea lanzada una excepción, se ejecute una sentencia de interrupción o una sentencia de retorno que transfieran la ejecución del código a otra parte situada fuera de un bloque sincronizado. Asimismo,

si un thread ejecuta bloques anidados de código que están sincronizados con respecto al mismo objeto, el indicador de ese objeto se libera correctamente al salir del bloque externo de la secuencia y se hace caso omiso del primer bloque interno.

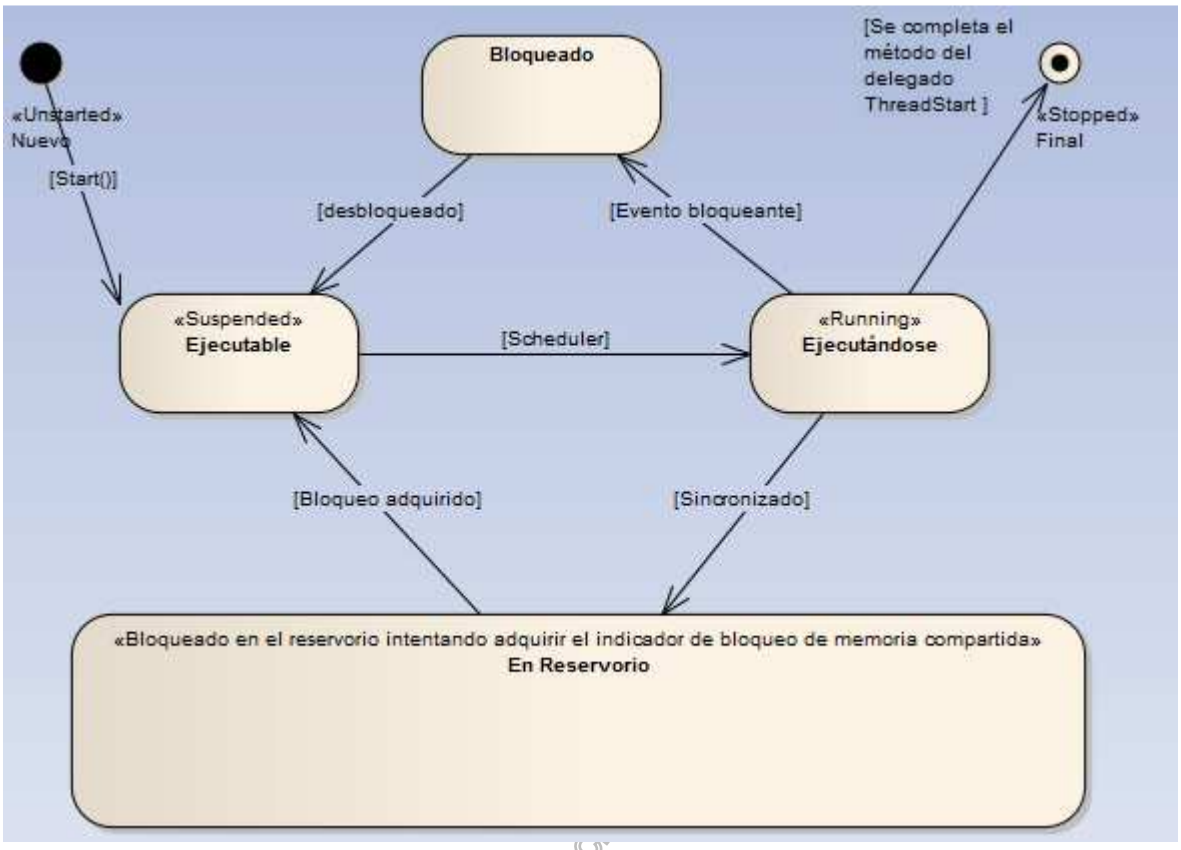
Uso de `lock` o `SyncLock` (C# o VB): conclusiones

El mecanismo de sincronización sólo funciona si todo el acceso a los datos "sensibles" se produce dentro de los bloques sincronizados.

Los datos protegidos por los bloques sincronizados deberían marcarse como `private` o `Private` (C# o VB). A esto se lo denomina memoria compartida. Si no se hace así, será posible acceder a los datos sensibles desde código situado fuera de la definición de la clase. Semejante situación permitiría a otro código sortear la protección y dañar los datos durante el tiempo de ejecución.

Diagrama de Estado de Sincronización de un Thread

Cuando se ejecuta un bloque de sincronizado, todo thread que queda a la espera de la oportunidad de ejecutarse mantiene una referencia en un reservorio (en inglés se lo suele llamar "pool") que es accesible por el Scheduler. Cuando se libera el indicador de bloqueo del objeto que posee la memoria compartida, los threads que estaban aguardando pasan a un estado de "ejecutable". El scheduler selecciona uno de ellos y le permite el acceso al indicador de bloqueo y le concede tiempo de CPU para ejecutarse. Un diagrama de estado que refleja la situación es el siguiente:



Interbloqueo (deadlock)

En programas donde hay varios threads compitiendo por acceder a diferentes recursos, puede producirse una situación conocida como interbloqueo. Se produce cuando un thread está esperando a adquirir un bloqueo retenido por otro, pero el otro thread está esperando a su vez otro bloqueo retenido por el primero. En esta situación, ninguno de los dos puede proseguir la ejecución hasta que el otro haya pasado el fin de su bloque `lock` o `SyncLock` (C# o VB). Como ninguno puede proseguir, tampoco pueden pasar el final de su bloque y su ejecución no termina nunca.

.Net ni detecta ni intenta evitar esta situación. Es responsabilidad del programador asegurarse de que no pueda producirse un interbloqueo.

