

Concurrencia en C#

Ángel Luis Bayón Romero
Alumno Colaborador de la asignatura

Mayo de 2016

Índice

1	Introducción al lenguaje	3
1.1	Técnicas básicas para compartir memoria.	3
2	Manipulación de hilos	3
2.1	Creación y ejecución de hilos sencillos	3
2.2	Hilos Demonio	4
2.3	Mayor complejidad	4
2.4	Pool de Threads	7
2.4.1	Heurísticas de creación de hilos.	8
2.4.2	Hilos de finalizado de E/S	8
2.5	Comparativa de tiempos.	9
2.6	Comparativa con Monte Carlo de π	11
3	Controlando la concurrencia.	13
3.1	Control con prioridades.	13
3.2	Lock y Synclock	13
3.3	Monitores	15
3.4	Eventos de Sincronización	17
3.5	Mutex	18
3.6	Clase Interlocked	19
3.7	Bloqueos ReaderWriterLock	20
3.8	Interbloqueos	22
4	Bibliografía	23
5	Enlaces de Interés.	23

1 Introducción al lenguaje

En este documento, vamos a ver varias formas de realizar ejecuciones diferentes de forma paralela. Pero... ¿por qué utilizar concurrencia en C#? La razón más obvia sería para poder explotar al máximo la potencia de nuestros ordenadores. El lenguaje ha ido introduciendo distintas modificaciones que iremos descubriendo a lo largo del documento sobre asincronía.

Lo primero que debemos saber del lenguaje es qué librerías van a ser utilizadas en el mismo. Para ello, siempre vamos a utilizar la librería `using System;` como se ha visto en el ejemplo de instalación, debido a que es la más completa. Vamos a crear una clase sencilla dentro de nuestro editor, y la dejaremos preparada con el main que utilizaremos.

```
1  using System;
2      public class PrimerHilo{
3          static public void Main (){
4
5          }
6      }
```

1.1 Técnicas básicas para compartir memoria.

Para este caso, los elementos en los que vamos a compartir los datos que van a manipular los hilos, va a realizarse en memoria, empleando la palabra reservada **static**, en los códigos posteriores veremos ejemplos concretos, no obstante, se adjunta la nomenclatura correspondiente:

```
1  public static int num = 0;
2  public static Object Objeto = new Object();
```

2 Manipulación de hilos

2.1 Creación y ejecución de hilos sencillos

Lo primero que debemos hacer para poder crear un hilo es añadir la librería encargada de los mismos `using System.Threading;`, con ella **Windows** va a identificar todo el comportamiento de los hilos.

Para crear el hilo previamente necesitamos generar un método de tipo void, que va a ser el que defina el comportamiento del mismo, y que no reciba parámetros de entrada como por ejemplo:

```
1  using System;
2  using System.Threading;
3
4  public class PrimerHilo
5  {
6      private static void HiloEjecutandose(){
7          Console.WriteLine("Me estoy ejecutando.");
8      }
```

```

8         }
9
10        static public void Main ()
11        {
12
13        }
14    }

```

Para este ejemplo vamos a realizar lo más sencillo, que sería mostrar por pantalla un mensaje.

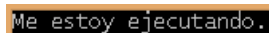
Para ejecutarlo, añadiremos unas simples líneas para convertir al método en un hilo que se va a encargar de envolver el método para pasarlo a hilo, quedando:

```

1  using System;
2  using System.Threading;
3
4  public class PrimerHilo
5  {
6      private static void HiloEjecutandose(){
7          Console.WriteLine("Me estoy ejecutando.");
8      }
9
10     static public void Main ()
11     {
12         Thread hilo = new Thread(HiloEjecutandose);
13         hilo.Start();
14
15         Console.ReadLine();
16     }
17 }

```

Añadimos el *ReadLine* para que no cierre el programa hasta que pulsemos cualquier tecla y comprobamos.



2.2 Hilos Demonio

En este apartado vamos a ver el tipo de hilo demonio, este hilo se queda en ejecución desde un segundo plano de forma constante, en este lenguaje encontramos que no existe ese tipo como tal, sino que es una propiedad de los propios hilos, ya que por defecto todos los hilos creados son los de tipo normal.

```

1  Hilo.IsBackground = true;

```

2.3 Mayor complejidad

Ahora vamos a intentar no sólo lanzar un hilo, sino lanzar múltiples. Vamos a intentar crear un vector de hilos y además vamos a introducir un elemento nuevo llamado "delegado". Dicho elemento no es más que un tipo que representa referencias a métodos de una lista de parámetros determinada y un tipo de

valor devuelto, por lo que lo usaremos en el ejemplo nuevo para incrementar la dificultad del mismo.

```
1  using System;
2  using System.Threading;
3
4  public class SegundoHilo{
5      public static int num = 0;
6
7      private static void HiloEjecutandose(){
8          for(int i=0; i<100; i++)
9          {
10             Console.WriteLine("Me estoy ejecutando "+num+".");
11             num++;
12         }
13     }
14
15     static public void Main (){
16         ThreadStart delegado = new ThreadStart(HiloEjecutandose);
17         int N = 10;
18
19         Thread[] hilos = new Thread[N];
20
21         for(int i= 0 ; i<N; i++)
22             hilos[i] = new Thread(delegado);
23
24         for(int i= 0 ; i<N; i++)
25             hilos[i].Start();
26
27         for(int i= 0 ; i<N; i++)
28             hilos[i].Join();
29
30         Console.ReadLine();
31     }
32 }
```

