

Seminario: Concurrencia en Lenguaje C#

Ángel Luis Bayón Romero,
Alumno Colaborador de la Asignatura

Julio de 2016

- Técnicas de creación de hebras.
 - Creación y ejecución de hilos, pools y ejecuciones.
- Control de concurrencia.
 - Lock.
 - Monitores.
 - Eventos de sincronización.
 - Mutex.
 - Clase Interlocked.
 - Bloqueos ReaderWriterLock.
 - Interbloqueos
- Bibliografía y enlaces de interés.

Técnicas de creación de hebras

Prueba1.cs

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

Técnicas de creación de hebras

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto. Puede invocar (o llamar) al método a través de la instancia del delegado.

Técnicas de creación de hebras I

Prueba2.cs

```
1 public class SegundoHilo{
2     public static int num = 0;
3
4     private static void HiloEjecutandose(){
5         for(int i=0; i<100; i++){
6             Console.WriteLine("Me estoy ejecutando "+
7                 num+".");
8             num++;
9         }
10    }
11    static public void Main (){
12        ThreadStart delegado = new ThreadStart(
13            HiloEjecutandose);
14        int N = 10;
15        Thread[] hilos = new Thread[N];
16
17        for(int i= 0 ; i<N; i++)
```

Técnicas de creación de hebras II

```
16         hilos[i] = new Thread(delegado);
17
18     for(int i= 0 ; i<N; i++)
19         hilos[i].Start();
20
21     for(int i= 0 ; i<N; i++)
22         hilos[i].Join();
23
24     Console.ReadLine();
25 }
26 }
```

Técnicas de creación de hebras I

Descargamos **Prueba3.cs**.

```
1 using System;
2 using System.Threading;
3
4 public class TercerHilo{
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 "+
10                             num+".");
11             num++;
12         }
13
14     private static void HiloRestador(){
15         for(int i=0; i<100; i++){
```

Técnicas de creación de hebras II

```
16         Console.WriteLine("Me estoy ejecutando "+
17             num+".");
18         num--;
19     }
20 }
21 static public void Main (){
22     ThreadStart delegado = new ThreadStart(
23         HiloSumador);
24     ThreadStart delegado2 = new ThreadStart(
25         HiloRestador);
26
27     int N = 1000;
28     Thread[] hilosSumadores = new Thread[N];
29     Thread[] hilosRestadores = new Thread[N];
30
31     for(int i= 0 ; i<N; i++){
32         hilosSumadores[i] = new Thread(delegado);
```


Técnicas de creación de hebras III

```
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 }
```

Técnicas de creación de hebras

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;
```

Descargamos **Pool.cs** y lo probamos.

Técnicas de creación de hebras I

Pool.cs

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 public class PoolThreads{
6
7     public static int num = 0;
8
9     private static void Sumador(){
10         for(int i=0; i<1000; i++){
11             Console.WriteLine("Me estoy ejecutando "+
12                               num+".");
13             num++;
14         }
15     }
16     private static void Restador(){
```

Técnicas de creación de hebras II

```
17     for(int i=0; i<1000; i++){
18         Console.WriteLine("Me estoy ejecutando "+
19             num+".");
20         num--;
21     }
22 }
23 static public void Main (){
24     ThreadStart delegado = new ThreadStart(Sumador);
25
26     int N = 100;
27     Thread[] hilos = new Thread[N];
28
29     for(int i= 0 ; i<N; i++)
30         hilos[i] = new Thread(delegado);
31
32     for(int i= 0 ; i<N; i++)
33         hilos[i].Start();
34 }
```

Técnicas de creación de hebras III

```
35     for(int i= 0 ; i<N; i++)
36         hilos[i].Join();
37
38     for(int i= 0 ; i<N; i++)
39         Task.Factory.StartNew(Restador);
40
41     Console.ReadLine();
42 }
43 }
```

Mejora de Rendimientos mediante Ejecutores

Para este caso, vamos a realizar una toma de tiempos comparativa, para ver como se mejora el rendimiento entre el vector de hilos y el pool que proporciona el lenguaje.

Descargamos **PoolTiempos.cs** y lo probamos.