Se pueden seguir simples reglas para evitar los interbloqueos que suelen ser difíciles de seguir cuando se diseña o se mantienen los programas y son las siguientes:

- *Si se dispone de múltiples objetos a los que se quiere proporcionar acceso sincronizado, tomar una decisión global sobre el orden en el que obtendrá los bloqueos y respetar dicho orden a lo largo del programa.*
- *Liberar los bloqueos en orden inverso a aquel con el que se han obtenido.*

- *Tratar de no anidar bloqueos cruzados.*

Ejemplo

C#

```
using System.Text;
using System.Threading;

namespace interbloqueo
{
    class Program
    {
        static StringBuilder texto = new StringBuilder();
        static StringBuilder otroTexto = new StringBuilder();

        static void Main(string[] args)
        {
            Thread primerThread = new Thread(new ThreadStart(Metodo1));
            Thread segundoThread = new Thread(new ThreadStart(Metodo2));
            primerThread.Start();
            segundoThread.Start();
        }

        public static void Metodo1()
        {
            lock (texto)
            {
                Thread.Sleep(10);
                for (int i = 1; i <= 20; i++)
                {
                    texto.Append(i.ToString() + " ");
                }
                lock (otroTexto)
                {
                    otroTexto.Append(texto.ToString());
                    for (int i = 1; i <= 20; i++)
                    {
                        otroTexto.Append(i.ToString() + " ");
                    }
                }
            }
        }
        public static void Metodo2()
        {
            lock (otroTexto)
            {
                Thread.Sleep(10);
                for (int i = 21; i <= 40; i++)
                {
                    otroTexto.Append(i.ToString() + " ");
                }
                lock (texto)
                {
                    texto.Append(otroTexto.ToString());
                    for (int i = 21; i <= 40; i++)
```

```
                    {
                        texto.Append(i.ToString() + " ");
                    }
                }
            }
        }
    }
}
```

VB

```
Imports System.Text
Imports System.Threading

Public Class BloqueosMuertos
    Private Shared texto As StringBuilder = New StringBuilder()
    Private Shared otroTexto As StringBuilder = New StringBuilder()

    Public Shared Sub Metodo1()
        SyncLock (texto)
            Thread.Sleep(10)
            For i As Integer = 0 To 19
                texto.Append(i.ToString() + " ")
            Next
            SyncLock (otroTexto)
                otroTexto.Append(texto.ToString())
                For i As Integer = 0 To 19
                    otroTexto.Append(i.ToString() + " ")
                Next
            End SyncLock
        End SyncLock
    End Sub

    Public Shared Sub Metodo2()
        SyncLock (otroTexto)
            Thread.Sleep(10)
            For i As Integer = 0 To 19
                texto.Append(i.ToString() + " ")
            Next
            SyncLock (texto)
                otroTexto.Append(texto.ToString())
                For i As Integer = 0 To 19
                    otroTexto.Append(i.ToString() + " ")
                Next
            End SyncLock
        End SyncLock
    End Sub
End Class

Imports System.Threading

Module Module1
    Sub Main()
        Dim primerThread As New _
             Thread(New ThreadStart(AddressOf BloqueosMuertos.Metodo1))
        Dim segundoThread As New _
```

```
            Thread(New ThreadStart(AddressOf BloqueosMuertos.Metodo2))
        primerThread.Start()
        segundoThread.Start()
    End Sub
End Module
```

Si se observa el código con detenimiento se puede ver que los métodos acceden a las variables de instancia en forma cruzada en bloques anidados, lo cual **asegura** un interbloque. La ejecución del programa debe pararse desde el entorno de desarrollo porque el programa no terminará nunca.

Interacción de los subprocesos: la clase Monitor

La clave para manejar un buen diseño de threads es no depender de su tiempo de ejecución, o sea, no pensarlos como trabajos secuenciales sino como procesamiento paralelo y no sincronizado. Cuando se quiere procesar threads en forma secuencial se debe recurrir a llamados de métodos como Join, pero en ese punto cabe el planteo si no se puede solucionar el problema con llamados a métodos simples. Sin embargo, se presentan situaciones en las cuales dos o más threads pueden necesitar cierto nivel de sincronía y este a su vez, sea independiente del thread principal de ejecución que los crea. Este es el caso en el cual los trabajos que realizan se relacionan de alguna manera y podría ser necesario programar algunas interacciones entre ellos.

En el desarrollo se presenta a menudo una situación en la cual una aplicación consulta a otra si un determinado suceso se ha producido. Este es el ejemplo en el que dos programas completamente disjuntos interaccionan. Por ejemplo, el driver del sistema de archivos de un sistema operativo se encarga de manejar todos los eventos que permiten gestionar documentos, directorios, etc., y una aplicación quiere “enterarse” si en un determinado directorio se ha cambiado la cantidad o el contenido de los archivos. Esta tendría supuestamente dos caminos a seguir: manejar por si mismas los archivos de dicho directorio o consultar a intervalos de tiempo si se produjeron cambios. La solución obvia siempre es la segunda porque la primera es una función del sistema operativo.

¿Cuál es el problema de este ejemplo? Qué los procesos son disjuntos, no se encuentran regidos por el mismo programa, por lo cual la única solución en consultar al otro programa por espacios de tiempo.

Se puede dar otra situación diferente y es que el programa maneje en su interior ambos procesos al mismo tiempo pero dichos procesos sean disjuntos. Por ejemplo, se quiere manejar la lectura y escritura a un archivo de bloques de información que puede ser actualizada aleatoriamente dentro del mismo. Es claro que si un bloque se actualiza mientras otro lee se produciría inconsistencias entre los bloques leídos y la información que realmente contiene el archivo porque mientras se está leyendo se puede estar modificando. Sin embargo del análisis también puede surgir que no se debe bloquear a todo el programa mientras se realiza una operación de entrada – salida porque éste necesita seguir procesando.

Dentro de las posibles soluciones hay una en particular que parece la óptima: seguir procesando y crear uno o más threads para leer, uno o más para escribir, que ambos tipos sean independientes del thread que los crea y que los de un tipo no puedan realizarse hasta que los del otro termine. La última afirmación indica indirectamente una situación de sincronía en la cual tipo de thread debe **avisar** a los del otro tipo que pueden iniciar su procesamiento.