Ahora vemos con la ejecución que todos los hilos se han ejecutado y han esperado a acabar con las ejecuciones de los mismos, pero a su vez comprobamos que hay algunos valores raros.

```

Me estoy ejecutando 726.
Me estoy ejecutando 727.
Me estoy ejecutando 548.
Me estoy ejecutando 729.
Me estoy ejecutando 730.
Me estoy ejecutando 731.
Me estoy ejecutando 732.
Me estoy ejecutando 733.
Me estoy ejecutando 734.
Me estoy ejecutando 562.
Me estoy ejecutando 736.
Me estoy ejecutando 737.
Me estoy ejecutando 738.
Me estoy ejecutando 739.
Me estoy ejecutando 740.
Me estoy ejecutando 735.
Me estoy ejecutando 742.
Me estoy ejecutando 743.
Me estoy ejecutando 744.
Me estoy ejecutando 745.
Me estoy ejecutando 746.
Me estoy ejecutando 747.
Me estoy ejecutando 748.
Me estoy ejecutando 749.
Me estoy ejecutando 750.
Me estoy ejecutando 751.
Me estoy ejecutando 752.
Me estoy ejecutando 753.
Me estoy ejecutando 754.
Me estoy ejecutando 755.
Me estoy ejecutando 609.
Me estoy ejecutando 757.
Me estoy ejecutando 756.
Me estoy ejecutando 759.

```

Lo que ha ocurrido ha sido que el vector de hilos ha completado la ejecución de todos ellos, pero las impresiones son las que han saltado de orden debido a las prioridades de los propios hilos. No obstante, al ver que ha llegado a 1000 (0-999) comprobamos que no ha ocurrido ningún problema.

Otro ejemplo con distintas co-rutinas sería:

```

1  using System;
2  using System.Threading;
3  public class TercerHilo
4  {
5      public static int num = 0;
6
7      private static void HiloSumador(){
8          for(int i=0; i<100; i++){
9              Console.WriteLine("Me estoy ejecutando "+num+".");
10             num++;
11         }
12     }
13
14     private static void HiloRestador(){
15         for(int i=0; i<100; i++){
16             Console.WriteLine("Me estoy ejecutando "+num+".");
17             num--;
18         }
19     }
20
21     static public void Main ()
22     {
23         ThreadStart delegado = new ThreadStart(HiloSumador);
24         ThreadStart delegado2 = new ThreadStart(HiloRestador);
25         int N = 1000;
26         Thread[] hilosSumadores = new Thread[N];
27         Thread[] hilosRestadores = new Thread[N];
28     }

```

```

29     for(int i= 0 ; i<N; i++){
30         hilosSumadores[i] = new Thread(delegado);
31         hilosRestadores[i] = new Thread(delegado2);
32     }
33
34     for(int i= 0 ; i<N; i++){
35         hilosSumadores[i].Start();
36         hilosRestadores[i].Start();
37     }
38
39     for(int i= 0 ; i<N; i++){
40         hilosSumadores[i].Join();
41         hilosRestadores[i].Join();
42     }
43
44     Console.ReadLine();
45 }
46 }

```

En este caso hemos empleado un par de delegados en las líneas 23 y 24 por comodidad, pero se podría usar el mismo reestableciéndolo en el propio bucle.

2.4 Pool de Threads

C#, al igual que Java, incorpora un sistema de pool de hilos. Aunque, en este caso, un pool de hilos estará monitorizado mediante un invocador, que será el encargado de lanzar hilos en el pool en caso de ser necesario. No obstante, es preciso destacar que en la medida de lo posible se intentará reusar un hilo ya creado anteriormente, debido al coste de creación y destrucción de los mismos.

La forma más común de usar un pool de hilos en C# es mediante la clase `Task`, cargando la siguiente librería:

```
using System.Threading.Tasks;
```

Veamos un ejemplo sencillo de uso.

```
Task.Factory.StartNew(HiloEjecutandose);
```

En el ejemplo, `StartNew` asocia el método `HiloEjecutandose` a una cola de espera para ser ejecutado por el pool de hilos. Si un hilo se encuentra disponible, se ejecutará directamente. En caso contrario, se esperará en cola hasta que alguno de los hilos del pool lo esté. Podría haberse usado también el método `Task.Run`, incorporado en .NET 4.5 para tal fin.