Cuando realicemos las ejecuciones, deberemos añadir por línea de comandos el total de hilos que se desean lanzar.

Técnicas de creación de hebras I

PoolTiempos.cs

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 public class PoolThreads{
6
7     public static int num = 0;
8
9     private static void Sumador(){
10         for(int i=0; i<1000; i++){
11             //Console.WriteLine("Me estoy ejecutando
12                 "+num+".");
13             num++;
14         }
15     }
16     private static void Restador(){
```

Técnicas de creación de hebras II

```
17     for(int i=0; i<1000; i++){
18         //Console.WriteLine("Me estoy ejecutando
19             "+num+".");
20         num--;
21     }
22 }
23 static public void Main (String[] args){
24     ThreadStart delegado = new ThreadStart(Sumador);
25
26     int N = int.Parse(args[0]);
27     Thread[] hilos = new Thread[N];
28
29     DateTime tiempoHilosInicio = DateTime.Now;
30
31     for(int i= 0 ; i<N; i++)
32         hilos[i] = new Thread(delegado);
33
34     for(int i= 0 ; i<N; i++)
```


Técnicas de creación de hebras III

```
35         hilos[i].Start();
36
37     for(int i= 0 ; i<N; i++)
38         hilos[i].Join();
39
40     DateTime tiempoHilosFinal = DateTime.Now;
41
42     DateTime tiempoPoolInicio = DateTime.Now;
43
44     Task[] tareas = new Task[N];
45
46     for(int i= 0 ; i<N; i++)
47         tareas[i] = Task.Factory.StartNew(Restador
48             );
49
50     Task.WaitAll(tareas);
51
52     DateTime tiempoPoolFinal = DateTime.Now;
```

Técnicas de creación de hebras IV

```
53     TimeSpan totalHilos = new TimeSpan(  
        tiempoHilosFinal.Ticks - tiempoHilosInicio.  
        Ticks);  
54  
55     Console.WriteLine("Tiempo total en hilos:" +  
        totalHilos.ToString());  
56  
57     TimeSpan totalPool = new TimeSpan(  
        tiempoPoolFinal.Ticks - tiempoPoolInicio.  
        Ticks);  
58  
59     Console.WriteLine("Tiempo total en el Pool:" +  
        totalPool.ToString());  
60  
61     Console.ReadLine();  
62 }  
63 }
```

Técnicas de creación de hebras

También tenemos, como hemos estudiado en Java, la posibilidad de asignar prioridades a los hilos para que se ejecuten, para ello se proporcionan en el lenguaje 5 niveles de prioridad *Lowest*, *BelowNormal*, *Normal*, *AboveNormal*, *Highest*.

Descargamos **Prioridad.cs** y lo probamos.

También tenemos los hilos Daemon, en C# no existe este tipo de hilo como tal, sino que es una propiedad del propio hilo, haciendo uso del método *IsBackground*.

Ejemplo: **Hilo.IsBackground = true;**

Técnicas de creación de hebras I

Prioridad.cs

```
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+"
11                           ejecutandose.");
12    }
13
14    static public void Main (){
15        Prioridad p = new Prioridad();
16
17        ThreadStart del = new ThreadStart(p.creador);
```

Técnicas de creación de hebras II

```
17
18     Thread hilo1 = new Thread(del);
19     hilo1.Name = "Hilo 1";
20     Thread hilo2 = new Thread(del);
21     hilo2.Name = "Hilo 2";
22     hilo2.Priority = ThreadPriority.BelowNormal;
23     Thread hilo3 = new Thread(del);
24     hilo3.Name = "Hilo 3";
25     hilo3.Priority = ThreadPriority.AboveNormal;
26
27     hilo1.Start();
28     hilo2.Start();
29     hilo3.Start();
30
31     Thread.Sleep(10000);
32     Console.ReadLine();
33 }
34 }
```

Lock

Lock para C# garantizan que el bloque de código que proteja se va a ejecutar hasta el final sin que lo interrumpa otro subproceso.

Por norma general, es preferible evitar el bloqueo en un tipo public o en instancias de objetos fuera del control de la aplicación, ya que puede ser problemático.

ControlandoLock.cs

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 public class ControlandoLock{
6     public static int num = 0;
7     public static Object Locker = new Object();
8
9     private static void HiloEjecutandose(){
10         for(int i=0; i<10000; i++){
11             Console.WriteLine("Me estoy ejecutando "+
12                               num+".");
13             lock (Locker){
14                 num++;
15             }
16         }
17     }
18 }
```

