PROGRAMACIÓN MULTI-HILO EN C# .NET

Un programa que usa Windows Forms, tiene por defecto un solo hilo de ejecución, por lo que toda instrucción se ejecuta una tras otra, sin cambiar de entorno. Internamente se trata de un bucle infinito que procesa todos los movimientos y clicks del ratón, así como pulsaciones del teclado. Dichas acciones envía mensajes a una cola, para ser tratados por un Scheduler que llama a los diferentes Event Handlers (despachadores de eventos). A ese hilo principal le podemos llamar UI Thread (o hilo del User Interface, hilo del interfaz gráfico). Pero ¿qué pasaría si una parte de ese código tuviera que hacer un proceso que tarde 10 segundos en terminar?. Es fácil de probar, solo tienes que coger el ejemplo Threads (en https://github.com/vidanio/csharping), que ejecuta diferentes contadores accedidos por 10 Timers, desde sus eventos Tick, y añadirle solamente a uno de ellos una linea como:

```
Thread.Sleep(10000); // espera durante 10 sec
```

Si ejecutas dicho ejemplo, verás que durante períodos de 10 segundos, el interfaz gráfico se queda bloqueado, y no responde. Eso ocurre, porque aunque los Timers parecen ejecutar procesos separados, a la vez, en realidad se hacen en un solo hilo, uno detrás de otro, conforme terminan. De hecho todos los componentes gráficos (botones, textbox, combos, listbox, etc) son controlados desde este hilo principal (UI Thread), y si tratas de controlar cualquier componente gráfico creado en el hilo principal desde otro hilo secundario, te saltará una excepción en plena ejecución. Como ejemplo de esto puede coger el ejemplo TestHilos, y descomentar la linea 55 del mismo, y luego ejecutarlo y pulsar el botón Start, que crea 2 hilos secundarios, y donde el hilo1, trata de cambiar el contenido del texto del label.

Queda entonces claro, que el hilo principal (UI Thread) es donde se crean todos los componentes gráficos, y que estos no pueden ser accedidos desde hilos secundarios. Sí que podemos acceder a variables globales o locales creadas en el hilo UI, sin que salten excepciones. Podemos incluso usar Mutex para proteger el acceso a una variable desde varios hilos a la vez. Sin embargo, no podemos acceder a las propiedades de los componentes visuales ¿entonces como podemos trabajar con hilos en programas gráficos?.

Antes de nada vamos a justificar, por qué necesitamos trabajar con Hilos, o tareas paralelas a la principal. Como probamos antes, si tu programa necesita hacer algo que lleva un tiempo de más de 1 segundo, y lo haces de manera sincrónica, es decir desde el hilo UI, harás que durante ese tiempo el interfaz se quede muerto (bloqueado). Podríamos de alguna manera hacer que dicho proceso se ejecutara en un hilo a parte, de manera que la ejecución del hilo principal mantuviera su dinamismo, y no quedara bloqueada. Cuando en TestHilos en las líneas 39 y 40 se inician 2 hilos secundarios, la ejecución pasa inmediatamente a la linea 45, mientras las funciones proc_hilo1() y proc_hilo2() se ejecutan en paralelo. La linea 45 bloquea el UI Thread, esperando a que el hilo1 acabe, y entonces pasa a la linea 46 bloqueando igualmente hasta que acabe el hilo2, o siguiendo si éste ya ha acabado. Este ejemplo también bloquea el UI Thread ya que espera a que los 2 hilos terminen, pero hemos podido hacer 2 procesos largos en paralelo a la vez.

Volvemos entonces al tema, de cómo hacer para no bloquear al interfaz gráfico mientras se ejecutan los hilos secundarios. Para ello, necesitaríamos poder lanzar los hilos secundarios de manera asíncrona, esto es, lanzarlos y que siga la ejecución sin bloquear del resto de instrucciones en el hilo UI. Pero entonces, ¿cómo sabemos y cuando, que nuestro hilos han terminado?. Pues bien para ello, existe un mecanismo, que es definir un evento delegado que es llamado cuando dicho hilo secundario termina, y por tanto programar un handler de dicho evento que se ejecute dentro del hilo

UI como un evento más, y por tanto sí que pueda acceder a las propiedades de cualquier componente visual del interfaz gráfico.
En el ejemplo TestBackgroundWorker, usamos un elemento gráfico llamado BackgroundWorker que desde .NET 2.0 es accesible al programador. Con él podemos, crear un hilo paralelo al hilo UI, y podemos programar 3 eventos, de los cuales 2 de ellos se ejecutarán en el hilo UI y podrán acceder libremente a los componentes visuales. De esta manera, el proceso largo podrá ejecutarse en paralelo al UI, y por tanto la ventana no quedará bloqueada. En este ejemplo, el UI lanza un hilo hijo mediante el BackgroundWorker (el método `bgWorker_DoWork()`). Éste a su vez, crea 2 hilos Thread a los que lanza y espera con Join a que acaben. Aunque el hilo hijo de UI queda bloqueado hasta que los 2 hilos hijos suyos acaben, el hilo principal UI, no queda bloqueado ya que funciona en paralelo, al ejecutar al BackgroundWorker asíncronamente.