Como se puede apreciar esto indica sincronía entre threads, pero no secuencialidad, ya que **cualquier thread de un tipo se puede ejecutar cuando cualquier thread del otro tipo indica que estás disponible los elementos que se desean procesar** (en este caso, el archivo a leer).

Si ambos tipos de threads compiten por obtener los recursos para esta operación, como ser el lugar de memoria donde se almacena lo leído, la información del tamaño del archivo, etc., donde puedan ocurrir operaciones de lectura o escritura, si uno de los de un tipo obtiene una parte de los recursos y otro de los otro tipo otra parte de **los mismos recursos necesarios, ambos se bloquean entre si esperando a que el otro libere recursos para bloquearlos y así lograr el acceso**. Esta es una situación clara de interbloqueo y se debe evitar.

Por lo tanto, se debe resolver ambos problemas a la vez, que un tipo de thread no corrompa la información que está usando los del otro tipo (situación que se resuelve sincronizando la memoria compartida) y que un tipo de thread se "entere" cuando el otro tipo deja la información y recursos a manejar en un estado consistente para su uso. En pocas palabras, los threads de un tipo esperan la notificación de los del otro tipo para procesar en forma confiable su tarea.

La clase Monitor controla el acceso a los objetos, concediendo un bloqueo para un objeto a un único subproceso. Los bloqueos de objetos permiten restringir el acceso a un bloque de código, normalmente denominado sección o área crítica, memoria compartida o memoria corrompible. Mientras que un subproceso posea el bloqueo de un objeto, ningún otro subproceso podrá adquirir ese bloqueo.

Nota: Monitor se debe utilizar para bloquear objetos (es decir, tipos por referencia), no tipos por valor.

La clase Monitor tiene las siguientes características:

- Está asociada a un objeto sobre demanda.
- Es independiente, lo que significa que se puede llamar a esta clase directamente desde cualquier contexto.
- No se puede crear una instancia de Monitor.

Para cada objeto sincronizado, se mantiene la siguiente información:

- Una referencia al subproceso que contenga el bloqueo actual.
- Una referencia a una cola de subprocesos en estado "Ejecutable" (ver diagrama de estados), que contenga los subprocesos que ya están listos para obtener el bloqueo.

- Una referencia a una cola en espera, que contenga los subprocesos que se encuentren a la espera de una notificación sobre un cambio de estado del objeto bloqueado (ver más adelante el diagrama completo de estados para procesos en estado de espera (WaitSleepJoin).

En la siguiente tabla se describen las acciones que pueden realizar los subprocesos que obtienen acceso a los objetos sincronizados:

Acción	Descripción
Enter ,TryEnter	Adquiere un bloqueo para un objeto. Esta acción marca el principio de una sección crítica. Ningún otro subproceso puede entrar en la sección crítica, a menos que esté ejecutando las instrucciones de la sección crítica utilizando un objeto bloqueado distinto.
Wait	Libera el bloqueo sobre un objeto para permitir a otros subprocesos bloquear y obtener acceso al objeto. El subproceso que realiza la llamada espera mientras otro subproceso tiene acceso al mencionado objeto. Las señales de pulsos (Pulse) se utilizan para notificar los cambios de estado de un objeto a los subprocesos en espera.
Pulse(señal),PulseAll	Envía una señal a uno o varios subprocesos en espera. La señal notifica al subproceso en espera que el estado de un objeto bloqueado ha cambiado, y el propietario del bloqueo está en condiciones de liberarlo. El subproceso en espera se coloca en la cola de subprocesos listos (estado Ejecutable) del objeto, de modo que pueda adquirir el bloqueo del objeto. Una vez que el subproceso dispone del bloqueo, puede comprobar el nuevo estado del objeto y así observar si se ha alcanzado el estado requerido.
Exit	Libera el bloqueo en un objeto. Esta acción también marca el final de una sección crítica protegida por el objeto bloqueado.

Utilizar los métodos Enter y Exit para marcar el principio y el final de una sección crítica. Si la sección crítica está formada por un conjunto de instrucciones contiguas, el bloqueo adquirido por el método Enter garantiza que sea un único subproceso el que pueda ejecutar el código delimitado en el objeto bloqueado. En tal caso, se recomienda incluir dichas instrucciones en un bloque `try` y colocar la instrucción Exit en un bloque `finally`. Esta utilidad se utiliza normalmente para sincronizar el acceso a un método de instancia o a un método estático de una clase. La funcionalidad proporcionada por los métodos Enter y Exit es idéntica a la que proporciona la instrucción `lock` de C# (`SyncLock` en Visual Basic), excepto que `lock` y `SyncLock` contienen la sobrecarga del método Enter(Object, Boolean) y el método Exit en un bloque `try`...`finally` (`Try`…`Finally` en Visual Basic) para asegurar que se libera el monitor.

A partir de .NET Framework 4, hay dos conjuntos de sobrecargas para los métodos TryEnter y Enter. Un conjunto de sobrecargas tiene un parámetro `ref` (`ByRef` en Visual Basic) `Boolean` que se establece atómicamente en `true` si se adquiere el bloqueo, aunque se produzca una excepción al adquirir el bloqueo. Usar estas sobrecargas si es fundamental liberar el bloqueo en todos los casos, incluso cuando los recursos que el bloqueo está protegiendo no están en un estado coherente.

De ahora en más, las instrucciones seleccionadas para manejar los bloqueos son `lock` de C# y `SyncLock` en Visual Basic por ser parte de cada lenguaje respectivamente.

Interacción entre los threads

Por lo que se explicó en la sección anterior, es necesario tener un mecanismo que permita a los threads interaccionar entre sí para que cuando se realiza algún tipo de procesamiento se habilite la capacidad de realizar otro procesamiento en un thread diferente. Esto va más allá de la utilización de recursos compartidos como la memoria. Esto significa que los recursos deben poseer cierto "estado" para otro tipo de thread pueda realizar sus tareas.