Lock II

```
17
18 static public void Main () {
19     int N = 10;
20     for (int i = 0 ; i < N ; i++)
21         Task.Factory.StartNew(HiloEjecutandose);
22
23     Console.ReadLine();
24 }
25 }
```


Monitores

C# incorpora una directiva predeterminada para el uso de monitores por defecto.

Este mismo es el que nos permite realizar llamadas desde la propia clase en lugar de instanciar nada.

```
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 }
```

Descargamos **ControlandoMonitor.cs** y lo probamos.

ControlandoMonitor.cs

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 public class ControlandoMonitor{
6     public static int num = 0;
7     public static Object monitor = new Object();
8     private static void HiloEjecutandose(){
9         for(int i=0; i<10000; i++){
10             Console.WriteLine("Me estoy ejecutando " +
11                               num+".");
12             Monitor.Enter(monitor);
13             try{
14                 num++;
15             }
16             finally{
17                 Monitor.Exit(monitor);
18             }
19         }
20     }
21 }
```

Monitores II

```
17         }
18     }
19 }
20 static public void Main () {
21     ThreadStart delegado = new ThreadStart(
22         HiloEjecutandose);
23     int N = 10;
24     Thread[] hilos = new Thread[N];
25     for(int i= 0 ; i<N; i++)
26         hilos[i] = new Thread(delegado);
27
28     for(int i= 0 ; i<N; i++)
29         hilos[i].Start();
30     for(int i= 0 ; i<N; i++)
31         hilos[i].Join();
32     Console.ReadLine();
33 }
```


Monitores I

Un monitor nos permite hacer uso de métodos como *Wait* y *NotifyAll* o *Notify*, dichos métodos tienen su versión para C# pasando a ser *Wait* y *Pulse*, para comprobar su funcionamiento, se ha implementado una versión de Productor-Consumidor.

```
1  public static bool Wait(  
2      object obj  
3  )  
4  public static void Pulse(  
5      object obj  
6  )
```

Descargamos **ProdCons.cs** y lo probamos.

ProdCons.cs

```
1 using System;
2 using System.Runtime.CompilerServices;
3 using System.Threading;
4
5 class Buffer {
6     public const int tam = 3;
7     int[] buffer = new int[tam];
8     int Inipos = 0;
9     int Finpos = 0;
10    int elementos = 0;
11
12    public void Produce(int valor) {
13        lock(this){
14             if(elementos == tam){
15                Console.WriteLine("Buffer lleno.");
16                Monitor.Wait(this);
17            }
18        }
19    }
20 }
```


Monitores II

```
18
19         elementos++;
20         buffer[Inipos] = valor;
21         Console.WriteLine("Produciendo "+valor
22                             +".");
23         Inipos = (Inipos+1) % tam;
24         if (elementos == 1)
25             Monitor.Pulse(this);
26     }
27
28     public int Consume() {
29         lock(this){
30             if(elementos == 0){
31                 Console.WriteLine("Buffer vacio.");
32                 Monitor.Wait(this);
33             }
34
35             elementos --;
```

Monitores III

```
36         int res = buffer[Finpos];
37         Console.WriteLine("Consumiendo "+res
38                             +".");
39         Finpos = (Finpos+1) % tam;
40         if (elementos == tam - 1)
41             Monitor.Pulse(this);
42         return res;
43     }
44 }
45
46 class Productor {
47     Buffer buff;
48     int n = 0;
49
50     public Productor(Buffer b) {
51         buff = b;
52         Thread hilo = new Thread(new ThreadStart(Run
53             ));
```

Monitores IV