El ejemplo Test2BackgroundWorker, hace algo más directo, esta vez usa 2 BackgroundWorker para directamente ejecutar los 2 hilos de manera asíncrona a UI. Para manejar de manera protegida a las variables compartidas por ambos hilos, hemos creado un clase llamada Counter en un fichero a parte (Counter.cs) que contiene la variable compartida por ambos hilos, counter y un Mutex que la protege durante las lecturas y escrituras en la misma. Tanto los métodos Increase() como GetCounter() son thread-safe, es decir, son seguros al acceso desde diferentes hilos, gracias al Mutex que les protege.

Vamos a tratar de cerca la composición interna de un BackgroundWorker para entender todo lo que esta pasando en estos ejemplos.

BackgroundWorker es una clase de la que se pueden crear objetos que encierren hilos que se ejecuten de manera asíncrona, paralelos al UI. Por cada hilo hijo de UI, podemos crear un objeto de esta clase. Esta clase tiene un método (el principal), que se encarga de ejecutar el hilo de manera asíncrona desde el hilo UI. Se llama *RunWorkerAsync()*, y puede llevar como parámetro un objeto con las variables que vaya a trabajar el hilo. Cuando este método es ejecutado desde código del UI, inmediatamente lanza el evento *DoWork()*, cuyo handler contendrá el código del hilo paralelo, y por tanto dentro de este código no podrá haber referencias a componentes del UI. Por eso, y puesto que el mismo componente BackgroundWorker ha sido creado en el UI, podemos acceder al mismo, dentro del código del hilo secundario extrayéndo de su referencia en el parámetro sender de dicho método. Así mismo, el parámetro pasado por *RunWorkerAsync()* al hilo secundario, ahora estaría disponible dentro de *DoWork()* en el objeto *e.Argument* y el resultado arrojado por este hilo, se puede almacenar en *e.Result* para ser devuelto al hilo principal en su momento. FullBackgroundWorker es un ejemplo completo con todos estos usos.

Esta clase, tiene medios también para llevar un seguimiento del progreso del proceso dentro del hilo, así como un medio para poder cancelarlo cuando se quiera, lo que nos da un control completo sobre el hilo secundario, desde el hilo UI. El método *CancelAsync()* llamado desde el UI nos permite enviar un mensaje al hilo, ya que éste pone la propiedad *CancellationPending* a true, y esta propiedad puede ser accedida desde dentro del hilo secundario, y en caso de ser afirmativa, se debe de poner *e.Cancel* a true dentro del hilo y salir del mismo, lo que desencadenará el evento *RunWorkerCompleted()*, con el parámetro *e.Cancelled* a true. Para poder ejecutar una cancelación de esta forma, se ha de poner la propiedad *WorkerSupportCancellation* a true.

Desde el código del hilo secundario en *DoWork()* se puede reportar al exterior el estado de lo que ocurre dentro del hilo al hilo UI, usando el método de la clase llamado *ReportProgress()* que admite hasta 2 parámetros, un entero indicando el % de progreso (de 0 a 100) y un objeto que puede llevar las variables internas compartidas del hilo. Cuando se llama a este método se lanza un evento que

ejecuta el handler *ProgressChanged()*, que recibe el progreso en *e.ProgressPercentage* y el objeto interno en *e.UserState*.

Si dentro del hilo que se ejecuta de manera asíncrona al UI, se produce una Excepción no controlada por el usuario (sin try-catch), entonces dicha Excepción es copiada al parámetro e.Error y se ejecuta el handler *RunWorkerCompleted()*, que tendrá que evaluarlo antes de nada.

Si el proceso del hilo acaba sin ser cancelado, y sin excepciones no controladas, entonces se ejecuta también el handler *RunWorkerCompleted()*.

El handler *RunWorkerCompleted()* aunque no es obligatorio de programar, sí que conviene hacerlo para controlar posibles Excepciones ocurridas dentro del hilo y sin controlar. Para ello, dentro de este handler, lo primero es revisar si *e.Result* no es *null*, en cuyo caso significaría que hubo una Excepción sin controlar en el hilo. Luego revisamos que *e.Cancelled* no este en true, lo que significaría que terminó el hilo pero por cancelación. Finalmente si ninguna de ambas cosas ha ocurrido, significa que el hilo ha terminado correctamente y podemos acceder sin miedo al resultado que se haya guardado en el objeto *e.Result*, si así se ha hecho.