> *A este tipo de procesamiento en el cual dos o más tipos diferentes de procesos pueden trabajar sobre los mismos recursos para cambiar su estado se los denomina procesos concurrentes.*

Métodos Monitor.Wait y Monitor.Pulse

Cuando existe concurrencia, los procesos se encuentran en un estado denominado Rendezvous, la cual es una primitiva de sincronización asimétrica que permite a dos procesos concurrentes llamados, el solicitante y el invocado, intercambiar datos de forma coordinada. El proceso que solicita el rendezvous debe esperar en un punto de rencuentro hasta que el proceso invocado llegue allí. Igualmente el proceso invocado puede llegar al rendezvous antes que el solicitante y debe esperar que este llegue al punto de encuentro para poder continuar procesando. La imagen de esperar en un punto de encuentro corresponde a colocar un proceso en espera inactiva hasta que la cita se cumpla. Durante el rendezvous los procesos pueden intercambiar datos.

Los datos intercambiados corresponden a parámetros de una llamada (desde el solicitante hacia el proceso invocado) y a resultados de una llamada (desde el proceso invocado hacia el solicitante), sin necesidad de almacenamiento intermedio.

La clase `System.Threading.Monitor` proporciona dos métodos para posibilitar la comunicación entre threads: Wait y Pulse. Si uno hace una llamada a Wait referida a un objeto de comunicación (Rendezvous) x, ese thread detiene su ejecución hasta que otro genera una llamada a Pulse para ese mismo objeto x.

Retomando el ejemplo de la pila que se expuso anteriormente, se puede decir que ninguna operación del tipo Sacar se debe realizar en tanto y en cuanto no se realice primero alguna operación Poner. Por lo tanto, un subproceso que sacar de la Pila debe esperar (Wait) a que al menos un subproceso que este agregando una caracter termine y avise (Pulse).

Ejemplo

C#

```
using System;
using System.Threading;

namespace monitor
```

```
{
    class PilaSincronizada
    {
        private int indice = 0;
        private int[] datos = new int[20];

        public void Poner(int d)
        {
            lock (this)
            {
                datos[indice] = d;
                Console.WriteLine("Poner\t{0}\t Índice: {1}",
                    Thread.CurrentThread.Name, indice);
                indice++;
                Monitor.Pulse(this);
            }
        }

        public int Sacar()
        {

            int aux = 0;
            lock (this)
            {
                // Puede entrar el mismo thread u otro, pero
                // si no hay datos en el vector, esperar.
                while (datos.Length == 0)
                {
                    try
                    {
                        Monitor.Wait(this);
                    }
                    catch (SynchronizationLockException e)
                    {
                        Console.WriteLine(e);
                    }
                    catch (ThreadInterruptedException e)
                    {
                        Console.WriteLine(e);
                    }
                }
                indice--;
                aux = datos[indice];
                Monitor.Pulse(this);
            }
            Console.WriteLine("Sacar\t{0}\tValor: {1}\tÍndice: {2}",
              Thread.CurrentThread.Name, aux, indice);
            return aux;
        }
    }
}

using System;
using System.Threading;
```

```
namespace monitor
{
    class Program
    {
        private static PilaSincronizada ps = new PilaSincronizada();

        static void Main(string[] args)
        {
            Thread[] threadsSincronizadosPoner = new Thread[6];
            for (int i = 0; i < threadsSincronizadosPoner.Length; i++)
            {
                threadsSincronizadosPoner[i] = new Thread(
                    new ThreadStart(PonerDatosSincronizados));
                threadsSincronizadosPoner[i].Name = "Thread " + i;
                threadsSincronizadosPoner[i].Start();
//                threadsSincronizadosPoner[i].Join();
            }

            Thread[] threadsSincronizadosSacar = new Thread[6];
            for (int i = 0; i < threadsSincronizadosSacar.Length; i++)
            {
                threadsSincronizadosSacar[i] =
                    new Thread(new ThreadStart(SacarDatosSincronizados));
                threadsSincronizadosSacar[i].Name = "Thread " + i;
                threadsSincronizadosSacar[i].Start();
//                threadsSincronizadosSacar[i].Join();
            }

            Console.ReadKey();
        }

        public static void PonerDatosSincronizados()
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("El {0} intenta poner el valor {1}",
                    Thread.CurrentThread.Name, i);
                ps.Poner(i);
            }
        }

        public static void SacarDatosSincronizados()
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("El {0} saca el valor {1}",
                    Thread.CurrentThread.Name, ps.Sacar());
            }
        }
    }
}
```

VB

```
Imports System.Threading
```

```
Public Class PilaSincronizada
    Private indice As Integer = 0
    Private datos(20) As Integer

    Public Sub Poner(d As Integer)
        SyncLock Me
            datos(indice) = d
            Console.WriteLine("Poner" + vbTab + "{0}" _
                              + vbTab + "Índice: {1}", _
                              Thread.CurrentThread.Name, indice)
            indice += 1
            System.Threading.Monitor.Pulse(Me)
        End SyncLock
    End Sub

    Public Function Sacar() As Integer
        Dim aux As Integer
        SyncLock Me
            While datos.Count = 0
                Try
                    System.Threading.Monitor.Wait(Me)
                Catch e As SynchronizationLockException
                    Console.WriteLine(e)
                Catch e As ThreadInterruptedException
                    Console.WriteLine(e)
                End Try
            End While
            indice -= 1
            aux = datos(indice)
            System.Threading.Monitor.Pulse(Me)
        End SyncLock
        Console.WriteLine("Sacar" + vbTab + "{0}" _
                          + vbTab + "Valor: {1}" + vbTab + _
                          "Índice: {2}", Thread.CurrentThread.Name, _
                          aux, indice)
        Return aux
    End Function
End Class

Imports System.Threading

Module Module1
    Private ps As PilaSincronizada = New PilaSincronizada()

    Sub Main()
        Dim threadsDesincronizadosPoner(6) As Threading.Thread

        For i As Integer = 0 To threadsDesincronizadosPoner.Length - 2
            threadsDesincronizadosPoner(i) = New Thread( _
              New ThreadStart(AddressOf PonerDatosDesincronizados))
            threadsDesincronizadosPoner(i).Name = "Thread " + i.ToString
            threadsDesincronizadosPoner(i).Start()
            '            threadsDesincronizadosPoner(i).Join()
        Next
```

```
        Dim threadsDesincronizadosSacar(6) As Threading.Thread
        For i As Integer = 0 To threadsDesincronizadosSacar.Length - 2
            threadsDesincronizadosSacar(i) = New Thread( _
              New ThreadStart(AddressOf SacarDatosDesincronizados))
            threadsDesincronizadosSacar(i).Name = "Thread " + i.ToString
            threadsDesincronizadosSacar(i).Start()
            '            threadsDesincronizadosSacar(i).Join()
        Next

        Console.ReadKey()
    End Sub

    Public Sub PonerDatosDesincronizados()
        For i As Integer = 0 To 2
            Console.WriteLine("El {0} intenta poner el valor {1}",
              Thread.CurrentThread.Name, i)
            ps.Poner(i)
        Next
    End Sub

    Public Sub SacarDatosDesincronizados()
        For i As Integer = 0 To 2
            Console.WriteLine("El {0} saca el valor {1}", _
              Thread.CurrentThread.Name, ps.Sacar())
        Next
    End Sub
End Module
```