Existen otras maneras de agrupar las diferentes tareas en subprocesos. La más evidente de ellas es a través de la clase `ThreadPool`. Su método `QueueUserWorkItem` funciona de una manera similar a `StartNew`. Se le asigna un delegado y se pondrán en cola los métodos para su ejecución. Sin embargo, se prefiere la clase `Task` introducida en .NET 4.0 debido a que `ThreadPool` utiliza una estrategia menos eficiente y también menos flexible a la hora de la asignación de tareas a los hilos.

```

1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class PoolThreads{
6      public static int num = 0;
7
8      private static void Sumador(){
9          for(int i=0; i<1000; i++){
10             Console.WriteLine("Me estoy ejecutando "+num+".");
11             num++;
12         }
13     }
14
15     private static void Restador(){
16         for(int i=0; i<1000; i++){
17             Console.WriteLine("Me estoy ejecutando "+num+".");
18             num--;
19         }
20     }
21
22     static public void Main (){
23         ThreadStart delegado = new ThreadStart(Sumador);
24         int N = 100;
25         Thread[] hilos = new Thread[N];
26
27         for(int i= 0 ; i<N; i++)
28             hilos[i] = new Thread(delegado);
29         for(int i= 0 ; i<N; i++)
30             hilos[i].Start();
31         for(int i= 0 ; i<N; i++)
32             hilos[i].Join();
33
34         Task[] tareas = new Task[N];
35         for(int i= 0 ; i<N; i++)
36             tareas[i] = Task.Factory.StartNew(Restador);
37         Task.WaitAll(tareas);
38         Console.ReadLine();
39     }
40 }

```

2.4.1 Heurísticas de creación de hilos.

El *Common Language Runtime* ajusta automáticamente el número de hilos basándose en la carga de trabajo que presentes. Las heurísticas que definen el ajuste no están documentadas, y han sido modificadas en cada nueva versión de .NET.

En .NET 4.5, el número máximo de hilos por defecto es de 1000 hilos para procesadores de 32 bit, y 32767 para procesadores de 64 bits.

2.4.2 Hilos de finalizado de E/S

Un pool contiene dos tipos de hebras: hebras trabajadoras, y hebras de finalizado de E/S.

Las hebras trabajadoras son usadas para la ejecución de los delegados que fueron mandados a la cola para su posterior lanzamiento mediante las técnicas an-

teriormente descritas. La clase `ThreadPool` también hace uso de dichas hebras mediante su método `QueueUserWorkItem`.

Las hebras de finalizado o de terminación de E/S son utilizadas para invocar métodos que se proporcionan como devoluciones de llamadas para cuando una operación de E/S (como la lectura de datos de un archivo o un socket) que se inició de forma asíncrona termine.

Internamente, el *Common Language Runtime* utiliza el mecanismo de puerto de finalización de E / S que Windows proporciona para el manejo de un gran número de operaciones asíncronas simultáneas de manera eficiente. Por su parte, el pool separa estas hebras de las trabajadoras.

Esto limita las posibilidades que que el sistema caiga en un *deadlock* cuando se ascienda al máximo número de hilos en el pool. Si el *Common Language Runtime* no tratara las hebras de terminación de E/S de manera separada, podría llegarse a un estado en el que todas las hebras del pool se mantuvieran ociosas esperando a que una operación de E/S termine. Dicho estado es susceptible de generar un *deadlock*, ya que podrían no existir hilos para la terminación de operaciones de E/S, y a los que el resto de hebras estuvieran esperando.

En la práctica, puede ignorarse esta distinción entre hebras en las tareas a ejecutar, ya que el *Common Language Runtime* decide cuál debe utilizar. Aunque puedes llegar a un conflicto con dicha distinción.

Por ejemplo, si se decide por alguna razón modificar el tamaño máximo del pool, es necesario especificar los límites específicos para cada tipo de hilo, mediante el método `SetMaxThreads`, el cual requiere ambos valores como argumentos.

2.5 Comparativa de tiempos.

Teniendo en cuenta las funciones que hemos declarado en el apartado anterior para los hilos como sumadores y restadores, supongamos ahora, que queremos tener una comparativa de eficiencia respecto al lanzamiento de hilos masivo frente al pool, modificaremos el código anterior de esta forma en el main correspondiente:

```
1  static public void Main (){
2      ThreadStart delegado = new ThreadStart(Sumador);
3      int N = 100;
4      Thread[] hilos = new Thread[N];
5
6      DateTime tiempoHilosInicio = DateTime.Now;
7
8      for(int i= 0 ; i<N; i++)
9          hilos[i] = new Thread(delegado);
10
11     for(int i= 0 ; i<N; i++)
12         hilos[i].Start();
```

```

13
14     for(int i= 0 ; i<N; i++)
15         hilos[i].Join();
16
17     DateTime tiempoHilosFinal = DateTime.Now;
18     DateTime tiempoPoolInicio = DateTime.Now;
19
20     Task[] tareas = new Task[N];
21
22     for(int i= 0 ; i<N; i++)
23         tareas[i] = Task.Factory.StartNew(Restador);
24
25     Task.WaitAll(tareas);
26     DateTime tiempoPoolFinal = DateTime.Now;
27
28     TimeSpan totalHilos = new TimeSpan(tiempoHilosFinal.Ticks
29         - tiempoHilosInicio.Ticks);
30     Console.WriteLine("Tiempo total en hilos:"+totalHilos.
31         ToString());
32     TimeSpan totalPool = new TimeSpan(tiempoPoolFinal.Ticks -
33         tiempoPoolInicio.Ticks);
34     Console.WriteLine("Tiempo total en el Pool:"+totalPool.
35         ToString());
36
37     Console.ReadLine();
38 }

```

Con ellos conseguimos usando las propias librerías del sistema, obtener una toma de tiempos simplificadas con las líneas 6, 17 (tomando los tiempos) y 28 (obteniendo la diferencia entre ellas en segundos con decimales) para después mostrarlo como un *String*.

