Programación Visual en C#.NET

Guion de la práctica 6

OBJETIVOS

- Aprender a trabajar con hilos

TEMPORIZACIÓN

Recogida del enunciado:	Semana del 8 de noviembre
Desarrollo de la práctica:	**1 semana**
Fecha de entrega:	Semana del 15 de noviembre junto con la práctica 5
Fecha límite de entrega:	Semana del 22 de noviembre

PRÁCTICA 6

Hilos

TABLA DE CONTENIDOS:

Introducción 3
10. Hilos 3
10.1 Pequeños Ajustes. 3
10.2 Barra de Progreso. Funcionalidad de los hilos. 4
10.3 Creación de un hilo. 5
10.4 Delegados 5
10.5 Parar un hilo. 6
10.6 Creación de un segundo hilo. 7
10.7 Sincronización de hilos. 7
10.8 Detener los procesos relacionados con los hilos. 8

Introducción

La práctica 6 consiste en crear una sencilla aplicación para ver la utilidad de los hilos. Queda dividida en los siguientes puntos:

En el punto 10.1 implementará la interfaz básica del programa, dotándola de una mínima funcionalidad.

En el punto 10.2 implementará un método con un bucle de forma que mantenga ocupada a la aplicación durante un tiempo, impidiendo que los controles respondan a las órdenes del usuario.

En los puntos 10.3 a 10.5 creará un hilo, un recurso muy utilizado en las aplicaciones profesionales. Su uso permite ejecutar varias tareas de forma simultánea. Verá como interactúa con la aplicación, y cómo puede ser detenido.

En los puntos 10.6 y 10.7 creará un segundo hilo para estudiar los problemas que se presentan cuando los hilos no se sincronizan. Existen varios métodos de sincronización.

En el punto 10.7, utilizando la técnica de exclusión mutua, verá dos formas muy similares de sincronizar hilos: utilizando un cerrojo (*lock*) o un monitor.

10. Hilos

10.1 Pequeños Ajustes.

En esta práctica se creará una nueva aplicación que consistirá en un formulario como el siguiente:

Figura 1.

Asigne al formulario el nombre `Principal` y el título “Hilos”. Añada una barra de progreso (**ProgressBar**) con el nombre `bp_Progreso`, un botón con el nombre

`bt_Mostrar` y otro con el nombre `bt_Iniciar`. Para ver por qué es necesario utilizar hilos, añada una caja de imagen (**PictureBox**) llamada `pb_Imagen`. Además, añada a la aplicación una etiqueta con el texto "Velocidad:", y un control **NumericUpDown** llamado `ct_Velocidad`.

El control `pb_Imagen` lo utilizará para almacenar una imagen cualquiera, usando su propiedad **Image** (se aconseja que la imagen elegida esté guardada en la carpeta *Resources*, para facilitar la portabilidad del programa). Además, tal vez tenga que cambiar la propiedad **SizeMode** del **PictureBox** para que toda la imagen sea visible.

Utilizando los conocimientos adquiridos hasta el momento, implemente el código necesario para que la imagen se muestre u oculte al pulsar `bt_Mostrar`, y para que cambie el texto del botón de acuerdo con la visibilidad de la imagen.

Añada a la aplicación el código necesario para que el valor de la propiedad **Maximum** de `bp_Progreso` coincida en todo momento con el valor de `ct_Velocidad`. De esta forma, la barra de progreso se rellenará de forma más lenta o más rápida, en función del valor de `ct_Velocidad`. Después, defina para `ct_Velocidad` un rango de valores de 0 a 10.000.000 con incrementos de 10.000 y con un valor inicial de 10.000, y para la barra de progreso un rango de 0 a `ct_Velocidad.Value` con incrementos de 1.

Para finalizar este apartado, deberá crear un menú "Archivo" con la orden "Salir" y un menú "Ayuda" con la orden "Acerca de", implementando el código necesario para que ambas órdenes funcionen correctamente.

Compilar y ejecutar para ver el resultado.

10.2 Barra de Progreso. Funcionalidad de los hilos.

Cree un manejador para el botón "Iniciar". Añada a `Principal` un campo privado llamado `m_Valores` de tipo `List<int>`, y un campo entero llamado `m_Índice`, e inicie ambos al comienzo de dicho manejador (lista con 0 elementos e índice a 0).

Implemente un método privado `Progresar`, cuyo fin será insertar en la lista de valores un valor que coincidirá con el contenido de `m_Índice`. La barra de progreso avanzará en función de este valor.

```
private void Progresar()
{
  // Añadir elemento en la lista m_Valores.
  // Actualizar barra de progreso.
  // Incrementar m_Índice.
}
```

Cree otro método privado, llamado `ActualizarA`, desde el que se llamará a `Progresar` mientras el índice sea menor o igual que el valor máximo de la barra de progreso.

Al final del manejador del botón "Iniciar", inhabilite tanto dicho botón como el control `ct_Velocidad`, y a continuación llame a `ActualizarA`. Por último, añada el código necesario al final de `Progresar` para comprobar si el valor de la barra `bp_Progreso` ha llegado a su valor máximo; en este caso, se deberán restaurar los controles necesarios para que la aplicación vuelva a su estado inicial.