Notar los comentarios en los llamados al método Join. Estas instrucciones se dejaron a propósito en el código para que se pueda evaluar la diferencia entre las uniones de threads y el bloqueo de la memoria compartida.

Tener en cuenta también que para evitar intentar sacar datos de la pila cuando está vacía se verifica la propiedad Count del objeto del tipo Stack. Esto se realiza debido a que el bloqueo sólo controla el acceso a memoria, no su contenido, por lo tanto se debe evitar que se produzca una excepción tratando de sacar un objeto inexistente.

Pero tanto para esperar, como para notificar, se debe realizar cuando la tarea este completa y no a medio hacer, con lo cual estas declaraciones deben estar dentro de un bloque sincronizado. Para el ejemplo anterior, el bloque sincronizado estaría dado, por ejemplo, por dos métodos del tipo Sacar() y Poner(). En el primero se colocaría un Wait para esperar que la Pila esté disponible para sacar de la pila y en el segundo un Pulse para indicar que el proceso de ingreso de datos finalizó. De esta manera se pueden manejar tantas extracciones y agregaciones como se quieran en procesos independientes concurrentes al recurso (la pila) ***sin que se produzcan interbloqueos ni se corrompan los datos***.

Descripción de los reservorios

Cuando un thread ejecuta código sincronizado que contiene una llamada a Wait referida a un determinado objeto, ese subproceso se coloca en el grupo de espera para dicho objeto (reservorio o "pool" en inglés). Se debe tener en cuenta que, por lo general aunque no necesariamente, el thread que está ejecutando la tarea está declarado en un objeto diferente a aquel que posee el bloque sincronizado donde se invoque a Wait o Pulse. Asimismo, el thread que llama a Wait libera el indicador de bloqueo que posee de ese objeto de forma automática.

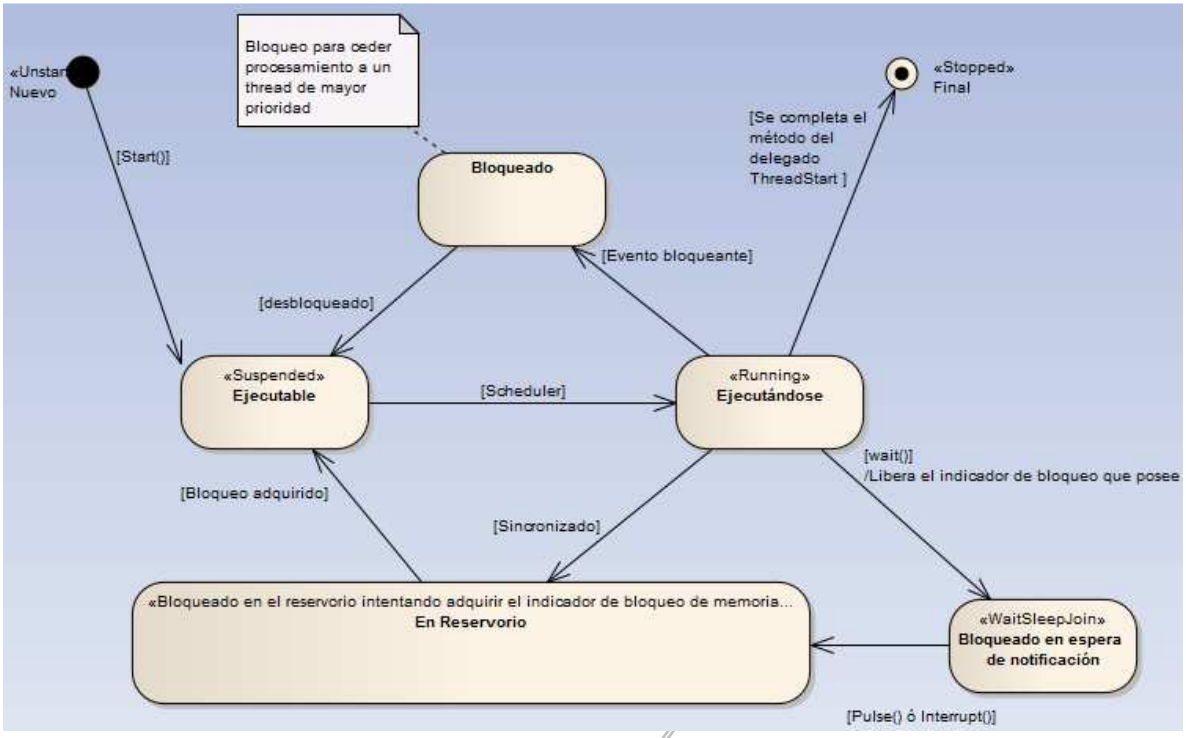
Cuando se ejecuta una llamada a Pulse referida a un determinado objeto (el cual es que posee los recursos compartidos que son accedidos desde un bloque sincronizado), un thread arbitrario se traslada desde el grupo de espera a un grupo de bloqueo para dicho objeto, donde los threads permanecen hasta que se libera el bloqueo de los recursos tomado por el bloque sincronizado que posee dicho objeto. Un thread sólo puede obtener el indicador de bloqueo del objeto desde el grupo de bloqueo, lo que le permite continuar ejecutándose en el punto donde se encontraba al llamar a Wait.

