|  |
| --- |
|  |
| UD 2: Programación multiproceso (III) |
|  |
| Programación de servicios y procesos |



**Centro de estudios SEIM**

4 de septiembre de 2012

Autor: Daniel Miguel

UD 2: Programación multiproceso (III)

Programación de servicios y procesos

Sincronización entre procesos en c#

En el documento anterior veíamos como los procesos pueden comunicarse entre sí mediante mediante pipes (con o sin nombre) o mediante el envío de mensajes de sistema si eran aplicaciones de ventana.

No obstante para que la comunicación sea exitosa es necesario que al mismo tiempo existan mecanismos de sincronización para evitar situaciones no deseadas derivadas de la concurrencia como veíamos en la UD 1.

La plataforma .net dispone de una serie de clases que ofrecen mecanismos de sincronización que permiten que cuando un proceso acceda de forma exclusiva a un recurso o que limitan el número de procesos que pueden estar en un determinado estado a la vez.

En este tema nos dedicaremos al estudio de estas clases.

# Semáforos:

Los semáforos limitan en número de subprocesos que pueden tener acceso a un recurso o grupo de recursos simultáneamente.

Una instancia de la clase semáforo se inicializa con un número inicial de “plazas” disponibles para subprocesos asociados al recurso o grupo de recursos y con un máximo de subprocesos concurrentes permitidos.

Cada vez que un proceso invoca al método **WaitOne** de una instancia de la clase semáforo, se decrementa en 1 el recuento de subprocesos, cuando llega a 0 no pueden entrar más.

Cuando el recuento llega a 0, el resto de los que invoquen al método **WaitOne** quedarán bloqueados y no se relanzarán hasta que alguno de los procesos en ejecución liberen el semáforo mediante el método **Release**.

La clase no lleva ningún control acerca del id del proceso que ocupa o libera el semáforo, lo cuál significa que es el programador quien tiene que asegurarse de que un proceso no libere el semáforo más veces de las que lo reserva. Si el recuento de procesos asociados al semáforo es el máximo de subprocesos permitidos y se vuelve a invocar a **Release** lo que ocurrirá será que se disparará la excepción se genera una excepción [SemaphoreFullException](http://msdn.microsoft.com/es-es/library/system.threading.semaphorefullexception(v=vs.80).aspx).

Los semáforos pueden ser locales o con nombre. Los que nos interesan en este momento son los segundos porque pueden ser compartidos por procesos independientes.

Los semáforos locales sirven para la sincronización de hilos y los veremos más adelante.

|  |
| --- |
| Constructor:Semaphore(Int32, Int32, String) Inicializa una nueva instancia de Semaphore indicando el recuento inicial de “plazas” disponibles, el límite de procesos y el nombre del semáforo. Métodos:Release() Sale del semáforo y devuelve el recuento anterior. Release(int32) Sale del semáforo un número especificado de veces. Devuelve el recuento anterior. WaitOne() Entra en el semáforo. WaitOne(Int32) Entra en el semáforo Bloquea el subproceso actual hasta que el objeto [WaitHandle](http://msdn.microsoft.com/es-es/library/system.threading.waithandle.aspx) actual recibe una señal, o transcurre un intervalo de tiempo, especificado utilizando un entero de 32 bits con signo para especificar el intervalo de tiempo. |

|  |
| --- |
| Ejemplo:Creación del semáforo. En el constructor del formulario Form1 del proceso padre, creamos un semáforo, con 2 plazas libres. El límite de subprocesos es 2  public Form1()  {  InitializeComponent();  s = new Semaphore(2, Vi2, "semaforo\_aparcar");  …    } Uso del semáforo. Al cargarse el formulario form1 del proceso hijo tendríamos lo siguiente:  public Form1()  {  InitializeComponent();  s = Semaphore.OpenExisting("semaforo\_aparcar");  …  }  Para entrar en el semáforo, hacemos lo siguiente en la función de respuesta al click del botón de entrada  private void button1\_Click(object sender, EventArgs e)  { …  button1.Enabled = false;  button2.Enabled = true;  s.WaitOne();  }  Y para salir del semáforo haríamos lo propio en el botón de respuesta al clic del botón de salida:  private void button2\_Click(object sender, EventArgs e)  { …  Int32 its=s.Release();//its recoge las plazas disponibles tras la salida.  button1.Enabled = true;  button2.Enabled = false;    } |

# Mutex:

Cuando dos o más subprocesos tienen que obtener acceso a un recurso compartido al mismo tiempo, el sistema necesita un mecanismo de sincronización para garantizar que sólo uno de los subprocesos utilice el recurso en ese momento. Las instancias de la clase Mutex otorgan acceso exclusivo al recurso compartido a un solo subproceso. Cuando un subproceso adquiere una exclusión mutua (mutex), el siguiente subproceso que intenta adquirir dicha exclusión mutua se suspende hasta que el primer subproceso libera la exclusión mutua.

Para solicitar la propiedad de una exclusión mutua puede utilizar el método WaitHandle.WaitOne. El subproceso que posee una exclusión mutua puede solicitar la misma exclusión mutua en llamadas repetidas a WaitOne sin bloquear su ejecución. Sin embargo, el subproceso debe llamar al método ReleaseMutex el mismo número de veces para liberar la propiedad de la exclusión mutua. La clase Mutex exige la identidad del subproceso, por lo que sólo podrá liberar la exclusión mutua el subproceso que la adquirió. Por el contrario, la clase Semaphore no exige la identidad del subproceso.

Si un subproceso finaliza mientras posee una exclusión mutua, se dice que la exclusión mutua está abandonada. El estado de la exclusión mutua se establece en señalado y el siguiente subproceso en espera obtiene la propiedad de la exclusión. Si no hay ningún propietario de la exclusión mutua, el estado de la exclusión mutua está señalado. A partir de la versión 2.0 de .NET Framework, se produce una excepción AbandonedMutexException en el siguiente subproceso que adquiere la exclusión mutua. En las versiones de .NET Framework anteriores a la 2.0, no se producía ninguna excepción.

Como los semáforos también pueden ser con nombre o locales. Los mútex con nombre son los que nos interesan para este tema, estando los locales adecuados para la sincronización de hilos, como veremos en la siguiente unidad.

|  |
| --- |
| Ejemplo: El siguiente ejemplo usa un mutex para no permitir más que una instancia de ejecución del mismo programa simultáneamente.  using System;  using System.Collections.Generic;  using System.Windows.Forms;  using System.Threading;  namespace SingleInstanceCSharp  {  static class Program  {  [STAThread]  static void Main()  {  bool mutexCreado = false;  Mutex miMutex = new Mutex(true, "nombreUnicoParaEstaAplicacion", out mutexCreado);  if (mutexCreado)  {  Application.EnableVisualStyles();  Application.SetCompatibleTextRenderingDefault(false);  Application.Run(new Form1());  miMutex.ReleaseMutex();  }  }  }  } |