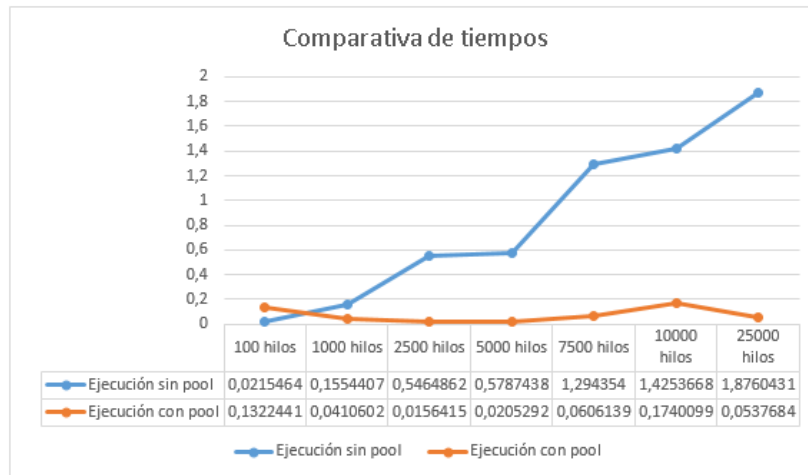
Se muestra un resultado como este:

```

Tiempo total en hilos:00:00:06.6428865
Tiempo total en el Pool:00:00:05.6906888

```

Ahora podemos observar distintos valores en hilos y pool que queda reflejada en esta gráfica comparativa.



Dichos valores han sido tomados con mi equipo que posee 4 núcleos.

Procesadores Intel(R) Core(TM) i5-4200H CPU @ 2.80 GHz, 2794 MHz, con 2 cores físicos y 2 virtuales, con una memoria caché de 1.9 GB, en un sistema operativo Windows 8.1 Enterprise, y con un compilador Mono JIT en su versión 4.2.1.

2.6 Comparativa con Monte Carlo de π .

En este apartado hemos elaborado un código simple, que controla el acceso a memoria de los hilos con un control simple aplicando un objeto como cerrojo, con el método *Lock* (Apartado 3.2), con el que hacemos distintas mediciones para comprobar la curva de ganancia en tiempo de esa versión en paralelo respecto a su versión iterativa (para este caso simplemente, el número de núcleos se reduce a 1).

```

1  using System;
using System.Threading;
using System.Threading.Tasks;
4
class PiMonteCarlo{
6      public static int SumaTotal = 0;
7      public static Object Locker = new Object();
8      public static int contador;
9
10     private static void coRutina(){
11
12         Random r = new Random();
13         int parcial = 0;
14
15         for(int i=0; i<contador; i++){
16             double x = r.NextDouble();
17             double y = r.NextDouble();
18             if(x*x + y*y <=1)

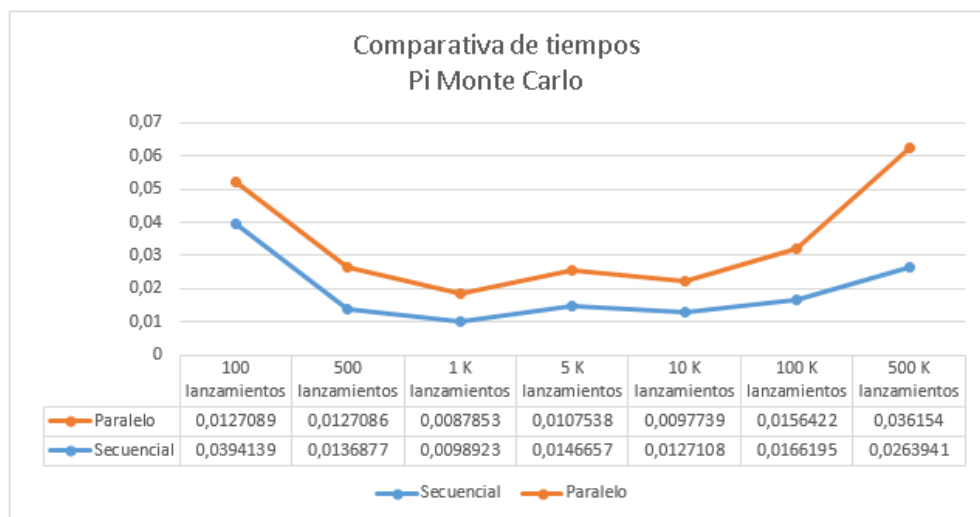
```

```

19             parcial++;
20         }
21     }
22     lock(Locker){
23         SumaTotal+=parcial;
24     }
25 }
26
27 static public void Main(){
28     double Lanzamientos = 10000;
29     int nTareas = Environment.ProcessorCount;
30     contador = (int) Lanzamientos/nTareas;
31
32     DateTime tiempoPoolInicio = DateTime.Now;
33
34     Task[] tareas = new Task[(int)nTareas];
35
36     for(int i= 0 ; i<nTareas; i++)
37         tareas[i] = Task.Factory.StartNew(coRutina);
38
39     Task.WaitAll(tareas);
40
41     DateTime tiempoPoolFinal = DateTime.Now;
42
43     Console.WriteLine("Resultado:" + (4*SumaTotal)/
44                       Lanzamientos + "\nRealizado en " + (
45                           tiempoPoolFinal - tiempoPoolInicio) + "
46                       segundos.");
47     Console.ReadLine();
48 }
49