Es posible hacer una llamada a Pulse aunque no haya ningún thread esperando. Si se llama a éste método con referencia a un objeto cuando no hay ningún thread en el grupo de espera aguardando el indicador de bloqueo de ese objeto, la llamada no tiene ningún efecto.

Las llamadas a Pulse *no se almacenan para una ejecución posterior nunca. Se ejecutan sólo en el momento de la invocación y tienen efecto sólo si se dan las condiciones antes explicadas respecto de un* Wait.

Estados de los subprocesos con los reservorios

El grupo de espera también es un estado especial de los threads. En la siguiente figura se puede ver el diagrama de transición de los estados de un thread hasta que es seleccionable para el recolector de basura.



Modelo de control de la sincronización

La coordinación entre dos threads que necesitan acceso a datos comunes puede ser compleja. Es necesario garantizar que ningún thread deje datos compartidos en un estado incongruente cuando existe la posibilidad de que otro acceda a dichos datos. También es importante asegurar que el programa no producirá situaciones de interbloqueo porque uno no pueda liberar un bloqueo que otro está esperando.

Recordar que todos los threads del mismo grupo de espera deben recibir la oportuna notificación del objeto que controla ese grupo. No diseñar nunca código donde los subprocesos deban recibir notificación sobre situaciones diferentes en el mismo grupo de espera.

Ejemplo de interacción de threads

El código de esta sección contiene un ejemplo de interacción de threads que muestra el uso de los métodos Wait y Pulse para resolver un problema clásico de productor-consumidor. Para eso se va a utilizar la misma pila genérica mostrada en el ejemplo de la clase Monitor.

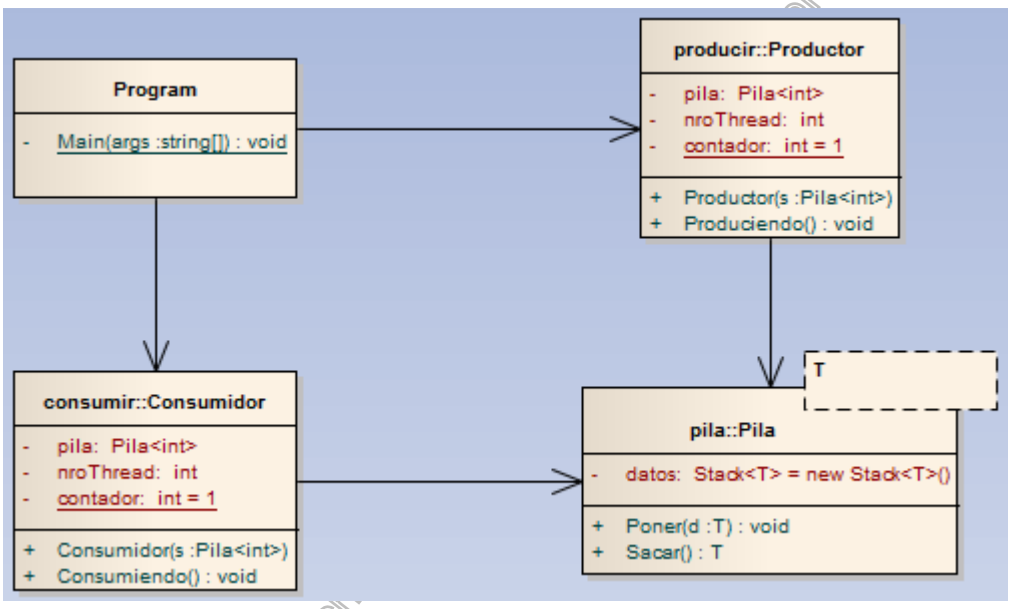
La idea parte de extender dicho ejemplo para que maneje el acceso de un objeto de tipo pila a la memoria compartida cuando invocan sus servicios threads concurrentes a ésta.

Con la idea de mostrar la separación de los elementos constituyentes del ejemplo, se colocó el productor, consumidor y la pila en ensamblados diferentes que conforman un mismo programa.

La idea es poder apreciar que el manejo de threads en .Net sólo se diferencia entre memoria gestionada (la del CLR) y la que no lo es.

Los servicios de objeto del tipo pila están brindados por los métodos Poner y Sacar. Ambos acceden al objeto genérico del tipo Stack definido en la clase para operar con él sacando y poniendo elementos. Si un thread accede al método sacar mientras que otro accede al método poner concurrentemente sobre esta pila, se está en presencia del caso mencionado anteriormente y debe manejarse para que ambos threads no invadan la memoria compartida simultáneamente.

Básicamente, el ejemplo planteado a partir de la clase Pila diseñada para los módulos anteriores tiene un diseño como el que se muestra a continuación:



Para resolver el problema de concurrencia se inicia declarando los métodos que acceden a la memoria compartida como bloques sincronizados. Esto implica que cada vez que se llame a uno de ellos, deberán competir para obtener un indicador de bloqueo cuando se encuentren llamando a los servicios de un mismo objeto.

Nota: *Cuando una clase declara bloques sincronizados en sus servicios determina que diferentes threads que acceden los servicios de un mismo objeto de su tipo deban obtener un indicador de bloqueo para acceder a las variables de instancia de éste. A esto se lo suele denominar "a salvo de los threads (thread safe)" porque ninguno puede invadir la memoria compartida cuando otro la está accediendo*.

La clase Pila

El código de la clase Pila queda como se muestra a continuación.