```
53         hilo.Start();
54     }
55
56     public void Run() { 
57         for (;;) {
58             buff.Produce(n++);
59             Thread.Sleep(1000);
60         }
61     }
62 }
63
64 class Consumidor {
65     Buffer buff;
66
67     public Consumidor(Buffer b) {
68         buff = b;
69         Thread hilo = new Thread(new ThreadStart(Run
70             ));
71         hilo.Start();
72     }
73 }
```


Monitores V

```
71     }  
72  
73     public void Run() {  
74         for (;;) {  
75             buff.Consume();  
76             Thread.Sleep(1000);  
77         }  
78     }  
79 }  
80  
81 public class ProdCons {  
82     static public void Main () {  
83         Buffer b = new Buffer();  
84         Productor p = new Productor(b);  
85         Consumidor c = new Consumidor(b);  
86     }  
87 }
```

Mutex

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`.

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.

El lenguaje nos proporciona la propia clase `Mutex`, para hacer uso de ella directamente.

Descargamos **ControlandoMutex.cs** y lo probamos.

Mutex I

ControlandoMutex.cs

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 public class ControlandoMutex{
6     public static int num = 0;
7     public static Mutex mut = new Mutex();
8
9     private static void HiloEjecutandose(){
10         mut.WaitOne();
11         Console.WriteLine("Me estoy ejecutando "+num
12             +".");
13         num++;
14         mut.ReleaseMutex();
15     }
16
17     static public void Main ()
```

Mutex II

```
17  {  
18      int N = 10000;  
19  
20      for(int i= 0 ; i<N; i++)  
21          Task.Factory.StartNew(HiloEjecutandose);  
22  
23      Console.ReadLine();  
24  }  
25 }
```

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.

Descargamos **ControlandoInterlock.cs** y lo probamos.

Clase Interlocked I

ControlandoInterlock.cs

```
1 using System;
2 using System.Threading;
3 using System.Threading.Tasks;
4
5 public class ControlandoInterlocked{
6     public static int num = 0;
7
8     static void control(){
9         if(Interlocked.Exchange(ref num, 1)==0){
10             Console.WriteLine("Tomando control.");
11             Thread.Sleep(500);
12             Console.WriteLine("Liberando control.");
13             Interlocked.Exchange(ref num, 0);
14         }
15         else
16             Console.WriteLine("Sin acceso a num.");
17     }
```

Clase Interlocked II

```
18
19 private static void HiloEjecutandose(){
20     for(int i=0; i<10; i++){
21         control();
22         Thread.Sleep(1000);
23     }
24 }
25
26 static public void Main (){
27     int N = 10000;
28
29     for(int i= 0 ; i<N; i++){
30         Thread.Sleep(250);
31         Task.Factory.StartNew(HiloEjecutandose);
32     }
33     Console.ReadLine();
34 }
35 }
```

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.

Evitar los interbloqueos es importante, la clave está en una cuidadosa planificación y a menudo es posible prevenir situaciones de interbloqueo mediante la creación de diagramas de las aplicaciones multiproceso, antes de empezar a escribir código.

Descargamos **Interbloqueo.cs** y lo probamos.

Clase Interlocked I

Interbloqueo.cs

```
1 using System;
2 using System.Threading;
3
4 public class Deadlock{
5     static object Obj1 = new object();
6     static object Obj2 = new object();
7
8     public static void Operacion1(){
9         lock(Obj1){
10             Thread.Sleep(1000);
11             lock(Obj2){
12                 Console.WriteLine("Operacion 1
13                                 ejecutandose.");
14             }
15         }
16     public static void Operacion2(){
```

Clase Interlocked II

```
17     lock(Obj2){
18         Thread.Sleep(1000);
19         lock(Obj1){
20             Console.WriteLine("Operacion 2
                ejecutandose.");
21         }
22     }
23 }
24
25 static public void Main(){
26     Thread Hilo1 = new Thread((ThreadStart)
        Operacion1);
27     Thread Hilo2 = new Thread((ThreadStart)
        Operacion2);
28     Hilo1.Start();
29     Hilo2.Start();
30 }
31 }
```

Bibliografía y enlaces de interés

- ❶ I. Griths. Programming C# 5.0. Programacion, 17, 2012.
 - ❷ J. Albahari and B. Albahari. C# 6.0 Pocket Reference. Programacion, 2015.
- *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>