Compilar y ejecutar para ver el resultado.

Ejecute su aplicación. Pulse el botón "Iniciar" y, mientras la barra de progreso se llena, observe lo que ocurre si intenta pulsar el botón "Ocultar imagen" o si intenta mover la ventana.

Nota: A la hora de ejecutar la aplicación, tenga en cuenta que el valor **Maximum** de la barra de progreso es relativo a la potencia de la máquina en la que ejecutamos el programa. Dado que sólo se busca ocupar el microprocesador durante un tiempo prudente para ver la necesidad de usar hilos, ajuste este valor mediante `ct_Velocidad` para que la barra se llene en unos cuantos segundos.

10.3 Creación de un hilo.

En .NET un hilo es un objeto de la clase `System.Threading.Thread`. Añada a su formulario un hilo llamado `m_HiloA` (para simplificar el código, añada la directiva **using System.Threading** al comienzo del archivo, y así evitará tener que especificar este espacio de nombres en el resto del código).

Inicie el hilo en el manejador de `bt_Iniciar` de la siguiente forma (este código debe sustituir a la llamada a `ActualizarA`):

```
m_HiloA = new Thread(new ThreadStart(ActualizarA));
m_HiloA.Start();
```

Existe una forma de crear el hilo más abreviada. Pruébela.

Compilar y ejecutar para ver el resultado.

Ejecute el programa en modo depuración (pulsando F5). Comprobará que se produce una excepción de tipo `InvalidOperationException`. Esta excepción se lanza cuando accedemos a un control de interfaz de usuario (en este caso, la barra de progreso) desde un hilo distinto del principal. La solución a este problema, que veremos en el siguiente apartado, pasa por forzar que todas las operaciones con controles de la interfaz de usuario se ejecuten en el hilo principal.

10.4 Delegados

Un delegado es un objeto de la clase `Delegate` que representa un fragmento de código (generalmente un método) que se puede ejecutar. Es similar a un puntero a función en C++, aunque ofrece una mayor flexibilidad. Ya hemos usado uno en esta misma práctica, como parámetro del constructor de `Thread`. A continuación utilizaremos otro.

Sustituya en `Progresar` la línea en la que actualiza el valor de la barra de progreso por una llamada al siguiente método:

```
private void ActualizarBarraProgreso()
{
  if (bpProgreso.InvokeRequired)
  {
    MethodInvoker delegado =
                    new MethodInvoker(ActualizarBarraProgreso);
    // Elija la sentencia más adecuada de las dos siguientes:
```

```
        // bpProgreso.BeginInvoke(delegado);
        // bpProgreso.Invoke(delegado);
      }
      else
        bpProgreso.PerformStep();
    }
```

Este código hace que el método `ActualizarBarraProgreso` se ejecute en el hilo principal, en lugar de en el secundario. Aplique esta técnica al resto de controles a los que se accede desde `Progresar` (métodos `ActualizarBotonIniciar` y `ActualizarCtVelocidad`)

Compilar y ejecutar para ver el resultado. Utilice Ctrl+F5 (sin depuración).

¿Qué ocurre si prueba a cerrar la aplicación antes de que la barra de progreso se haya llenado? Abra el administrador de tareas de Windows y haga que se muestre la columna de subprocesos. Observe esta columna antes y después de lanzar los hilos.

10.5 Parar un hilo.

Para evitar que un hilo siga "vivo" tras haber cerrado la aplicación, deberá utilizar dos señales. La primera, que denominaremos "Parar", será activada por el hilo principal para notificar al secundario de que se debe detener, ya que la aplicación se está cerrando. La segunda señal, a la que llamaremos "Parado", servirá para que el hilo secundario avise al principal cuando se haya detenido.

En .NET, una señal se puede representar mediante un objeto de la clase `ManualResetEvent`. Así, las dos señales que necesitamos se pueden representar mediante los siguientes campos privados:

```
ManualResetEvent m_PararHiloA = new ManualResetEvent(false);
ManualResetEvent m_ParadoHiloA = new ManualResetEvent(false);
```

Cree ahora un método llamado `PararHiloA` que se encargará de decirle al hilo secundario que se detenga y de esperar a que dicha detención se haga efectiva. A la vez se procesarán eventos de la cola de eventos de la aplicación para que la aplicación siga respondiendo a las órdenes del usuario.

```
private void PararHiloA()
{
  if (m_HiloA != null && m_HiloA.IsAlive)
  {
    // Activamos el evento para decirle al hilo que pare
    m_PararHiloA.Set();
    // Esperar a que el hilo haya terminado
    while (m_HiloA.IsAlive)
    {
      m_ParadoHiloA.WaitOne(200,false); // evitar una espera activa
      Application.DoEvents(); // procesar eventos pendientes
    }
  }
}
```

Por otro lado, modifique el método `ActualizarA` para que verifique en cada iteración si se ha activado la señal "Parar". El hilo quedará bloqueado 0 milisegundos. En caso afirmativo, deberá activar a su vez la señal "Parado" y dejar de iterar.