Ejemplo

C#

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace pila
{
    public class Pila<T>
    {
        private Stack<T> datos = new Stack<T>();

        public void Poner(T d)
        {
            lock (this)
            {
                datos.Push(d);
                Monitor.Pulse(this);
            }
            Console.WriteLine("Poner\t{0}\t Valor: {1}",
              Thread.CurrentThread.Name, d);
        }

        public T Sacar()
        {

            T aux = default(T);
            lock (this)
            {
                // Puede entrar el mismo thread u otro, pero
                // si no hay datos en el vector, esperar.
                while (datos.Count == 0)
                {
                    try
                    {
                        Monitor.Wait(this);
                    }
                    catch (SynchronizationLockException e)
                    {
                        Console.WriteLine(e);
                    }
                    catch (ThreadInterruptedException e)
                    {
                        Console.WriteLine(e);
                    }
                }
                aux = datos.Pop();
                Monitor.Pulse(this);
            }
            Console.WriteLine("Sacar\t{0}\tValor: {1}",
              Thread.CurrentThread.Name, aux);
            return aux;
        }
```

```
    }
}
```

VB

```
Imports System.Threading

Public Class Pila(Of T)
    Private datos As New Stack(Of T)

    Public Sub Poner(d As T)
        SyncLock (Me)
            datos.Push(d)
            Monitor.Pulse(Me)
        End SyncLock
        Console.WriteLine("Poner" + vbTab + "{0}" _
                          + vbTab + "Valor: {1}", _
                          Thread.CurrentThread.Name, d)
    End Sub

    Public Function Sacar() As T
        Dim aux As T
        SyncLock (Me)
            ' Puede entrar el mismo thread u otro, pero
            ' si no hay datos en el vector, esperar.
            While datos.Count = 0
                Try
                    Monitor.Wait(Me)
                Catch e As SynchronizationLockException
                    Console.WriteLine(e)
                Catch e As ThreadInterruptedException
                    Console.WriteLine(e)
                End Try
            End While
            aux = datos.Pop()
            Monitor.Pulse(Me)
        End SyncLock
        Console.WriteLine("Sacar" + vbTab + "{0}" _
                          + vbTab + "Valor: {1}", _
                           Thread.CurrentThread.Name, aux.ToString)
        Return aux
    End Function
End Class
```


Productor

A continuación se busca crear una clase que pueda producir elementos para un objeto de tipo de la clase Pila. La clase a crear debe poder llamar al método Poner del objeto del tipo Pila en un thread independiente porque se desea que un programa pueda crear tantos productores como le sea necesario.

Ejemplo

C#

```
using System;
using pila;

namespace producir
{
    public class Productor
    {
        private Pila<int> pila;
        private int nroThread;
        private static int contador = 1;

        public Productor(Pila<int> s)
        {
            pila = s;
            nroThread = contador++;
        }

        public void Produciendo()
        {
            for (int i = 0; i < 20; i++)
            {
                pila.Poner(i);
                Console.WriteLine("Productor " + nroThread + ": " + i);
            }
        }
    }
}
```

VB

```
Imports pila

Public Class Productor
    Private pila As Pila(Of Integer)
    Private nroThread As Integer
    Private Shared contador As Integer = 1

    Public Sub New(s As Pila(Of Integer))
        pila = s
        contador += 1
        nroThread = contador
    End Sub

    Public Sub Produciendo()
        For i As Integer = 0 To 19
            pila.Poner(i)
            Console.WriteLine("Productor " + nroThread.ToString _
                              + ": " + i.ToString)
        Next
    End Sub
End Class
```

Consumidor

Análogamente, se desea crear una clase que pueda consumir elementos de un objeto del tipo Pila. La clase a crear debe poder llamar al método sacar del objeto del tipo Pila en un thread independiente porque se desea que un programa pueda crear tantos consumidores como le sea necesario.

Ejemplo

C#

```
using System;
using pila;

namespace consumir
{
    public class Consumidor
    {
        private Pila<int> pila;
        private int nroThread;
        private static int contador = 1;

        public Consumidor(Pila<int> s)
        {
            pila = s;
            nroThread = contador++;
        }

        public void Consumiendo()
        {
            int aux = 0;

            for (int i = 0; i < 20; i++)
            {
                aux = pila.Sacar();
                Console.WriteLine("Consumidor " + nroThread + ": " + aux);
            }
        }
    }
}
```

VB

```
Imports pila

Public Class Consumidor
    Private pila As Pila(Of Integer)
    Private nroThread As Integer
    Private Shared contador As Integer = 1

    Public Sub New(s As Pila(Of Integer))
        pila = s
        contador += 1
        nroThread = contador
    End Sub
```

```
    Public Sub Consumiendo()
        Dim aux As Integer

        For i As Integer = 0 To 19
            aux = pila.Sacar()
            Console.WriteLine("Consumidor " + nroThread.ToString _
                              + ": " + aux.ToString)
        Next
    End Sub
End Class
```

El programa de verificación

El último paso restante para probar como diferentes productores y distintos consumidores pueden trabajar sobre un mismo objeto del tipo pila es ejecutarlos en un programa. Utilizando valores enteros como elementos de la pila, por una cuestión de simplicidad solamente, los productores producen valores y los consumidores los leen para presentarlos por pantalla en una salida por consola. Cabe notar que como la clase Pila es genérica, se puede utilizar cualquier tipo como elemento de ésta, pero las clases Productor y Consumidor que trabajan específicamente con los elementos que posea la pila, deberán ajustar su código en base al tipo que se quiera utilizar.

Ejemplo

C#

```
using System;
using System.Threading;
using pila;
using producir;
using consumir;

namespace productorConsumidor
{
    class Program
    {
        static void Main(string[] args)
        {
            Pila<int> pila = new Pila<int>();

            Productor p1 = new Productor(pila);
            Thread prodT1 = new Thread(new ThreadStart(p1.Produciendo));
            prodT1.Start();
 //           prodT1.Join();

