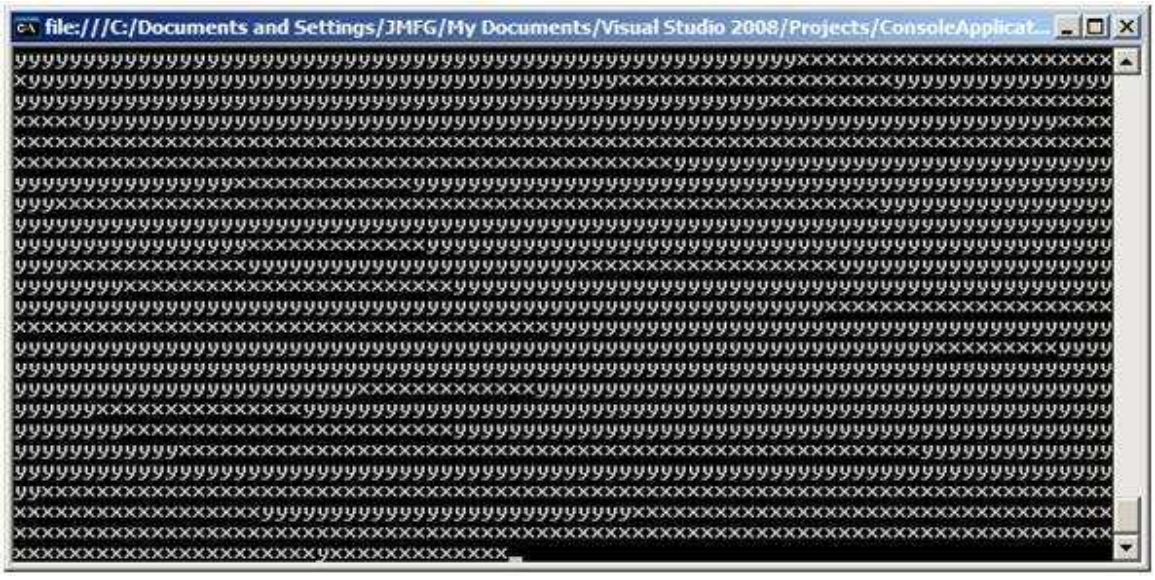
Hilos en C#: Conceptos

Archivado en: C# — elburgues @ 11:22 PM
Tags: MultiThreading

C# soporta la ejecución paralela de código a través de múltiples hilos. Un hilo es una ruta independiente de ejecución, capaz de ejecutarse simultáneamente con otros hilos. Un programa C# arranca en un hilo principal creado automáticamente por el CLR y el sistema operativo y puede estar compuesto de múltiples subprocesos mediante la creación de hilos adicionales.He aquí un ejemplo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
class PruebaHilos
{
static void Main()
{
Thread t = new Thread(EscribeY);
t.Start(); // Arranca EscribeY en un hilo nuevo.
while (true) Console.Write("x"); // Escribe 'x' indefinidamente.
}
static void EscribeY()
{
while (true) Console.Write("y"); // Escribe 'y' indefinidamente.
}
}
```



El hilo principal crea un nuevo hilo 't' en el cual se va a ejecutar un método que imprime el carácter 'y' repetidas veces en la consola. Simultáneamente, el hilo principal hace lo mismo con el carácter 'x'. El CLR asigna a cada hilo su propia pila de memoria de forma que las variables locales permanecen separadas. En el siguiente ejemplo se define un método con una variable local y se llama al método simultáneamente tanto desde el hilo principal como del hilo recién creado:

```
class PruebaHilos
{
static void Main()
{
new Thread(VamosAlla).Start(); // Llama a VamosAlla() en un nuevo hilo.
VamosAlla(); // Llama a VamosAlla() en el hilo principal.
}
static void VamosAlla()
{
// Declara y usa la variable local ciclos
for (int ciclos = 0; ciclos < 5; ciclos++) Console.Write('?');
}
}
```

Una copia separada de la variable "*ciclos*" es creada en cada pila de memoria de cada hilo, de forma que el carácter '?' se imprime 10 veces:

Los hilos comparten datos si tienen una referencia común a la misma instancia, por ejemplo:

```
class PruebaHilos
{
bool hecho;
static void Main()
{
PruebaHilos ph = new PruebaHilos(); // Crea una instancia común.
new Thread(ph.VamosAlla).Start();
ph.VamosAlla();
}
// Date cuenta de que VamosAlla() es ahora un método de instancia.
void VamosAlla()
{
if (!hecho) { hecho = true; Console.WriteLine("Hecho"); }
```

```
}
}
```

Al llamar ambos hilos al método *VamosAlla()* sobre la misma instancia de *PruebaHilos*, comparten el campo "*hecho*". Esto hace que se imprima el mensaje "*Hecho*" una vez en vez de dos veces:

Los campos estáticos ofrecen otra manera de compartir datos entre hilos. Éste es el mismo ejemplo con el campo "*hecho*" estático:

```
class PruebaHilos
{
static bool hecho; //Los campos estáticos son compartidos entre diferentes
hilos.
static void Main()
{
new Thread(VamosAlla).Start();
VamosAlla();
}
static void VamosAlla()
{
if (!hecho) { hecho = true; Console.WriteLine("Hecho"); }
}
}
```

Los dos últimos ejemplos ilustran otro concepto clave, el de la seguridad de procesos y es que la salida que se produce es actualmente indeterminada. Incluso es posible, aunque poco probable que la salida "*Hecho*" pudiera ser pintada dos veces por consola. Sin embargo, si cambiamos el orden de las instrucciones que contiene *VamosAlla()*, las posibilidades de que la salida "*Hecho*" se imprima dos veces aumentan de manera considerable:

```
static void VamosAlla()
{
if (!hecho) { Console.WriteLine("Hecho"); hecho = true; }
}
```

El problema es que un hilo puede estar evaluando la condición mientras que el otro hilo está ejecutando ya la sentencia que imprime en la consola antes de que tuviera la oportunidad de cambiar el valor de la variable "*hecho*". La solución es obtener un bloqueo exclusivo mientras se lee o se escribe en el campo común. C# proporciona la sentencia ***lock*** para éste propósito:

```
class PruebaHilos
{
static bool hecho;
static object bloqueador = new object();
static void Main()
{
new Thread(VamosAlla).Start();
VamosAlla();
}
// Date cuenta de que VamosAlla() es ahora un método de instancia.
static void VamosAlla()
{
lock (bloqueador)
{
if (!hecho) { Console.WriteLine("Hecho"); hecho = true; }
}
}
}
```

Cuando dos hilos sostienen simultáneamente un bloqueo, un hilo espera o se bloquea, hasta que el bloqueo se deshace y todo vuelve a estar disponible. Con esto nos aseguramos de que solamente un hilo puede entrar en la sección crítica de código a un mismo tiempo, y el contenido de la variable "*hecho*" será impreso en la consola siempre una sola vez. El código fuente protegido de esta manera, frente a indeterminaciones en un contexto multihilo es llamado seguridad de hilos.

Pausar temporalmente o bloquear, es una característica esencial en la coordinación o sincronización de las actividades de los hilos. Esperar a que se libere un bloqueo exclusivo es una razón por la que un hilo puede bloquearse. Otra razón es que un hilo quiera hacer una pausa (un hilo, mientras está pausado, no consume recursos de cpu). Se puede hacer así:

```
Thread.Sleep (TimeSpan.FromSeconds (30)); // Pausa de 30 segundos.
```

Se puede hacer que un hilo también espere a que otro hilo termine de ejecutarse por completo, usando el método *Join*:

```
Thread t = new Thread(VamosAlla);
t.Start();
t.Join(); // La ejecución se detendrá aquí hasta que el hilo 't' finalice.
```

Funcionamiento

Los hilos son manejados internamente por un programador o "gestor" de hilos, una función que el CLR típicamente delega en el sistema operativo. Un programador de hilos asegura que a todos los hilos se les asigna un tiempo de ejecución apropiado y que todos aquellos hilos que estén bloqueados o pausados no consuman tiempo de CPU. En un ordenador con un solo procesador, un programador de hilos gestiona una alternancia rápida entre los hilos activos, derivando en un comportamiento entrecortado (como se puede ver en la 1ª pantalla de consola de este artículo), dónde cada bloque de 'x' o de 'y' consecutivas se corresponde con una asignación de tiempo concreta para un hilo u otro. En Windows XP, la asignación de tiempo para cada hilo está en el orden de las decenas de milisegundos.

Hilos y procesos

Todos los hilos dentro de una aplicación están dentro de un proceso. Hilos y procesos tienen similitudes, por ejemplo, la ejecución de procesos también es alternada con otros procesos de la misma manera que sucede con los hilos dentro de una aplicación C#. La diferencia principal es que los procesos están completamente aislados unos de otros, sin embargo, los hilos comparten memoria con otros hilos que pertenecen al mismo proceso. Eso es lo que hace que los hilos sean útiles. Un hilo puede estar generando datos en un segundo plano mientras que otro hilo muestra los datos que van llegando.

¿Cuándo usar hilos?

Normalmente se utilizan para realizar trabajos en un segundo plano que típicamente vayan a consumir mucho tiempo hasta que se completen, por ejemplo, operaciones masivas con bases de datos, operaciones de cálculo…, mientras que la interfaz de usuario queda libre para poder trabajar en otra cosa. De otra manera, ésta no respondería, se vería bloqueada, aumentando la insatisfacción del usuario. También es posible que desde dicha interfaz pueda ofrecerse al usuario la posibilidad de cancelar este tipo de operaciones u ofrecer incluso información acerca del estado o porcentaje de operación completado. Una clase que puede resultar muy útil para este tipo de situaciones es la clase *BackGroundWorker*.

No todo son ventajas manejando hilos. La interacción entre hilos puede ser compleja y sin son usados en exceso o inadecuadamente, conllevan una penalización en el rendimiento de la CPU.

Manejo de Excepciones en los hilos

Archivado en: C# — elburgues @ 12:06 AM
Tags: MultiThreading
Consideremos el siguiente programa:

```
using System;
using System.Threading;

namespace Pruebas
{
   static class Program
   {
      /// <summary>
      /// El punto de entrada principal de la aplicación.
      /// </summary>
      [STAThread]
      public static void Main()
      {
         try
         {
            new Thread (Go).Start();
         }
         catch (Exception ex)
         {
            // Nosotros nunca llegaremos hasta aquí!
            Console.WriteLine ("¡Excepción!");
         }
      }

      static void Go()
      {
         throw null;
      }
   }
}
```

Resulta que la sentencia *Try/Catch* en este ejemplo no tiene ninguna utilidad y el nuevo hilo creado va a tener que cargar con una *NullReferenceException*. Este comportamiento adquiere sentido cuando se considera a los hilos como rutas independientes de ejecución. El remedio que se puede aplicar a esta situación es el que indica el siguiente código:

```
using System;
using System.Threading;

namespace Pruebas

{
   static class Program
```

```
    {
        /// <summary>
        /// El punto de entrada principal de la aplicación.
        /// </summary>
        [STAThread]
        public static void Main()
        {
            new Thread(Go).Start();
        }
        static void Go()
        {
            try
            {
                // Ésta excepción si que va a ser capturada abajo.
                throw null;
            }
            catch (Exception ex) {
                // Registrar de lo sucedido o informar a otros hilos
                // de que nos hemos vuelto inestables.
            }
        }
    }
}
>
```

Conclusión: **Al menos en aplicaciones que estén ya en producción, un manejo explícito de excepciones debe ser incorporado en todos los métodos de entrada de todos los hilos que hayamos creado en la aplicación**. Y es que desde *.NET Framework 2.0* en adelante, cualquier excepción no manejada en cualquier hilo dará al traste con la aplicación entera, así que ignorar esto que te estoy contando no debería ser una opción, pero bueno, tú eliges. Además, puede ser particularmente molesto para todos aquellos programadores *Windows Forms* que estén acostumbrados a usar un manejador global de excepciones para toda la aplicación, como el siguiente:

```
using System;
using System.Windows.Forms;

namespace Pruebas
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
```

```
            Application.ThreadException +=
                new
System.Threading.ThreadExceptionEventHandler(Application_ThreadException)
;
            Application.Run(new Form1());
        }
        static void Application_ThreadException(object sender,
System.Threading.ThreadExceptionEventArgs e)
        {
            // Éste es el manejador global de excepciones de toda la aplicación.
        }
    }
}
```

Las aplicaciones Windows están basadas en eventos. Eso quiere decir que actúan según mensajes que el sistema operativo manda al hilo principal de la aplicación. Estos mensajes son recibidos por la aplicación mediante llamadas repetitivas a una cola de mensajes en una sección de código llamada *"Windows message loop".* Aunque las aplicaciones .NET no requieren tener acceso directo a este *"Windows message loop"*, automáticamente encaminan dichos eventos (como las pulsaciones de teclas, ratón…) hacia sus apropiados manejadores definidos dentro del *Framework*, pero en realidad, *"Windows message loop"* está bajo el *Framework*.

El evento *Application.ThreadExecption* se dispara cuando una excepción es lanzada desde el último código que fue llamado como resultado de un mensaje de Windows, como por ejemplo una pulsación de teclado, de ratón y más en concreto, todo el código típico de una aplicación *Windows Form*. Esto puede crear una falsa sensación de seguridad de que todas las excepciones que se produzcan serán capturadas por el manejador centralizado de excepciones, pero no es así. Las excepciones lanzadas dentro de otros hilos de trabajo son un buen ejemplo de excepciones que no serán capturadas por el manejador centralizado de excepciones. El código que haya dentro del método *Main* o el constructor del formulario principal de la aplicación son otros dos buenos ejemplos, ya que son códigos que se ejecutan antes de que *"Windows message loop"* arranque.

.NET Framework provee de un evento de más bajo nivel para el manejo de excepciones globales: *AppDomain.UnhandledException*. Este evento se dispara cuando se produce una excepción no manejada en cualquier hilo, en cualquier tipo de aplicación (con o sin interfaz de usuario). De todas formas, aunque ofrece un buen mecanismo en última instancia para atrapar excepciones no manejadas, no ofrece la posibilidad de que la aplicación no se cierre, ni de que no aparezca el cuadro de diálogo de excepción no manejada.