Por último, cree un manejador para el evento **FormClosed** del formulario principal, llamando desde él a `PararHiloA`. Una vez que el hilo esté detenido, muestre un **MessageBox** para informar al usuario de esta circunstancia.

Compilar y ejecutar para ver el resultado.

10.6 Creación de un segundo hilo.

Añada un nuevo hilo, llamado `m_HiloB`, vinculado a un método `ActualizarB` copiado de `ActualizarA`. Este segundo hilo también se pondrá en marcha cuando se pulse el botón "Iniciar". Asegúrese de que se detiene correctamente al cerrarse la aplicación (use para ello un método `PararHiloB` y dos nuevas señales `m_PararHiloB` y `m_ParadoHiloB`).

Compilar y ejecutar para ver el resultado.

Ahora existen dos hilos haciendo avanzar la barra de progreso y añadiendo valores a la lista de forma concurrente. Para comprobar que pese a ello todo funciona correctamente, añada al final del método `ActualizarA` el siguiente código (fuera del bucle):

```
for (int i = 0; i < m_Valores.Count; i++)
   if (i != m_Valores[i])
   {
      MessageBox.Show("valores[" + i + "] = " + m_Valores[i] +
                      " y debía valer " + i);
      break;
   }
```

Compilar y ejecutar para ver el resultado.

Puede que en apariencia no haya ningún problema, lo cual no quiere decir que la aplicación esté correctamente diseñada. Para comprobarlo, fuerce un cambio de contexto, con `Thread.Sleep(0)`, antes de incrementar el índice en el método `Progresar`. Después de esta comprobación, quite `Thread.Sleep(0)`.

Compilar y ejecutar para ver el resultado.

Como puede ver ahora existen valores erróneos en la lista, lo que significa que los hilos no están cooperando correctamente (no están añadiendo a la lista los valores que se supone que deben añadir). A este tipo de problema se le denomina habitualmente "condición de carrera").

10.7 Sincronización de hilos.

C# y la biblioteca de .NET nos ofrecen variados mecanismos de sincronización de hilos. El más habitual es la sentencia **lock**, que permite crear zonas de exclusión mutua con mucha comodidad. Su sintaxis es la siguiente:

```
lock(objeto) // exclusión mutua
{
   // sección crítica
}
```

En general evite bloquear un objeto **public** que esté fuera del control de su código. Por ejemplo, `lock(this)` incumpliría esta recomendación. Es mejor utilizar un objeto **privado**.

Dentro de los métodos `ActualizarA/B` introduzca las llamadas a `Progresar` en sendos bloques `lock` para que se ejecuten en exclusión mutua.

Compilar y ejecutar para ver el resultado. Utilice Ctrl+F5 (sin depuración).

La sentencia `lock` equivale a utilizar el siguiente código:

```
Monitor.Enter(objeto);
try // sección crítica
{
   // Código en exclusión mutua
}
finally
{
   Monitor.Exit(objeto);
}
```

Para comprobarlo sustituya en su programa los bloques `lock` por fragmentos de código del tipo anterior.

Compilar y ejecutar para ver el resultado.

Nota: Usar `lock` tiene dos ventajas frente a `Monitor.Enter/Exit`: el código resultante es más claro y compacto, y además es más eficiente ya que el objeto usado para la sincronización sólo se evalúa una vez.

A través de `Invoke`/`BeginInvoke` también se puede obtener exclusión mutua, gracias a que el código ejecutado con dicho método se ejecuta en el hilo principal. Para comprobarlo, elimine cualquier otro mecanismo de exclusión mutua (en nuestro caso `lock`) y mueva la línea que añade el índice a la lista y la que lo incrementa, ambas del método `Progresar` a uno de los métodos que se ejecutan con `Invoke`/`BeginInvoke`, en nuestro caso al método `ActualizarBarraProgreso`. Después de la comprobación, deshaga los cambios para dejar el código como estaba antes de estas modificaciones.

Compilar y ejecutar para ver el resultado.

Si tiene problemas al ejecutar la aplicación, pruebe a cambiar `Invoke` por `BeginInvoke` o viceversa en `ActualizarBarraProgreso`.

10.8 Detener los procesos relacionados con los hilos.

Para tener la posibilidad de interrumpir los procesos relacionados con los hilos en cualquier instante, añada al menú "Archivo" una orden "Detener". Una vez detenidos los hilos se podrá volver a reiniciar la ejecución de los mismos cuando se requiera. Escriba el controlador de la orden "Detener" para que, utilizando únicamente los métodos ya escritos en la aplicación, permita realizar la tarea descrita.

Compilar y ejecutar para ver el resultado.

Ejecute el programa en modo depuración (pulsando F5) y compruebe que no se produce una excepción de tipo `InvalidOperationException` debido a que un hilo secundario intenta modificar el estado de un control de la interfaz de usuario.