            Consumidor c1 = new Consumidor(pila);
            Thread consT1 = new Thread(new ThreadStart(c1.Consumiendo));
            consT1.Start();
 //           consT1.Join();

            Productor p2 = new Productor(pila);
            Thread prodT2 = new Thread(new ThreadStart(p2.Produciendo));
            prodT2.Start();
```

```
//            prodT2.Join();

            Consumidor c2 = new Consumidor(pila);
            Thread consT2 = new Thread(new ThreadStart(c2.Consumiendo));
            consT2.Start();
//            consT2.Join();

            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System.Threading
Imports pila
Imports producir
Imports consumir

Module Module1

    Sub Main()
        Dim pila As New Pila(Of Integer)

        Dim p1 As New Productor(pila)
        Dim prodT1 As New Thread(New ThreadStart(AddressOf p1.Produciendo))
        prodT1.Start()
        '            prodT1.Join()

        Dim c1 As New Consumidor(pila)
        Dim consT1 As New Thread(New ThreadStart(AddressOf c1.Consumiendo))
        consT1.Start()
        '            consT1.Join()

        Dim p2 As New Productor(pila)
        Dim prodT2 As New Thread(New ThreadStart(AddressOf p2.Produciendo))
        prodT2.Start()
        '            prodT2.Join()

        Dim c2 As New Consumidor(pila)
        Dim consT2 As New Thread(New ThreadStart(AddressOf c2.Consumiendo))
        consT2.Start()
        '            consT1.Join()

        Console.ReadKey()
    End Sub

End Module
```


Una muestra de la posible salida (recordar que los valores se generan al azar) es la siguiente:

```
Poner           Valor: 0
Productor 2: 0
Poner           Valor: 1
Productor 2: 1
```

```
Poner           Valor: 2
Productor 2: 2
Poner           Valor: 3
Productor 2: 3
Poner           Valor: 4
Productor 2: 4
Poner           Valor: 5
Productor 2: 5
Poner           Valor: 6
Productor 2: 6
Poner           Valor: 7
Productor 2: 7
Poner           Valor: 8
Productor 2: 8
Poner           Valor: 9
Productor 2: 9
Poner           Valor: 10
Productor 2: 10
Poner           Valor: 11
Productor 2: 11
Poner           Valor: 12
Productor 2: 12
Sacar           Valor: 0
Consumidor 2: 0
Poner           Valor: 0
Productor 3: 0
Poner           Valor: 1
Productor 3: 1
Poner           Valor: 2
Productor 3: 2
Poner           Valor: 3
Productor 3: 3
Poner           Valor: 4
Productor 3: 4
Poner           Valor: 5
Productor 3: 5
Poner           Valor: 6
Productor 3: 6
Poner           Valor: 7
Sacar           Valor: 12
Consumidor 3: 12
Sacar           Valor: 7
Consumidor 3: 7
Sacar           Valor: 6
Consumidor 3: 6
```

```
Sacar           Valor: 5
Consumidor 3: 5
Sacar           Valor: 4
Consumidor 3: 4
Sacar           Valor: 3
Consumidor 3: 3
Poner           Valor: 13
Sacar           Valor: 13
Consumidor 2: 13
Sacar           Valor: 1
Consumidor 2: 1
Sacar           Valor: 0
Consumidor 2: 0
Sacar           Valor: 11
Consumidor 2: 11
Productor 3: 7
Poner           Valor: 8
Productor 3: 8
Poner           Valor: 9
Productor 3: 9
Poner           Valor: 10
Productor 3: 10
Poner           Valor: 11
Productor 3: 11
Poner           Valor: 12
Productor 3: 12
Poner           Valor: 13
Productor 3: 13
Poner           Valor: 14
Productor 3: 14
Poner           Valor: 15
Productor 3: 15
Poner           Valor: 16
Productor 3: 16
Poner           Valor: 17
Productor 3: 17
Poner           Valor: 18
Productor 3: 18
Productor 2: 13
Poner           Valor: 14
Productor 2: 14
Sacar           Valor: 2
Consumidor 3: 2
Sacar           Valor: 15
Consumidor 3: 15
```

```
Sacar           Valor: 14
Consumidor 3: 14
Sacar           Valor: 19
Consumidor 3: 19
Sacar           Valor: 18
Consumidor 3: 18
Sacar           Valor: 17
Consumidor 3: 17
Sacar           Valor: 16
Sacar           Valor: 10
Consumidor 2: 10
Sacar           Valor: 15
Consumidor 2: 15
Sacar           Valor: 14
Consumidor 2: 14
Sacar           Valor: 13
Consumidor 2: 13
Sacar           Valor: 12
Consumidor 2: 12
Poner           Valor: 15
Productor 2: 15
Poner           Valor: 16
Productor 2: 16
Poner           Valor: 17
Productor 2: 17
Poner           Valor: 18
Poner           Valor: 19
Productor 3: 19
Productor 2: 18
Poner           Valor: 19
Productor 2: 19
Consumidor 3: 16
Sacar           Valor: 19
Consumidor 3: 19
Sacar           Valor: 18
Consumidor 3: 18
Sacar           Valor: 17
Consumidor 3: 17
Sacar           Valor: 16
Consumidor 3: 16
Sacar           Valor: 10
Consumidor 3: 10
Sacar           Valor: 9
Consumidor 3: 9
Sacar           Valor: 11
```

```
Consumidor 2: 11
Sacar           Valor: 9
Consumidor 2: 9
Sacar           Valor: 8
Consumidor 2: 8
Sacar           Valor: 7
Consumidor 2: 7
Sacar           Valor: 6
Consumidor 2: 6
Sacar           Valor: 5
Consumidor 2: 5
Sacar           Valor: 4
Consumidor 2: 4
Sacar           Valor: 3
Consumidor 2: 3
Sacar           Valor: 2
Consumidor 2: 2
Sacar           Valor: 1
Consumidor 2: 1
Sacar           Valor: 8
Consumidor 3: 8
```