```

Tras muchos lanzamientos, obtenemos un resultado como este:



Para este caso hemos empleado el código elaborado en las prácticas de la asignatura sobre el cálculo de π , pero lo hemos adaptado a C#.

3 Controlando la concurrencia.

3.1 Control con prioridades.

En el lenguaje encontramos que existen 5 niveles diferentes de prioridad indicados con los tipos : *Lowest*, *BelowNormal*, *Normal*, *AboveNormal* y *Highest*.

Aun que dichos niveles van a ser utilizados para priorizar entre los hilos, no significa que se puedan realizar en tiempo real, ya que su ejecución se encuentra limitada a la prioridad del propio proceso, veamos un ejemplo:

```
1  using System;
2  using System.Threading;
3
4  class Prioridad{
5      public static bool salto=true;
6      public static long contador = 0;
7
8      public void creador(){
9          contador++;
10         Console.WriteLine("Hilo "+contador+" ejecutandose.");
11     }
12
13     static public void Main (){
14         Prioridad p = new Prioridad();
15         ThreadStart del = new ThreadStart(p.creador);
16         Thread hilo1 = new Thread(del);
17         hilo1.Name = "Hilo 1";
18         Thread hilo2 = new Thread(del);
19         hilo2.Name = "Hilo 2";
20         hilo2.Priority = ThreadPriority.BelowNormal;
21         Thread hilo3 = new Thread(del);
22         hilo3.Name = "Hilo 3";
23         hilo3.Priority = ThreadPriority.AboveNormal;
24         hilo1.Start();
25         hilo2.Start();
26         hilo3.Start();
27         Thread.Sleep(10000);
28         Console.ReadLine();
29     }
30 }
```

3.2 Lock y Synclock

Las instrucciones *lock* (C#) y *SyncLock* (Visual Basic) se puede utilizar para garantizar que un bloque de código se ejecuta hasta el final sin que lo interrumpan otros subprocesos. Esto se logra obteniendo un bloqueo de exclusión mutua para un objeto determinado durante la ejecución de un bloque de código.

Para este caso vamos a aplicar el control con un cerrojo de tipo simple llamado "lock". Dicho cerrojo va a ser el encargado de permitir los accesos al atributo compartido, quedando:

```

1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ControlandoLock
6  {
7      public static int num = 0;
8      public static Object Locker = new Object();
9
10     private static void HiloEjecutandose(){
11         for(int i=0; i<10000; i++)
12         {
13             Console.WriteLine("Me estoy ejecutando "+num+"
14                               .");
15             lock (Locker){ num++; }
16         }
17     }
18     static public void Main ()
19     {
20         int N = 10;
21
22         for(int i= 0 ; i<N; i++)
23             Task.Factory.StartNew(HiloEjecutandose);
24
25         Console.ReadLine();
26     }
27 }

```

En este caso hemos decidido utilizar un pool de threads ya que hemos aprendido anteriormente como hacerlo.

Encontramos también que en la versión para Visual Basic la sección crítica se puede controlar con SyncLock, escribiendo como estructura

```

1  SyncLock Locker
2      [ Seccion Critica ]
3  End SyncLock

```

Generalmente, es mejor evitar el bloqueo en un tipo public o en instancias de objeto que estén fuera del control de la aplicación. Por ejemplo, lock(this) puede ser problemático si se puede tener acceso a la instancia públicamente, ya que el código que está fuera de su control también puede bloquear el objeto. Esto podría crear situaciones del interbloqueo, en las que dos o más subprocesos esperan a que se libere el mismo objeto. El bloqueo de un tipo de datos público, como opuesto a un objeto, puede producir problemas por la misma razón.

El bloqueo de cadenas literales es especialmente arriesgado porque el Common Language Runtime (CLR) interna las cadenas literales. Esto significa que hay una instancia de un literal de cadena determinado para todo el programa, exactamente el mismo objeto representa el literal en todos los dominios de la aplicación en ejecución, en todos los subprocesos. Como resultado, un bloqueo sobre una cadena que tiene el mismo contenido en cualquier parte del proceso de la aplicación bloquea todas las instancias de esa cadena en la aplicación.

Por tanto, es mejor bloquear un miembro privado o protegido que no esté internado. Algunas clases proporcionan específicamente los miembros para bloquear. Por ejemplo, el tipo `Array` proporciona `SyncRoot`. Muchos tipos de colección también proporcionan un miembro `SyncRoot`.