Otra cuestión importante a tener en cuenta el la propiedad de la clase llamada *IsBusy*, que nos indica si el hilo esta en ejecución o esta terminado. Es importante revisar esta propiedad antes de ejecutar el método *RunWorkerAsync()*, para evitar que salte una Excepción, y que también la revisemos antes de ejecutar el método *CancelAsync()*, para que no de una Excepción por intentar cancelar algo que no se esta ejecutando.

Todos estos usos están contenidos en el ejemplo FullBackgroundWorker. Como vemos, este componente tiene todo lo necesario para lanzar hilos de manera asíncrona y totalmente controlada desde el hijo UI. Podríamos intentar crear nosotros algo similar usando las clases Thread, Task, Mutex, etc, pero nunca sería tan completo como éste, por lo que te animo a que uses la clase BackgroundWorker en tus hilos no básicos.

De igual manera, tenemos que hablar de otro de los problemas principales de los hilos, que es el acceso a la variables compartidas por varios hilos. Como hemos visto, el acceso a los componentes del UI ya no es un problema, ya que desde los handlers *ProgressChanged()* y *RunWorkerCompleted()* se puede hacer sin problemas (nunca desde *DoWork()*). Pero las variables se pueden acceder sin problemas, pero no por ello esto esta exento de problemas, ya que el acceso concurrente de varios hilos a las mismas variables, puede producir problemas en el contenido de las mismas. Para ellos te ofrecemos 2 soluciones:
1.- Usar una clase que contenga todas las variables compartidas por los hilos (Shared) y con uno o varios Mutex que las haga thread-safe.
2.- El uso de colecciones concurrentes contenidas en System.Collections.Concurrent, como ConcurrentQueue (que es como Queue pero thread-safe), o ConcurrentStack, etc. Todas ellas podríamos haberlas construido nosotros mismos como en la solución 1, pero desde .NET 4.0 ya están creadas para que las usemos sin más.

De todos los problemas a los que se enfrenta un programador con los hilos, el más peliagudo es el problema del Producer-Consumer (productor y consumidor). Imagina que tenemos una colección de datos compartida por varios hilos, y tenemos uno o varios hilos que añaden elementos a esta colección, y tenemos también uno o varios hilos que recogen elementos de esta colección. ¿Cómo hacemos para coordinar todo este follón?. Porque a no ser que el ritmo de los productores sea exactamente que el ritmo de los consumidores, lo cual es casi imposible, o se queda la colección

vacía, o se desborda. Para ello, en ese mismo namespace existe una clase contenedora llamada BlockingCollection<T> que por defecto trabaja sobre una colección ConcurrentQueue, pero que se puede definir sobre cualquiera de las otras, y que se le puede limitar en el número de elementos que contiene (BoundedCapacity). Esta clase es totalmente thread-safe y puede ser accedida desde diversos hilos a la vez sin problemas. Así mediante el método Add() se le pueden añadir elementos, y desde GetConsumingEnumerable se pueden recoger elementos conforme son accesibles, y siempre sin sobrepasar su capacidad indicada por BoundedCapacity.

En el ejemplo TestConsumerProducer, tenemos un ejemplo de uso de esta clase contenedora, donde están comentados sus pormenores. Todo este namespace se creó en .NET 4.0 por lo que incluso aplicaciones compatibles para Windows XP, pueden beneficiarse de esta programación avanzada, sin necesidad de crear a mano estructuras complejas que de seguro funcionarán peor.

Y esto es todo referente al uso genérico de la programación multi-hilo, que aunque no acaba aquí, con estos elementos hay más que suficiente para hacer buenos y potentes programas que no bloquean el UI, y que no dan problemas de concurrencia al acceder a zonas compartidas.

No obstante, no todo vale para todo, de hecho se recomienda usar BackgroundWorker solamente si vas a necesitar ser compatible con XP (.NET 4) y necesitas actualizar componentes del UI de acuerdo a cálculos u operaciones llevadas a cabo en el hilo. Si puede programar para .NET 4.5 o superior, entonces puedes usar async, wait y Task, junto con un delegado evento, tal como aparece en TestDelegadoEvento. Si no vas a necesitar actualizar componentes del UI, lo más económico en cuanto a recursos, y cantidad de código escrito, además de optimización de uso de CPUs multicores, es el uso de Task.Factory de manera a como lo hace TestTaskC4, ejemplo compatible con .NET 4.

El uso de Thread, al ser más costoso en cuanto a CPU y RAM, ha caído en desuso, en favor de de Task (TPL = Task Parallel Library). El uso de la clase estática ThreadPool, puede ser en algunos casos aceptable, aunque no si tu hilo va vivir durante toda la ejecución de tu programa. A pesar de todo BackgroundWorker sigue ocupando un lugar importante sobre todo por su compatibilidad con XP.

Happy C# Programming !!! (2018)