3.3 Monitores

El monitor como ya hemos aprendido en java es una forma bastante usada de controlar el acceso a aquellas variables que sean compartidas, para hacer uso de monitores en C# tenemos acceso a una librería propia que ya los trae implementados, dejandonos solamente teniendo que crear el objeto y usarlo como vigilante a los accesos, usando esta estructura:

```
1  using System;
2  using System.Threading;
3
4  public class PruebaMonitor{
5      var obj = new Object();
6
7      Monitor.Enter(obj);
8      try{
9          //Lo que haya con controlar
10     } finally{ Monitor.Exit(obj); }
11 }
```

Con ello vamos a controlar los accesos de forma que a la variable sean secuenciales y no haya problemas por lecturas de datos sin actualizar o sobreescrituras, solucionando los problemas de las secciones críticas.

Normalmente, es preferible utilizar la palabra clave *lock* (C#) o *SyncLock* (Visual Basic) en vez de utilizar directamente la clase *Monitor*, porque *lock* o *SyncLock* es más conciso y porque *lock* o *SyncLock* garantiza que se libera el monitor subyacente aunque el código protegido produzca una excepción. Esto se logra con la palabra clave *finally*, que ejecuta su bloque de código asociado independientemente de que se produzca una excepción.

Aplicando al código anterior de ejemplo y añadiendo más iteraciones al bucle vemos que se ha controlado y las visualizaciones se han hecho de la forma correcta.

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ControlandoMonitor
6  {
7      public static int num = 0;
8      public static Object monitor = new Object();
9
10     private static void HiloEjecutandose(){
11         for(int i=0; i<10000; i++){
12             {
13                 Console.WriteLine("Me estoy ejecutando "+num+".");
14                 Monitor.Enter(monitor);
```

```

15         try{ num++; } finally{ Monitor.Exit(monitor); }
16     }
17 }
18
19 static public void Main ()
20 {
21     int N = 10;
22     ThreadStart delegado = new ThreadStart(HiloEjecutandose
23         );
24
25     Thread[] hilos = new Thread[N];
26
27     for(int i= 0 ; i<N; i++)
28         hilos[i] = new Thread(delegado);
29
30     for(int i= 0 ; i<N; i++)
31         hilos[i].Start();
32
33     for(int i= 0 ; i<N; i++)
34         hilos[i].Join();
35
36     Console.ReadLine();
37 }

```

Los monitores también nos permiten como ya lo hacían en Java, hacer uso de los métodos "wait" y "notify" o "notifyAll", solo que en C# los vamos a llamar "Wait" y "Pulse" para sus correspondientes, ahora pensemos por ejemplo en el típico Productor-Consumidor, en C# sería:

```

1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ProductorConsumidor
6  {
7      public static int num = 0;
8      public static object Locker = new object();
9
10     private static void Prod(){
11         lock(Locker){
12             Console.WriteLine("Produciendo "+num+".");
13             num++;
14             while(num<1){ Monitor.Pulse(Locker); }
15         }
16     }
17
18     private static void Cons(){
19         lock(Locker){
20             Console.WriteLine("Consumiendo "+num+".");
21             num--;
22             while(num>0){ Monitor.Wait(Locker); }
23         }
24     }
25
26     static public void Main (){
27         int N = 1000;
28
29         for(int i= 0 ; i<N; i++)
30             Task.Factory.StartNew(Prod);
31

```



```

32         for(int i= 0 ; i<N; i++)
33             Task.Factory.StartNew(Cons);
34
35         Console.ReadLine();
36     }
37 }

```

3.4 Eventos de Sincronización

El uso de un bloqueo o un monitor es útil para evitar la ejecución simultánea de bloques de código utilizados por varios subprocesos, pero estas construcciones no permiten que un subproceso comunique un evento a otro. Esto requiere eventos de sincronización, que son objetos que tienen uno de dos estados (señalizado y no señalado) y se pueden utilizar para activar y suspender subprocesos.

Los subprocesos se pueden suspender haciendo que esperen a que se produzca un evento de sincronización que no esté señalado y se pueden activar cambiando el estado del evento a señalado. Si un subproceso intenta esperar a que se produzca un evento que ya está señalado, el subproceso se sigue ejecutando sin retraso.

Hay dos tipos de eventos de sincronización: `AutoResetEvent` y `ManualResetEvent`. Sólo difieren en que `AutoResetEvent` cambia automáticamente de señalado a no señalado siempre que activa un subproceso. A la inversa, `ManualResetEvent` permite que cualquier número de subprocesos esté activado si su estado es señalado y sólo vuelve al estado no señalado cuando se llama a su método `Reset`.

Se puede hacer que los subprocesos esperen en eventos llamando a uno de los métodos de espera como `WaitOne`, `WaitAny` o `WaitAll`. `WaitHandle.WaitOne` hace que el subproceso espere hasta que se señale un único evento, `WaitHandle.WaitAny` bloquea un subproceso hasta que se señalen uno o varios eventos especificados y `WaitHandle.WaitAll` bloquea el subproceso hasta que se señalen todos los eventos indicados. Un evento se señala cuando se llama a su método `Set`.

Para este tipo de sincronización hacemos uso de una clase de eventos propia del lenguaje que nos permite controlar las secciones críticas, pero necesita retardos para confirmar que se está haciendo de forma correcta, quedando:

```

1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ControlandoEvento
6  {
7      public static int num = 0;
8      public static AutoResetEvent autoevento;
9
10     private static void HiloEjecutandose(){
11         Console.WriteLine("Me estoy ejecutando "+num+".");
12         autoevento.WaitOne();

```

```

13         num++;
14     }
15
16     static public void Main ()
17     {
18         int N = 10000;
19         autoevento = new AutoResetEvent(false);
20         for(int i= 0 ; i<N; i++){
21             Task.Factory.StartNew(HiloEjecutandose);
22             autoevento.Set();
23             Thread.Sleep(250);
24         }
25         Console.ReadLine();
26     }
27 }

```

3.5 Mutex

La exclusión mutua es similar a un monitor lo que impide la ejecución simultánea de un bloque de código por más de un subproceso a la vez .De hecho, el nombre "mutex" es una forma abreviada del término "mutuamente exclusivo". Sin embargo, a diferencia de los monitores, una exclusión mutua se puede utilizar para sincronizar los subprocesos entre varios procesos. Una exclusión mutua se representa mediante la clase Mutex.

Cuando se utiliza para la sincronización entre procesos, una exclusión mutua se denomina una exclusión mutua con nombre porque va a utilizarla otra aplicación y, por tanto, no se puede compartir por medio de una variable global o estática. Se debe asignar un nombre para que ambas aplicaciones puedan tener acceso al mismo objeto de exclusión mutua.

Aunque se puede utilizar una exclusión mutua para la sincronización de subprocesos dentro de un proceso, normalmente es preferible utilizar Monitor porque los monitores se diseñaron específicamente para .NET Framework y, por tanto, hacen un mejor uso de los recursos. Por el contrario, la clase Mutex es un contenedor para una construcción de Win32. Aunque es más eficaz que un monitor, la exclusión mutua requiere transiciones de interoperabilidad, que utilizan más recursos del sistema que la clase Monitor. Para obtener un ejemplo de uso de la exclusión mutua.

Adaptamos nuestro código para probar el control con Mutex:

```

1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ControlandoMutex
6  {
7      public static int num = 0;
8      public static Mutex mut = new Mutex();
9
10     private static void HiloEjecutandose(){
11         mut.WaitOne();
12         Console.WriteLine("Me estoy ejecutando "+num+".");

```

```

13         num++;
14         mut.ReleaseMutex();
15     }
16
17     static public void Main ()
18     {
19         int N = 10000;
20         for(int i= 0 ; i<N; i++)
21             Task.Factory.StartNew(HiloEjecutandose);
22         Console.ReadLine();
23     }
24 }

```

3.6 Clase Interlocked

Con esta clase evitaremos los problemas que se pueden producir cuando varios subprocesos intentan actualizar al mismo tiempo un valor, controlando así la región crítica:

```

1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ControlandoInterlocked
6  {
7      public static int num = 0;
8
9      static void control(){
10         if(Interlocked.Exchange(ref num, 1)!=0)
11         {
12             Console.WriteLine("Tomando control.");
13             Thread.Sleep(500);
14             Console.WriteLine("Liberando control.");
15             Interlocked.Exchange(ref num, 0);
16         }
17         else
18             Console.WriteLine("Sin acceso a num.");
19     }
20
21     private static void HiloEjecutandose(){
22         for(int i=0; i<10; i++){
23             control();
24             Thread.Sleep(1000);
25         }
26     }
27
28     static public void Main ()
29     {
30         int N = 10000;
31         for(int i= 0 ; i<N; i++){
32             Thread.Sleep(250);
33             Task.Factory.StartNew(HiloEjecutandose);
34         }
35         Console.ReadLine();
36     }
37 }

```

3.7 Bloqueos ReaderWriterLock

En algunos casos, quizá desee bloquear un recurso sólo mientras se están escribiendo los datos y permitir que múltiples clientes puedan leer datos simultáneamente cuando no se estén actualizando los datos.

La clase `ReaderWriterLock` fuerza el acceso exclusivo a un recurso mientras hay un subproceso modificando el recurso, pero permite el acceso no exclusivo al leer el recurso. Los bloqueos de `ReaderWriter` son una alternativa útil a los bloqueos exclusivos que hacen esperar a otros subprocesos, incluso cuando esos subprocesos no necesitan actualizar datos.

En este caso haremos un código nuevo para probar el uso de la clase y ver como se controla la exclusión mutua:

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  public class ControlandoReaderWriterLock{
6
7      public static int num = 0;
8      static Random rnd = new Random();
9      static ReaderWriterLock rwl = new ReaderWriterLock();
10     static int readerTimeouts = 0;
11     static int writerTimeouts = 0;
12     static int reads = 0;
13     static int writes = 0;
14
15     static void ReadFromResource(int timeOut){
16         try{ rwl.AcquireReaderLock(timeOut);
17             try{
18                 Console.WriteLine("Leyendo dato " + num);
19                 Interlocked.Increment(ref reads);
20             } finally{ rwl.ReleaseReaderLock(); }
21         } catch (ApplicationException){
22             Interlocked.Increment(ref readerTimeouts);
23         }
24     }
25
26     static void WriteToResource(int timeOut){
27         try{ rwl.AcquireWriterLock(timeOut);
28             try{
29                 num = rnd.Next(500);
30                 Console.WriteLine("Escribiendo dato " + num);
31                 Interlocked.Increment(ref writes);
32             } finally{ rwl.ReleaseWriterLock(); }
33         } catch (ApplicationException){
34             Interlocked.Increment(ref writerTimeouts);
35         }
36     }
37
38     static void UpgradeDowngrade(int timeOut){
39         try{ rwl.AcquireReaderLock(timeOut);
40             try{
41                 Console.WriteLine("Leyendo dato" + num);
42                 Interlocked.Increment(ref reads);
43             } try {
44                 LockCookie lc = rwl.UpgradeToWriterLock(timeOut);
45                 try{
46                     num = rnd.Next(500);
```

```

47         Console.WriteLine("Escribiendo dato " + num);
48         Interlocked.Increment(ref writes);
49     } finally{ rwl.DegradeFromWriterLock(ref lc); }
50 } catch (ApplicationException) {
51     Interlocked.Increment(ref writerTimeouts);
52 }
53     Console.WriteLine("Leyendo dato " + num);
54     Interlocked.Increment(ref reads);
55 } finally { rwl.ReleaseReaderLock(); }
56 } catch (ApplicationException) { Interlocked.Increment(
    ref readerTimeouts); }
57 }
58
59 static void ReleaseRestore(int timeOut){
60     int lastWriter;
61
62     try{
63         rwl.AcquireReaderLock(timeOut);
64         try{
65             int resourceValue = num;
66             Console.WriteLine("Leyendo dato" + resourceValue);
67             Interlocked.Increment(ref reads);
68             lastWriter = rwl.WriterSeqNum;
69             LockCookie lc = rwl.ReleaseLock();
70             Thread.Sleep(rnd.Next(250));
71             rwl.RestoreLock(ref lc);
72
73             if (rw1.AnyWritersSince(lastWriter)) {
74                 resourceValue = num;
75                 Interlocked.Increment(ref reads);
76                 Console.WriteLine("dato cambiado" + resourceValue
77                     );
78             }
79             else
80                 Console.WriteLine("dato no cambiado" +
81                     resourceValue);
82         } finally{ rwl.ReleaseReaderLock(); }
83     } catch (ApplicationException){ Interlocked.Increment(
84         ref readerTimeouts); }
85 }
86
87 static void HiloEjecutandose(){
88     while (true){
89         double action = rnd.NextDouble();
90         if (action < .8)
91             ReadFromResource(10);
92         else if (action < .81)
93             ReleaseRestore(50);
94         else if (action < .90)
95             UpgradeDowngrade(100);
96         else
97             WriteToResource(100);
98     }
99 }
100
101 static public void Main (){
102     int N = 10000;
103     for(int i= 0 ; i<N; i++){
104         Thread.Sleep(250);
105         Task.Factory.StartNew(HiloEjecutandose);
106     }
107     Console.ReadLine();

```

```
105     }
106 }
```

Así queda implementado el Lector-Escritor.

3.8 Interbloqueos

La sincronización de subprocesos resulta de un valor incalculable en aplicaciones multiproceso, pero siempre existe el peligro de crear un *deadlock*, en el que varios subprocesos están esperando unos a otros y la aplicación se bloquea.

Un interbloqueo es una situación análoga a otra en la que hay automóviles parados en un cruce con cuatro señales de alto y cada uno de los conductores está esperando a que los otros se pongan en marcha. Evitar los interbloqueos es importante; la clave está en una cuidadosa planificación. A menudo es posible prevenir situaciones de interbloqueo mediante la creación de diagramas de las aplicaciones multiproceso, antes de empezar a escribir código.

```
1  using System;
2  using System.Threading;
3
4  public class Deadlock{
5
6      static object Obj1 = new object();
7      static object Obj2 = new object();
8
9      public static void Operacion1(){
10         lock(Obj1){
11             Thread.Sleep(1000);
12             lock(Obj2){ Console.WriteLine("Operacion 1
13                             ejecutandose."); }
14         }
15     }
16
17     public static void Operacion2(){
18         lock(Obj2){
19             Thread.Sleep(1000);
20             lock(Obj1){ Console.WriteLine("Operacion 2
21                             ejecutandose.");}
22         }
23     }
24
25     static public void Main(){
26         Thread Hilo1 = new Thread((ThreadStart)Operacion1);
27         Thread Hilo2 = new Thread((ThreadStart)Operacion2);
28
29         Hilo1.Start();
30         Hilo2.Start();
31     }
32 }
```

4 Bibliografía

Referencias.

- [1] I. Griffiths. Programming C# 5.0. *Programación*, 17, 2012.
- [2] J. Albahari and B. Albahari. C# 6.0 Pocket Reference. *Programación*, 2015.

5 Enlaces de Interés.

- *URL*: <http://nelson-venegas.blogspot.com.es/2012/05/procesos-hilos-threads-subproceso.html>
- *URL*: <http://jcesarmoreno.blogspot.com.es/2014/11/programacion-concurrente-en-c-thread.html>