Hilos y Multitarea en C#

by

ArcanuS

Hola a todos. Hoy vamos a tratar un tema referente al desarrollo de software. Es algo con lo que muchos se vuelven locos y algo que muchos otros piensan que es complicadísimo de entender e implementar, y peor aun, existen "programadores" que hasta creen que es algo innecesario. Estoy hablando nada mas y nada menos que de "**Threading**" o (en criollo) "**Hilos**".

La implementación de hilos en nuestras aplicaciones, mejorarán el rendimiento y la organización a bajo nivel de las mismas.

Un Hilo, no es mas que un **Sub Proceso**. Por tanto, delegar un procedimiento o una función a un hilo, no es más que hacerla correr como un Sub Proceso del Proceso principal, que generalmente es el main de la aplicación.

La ventaja principal de trabajar de esta manera, es poder tener "Cosas" corriendo en **BackGround**. Esto es, tener por ejemplo un procedimiento corriendo, y poder seguir usando el programa principal de forma normal. Esto último lo van a entender bien cuando hagamos la práctica.

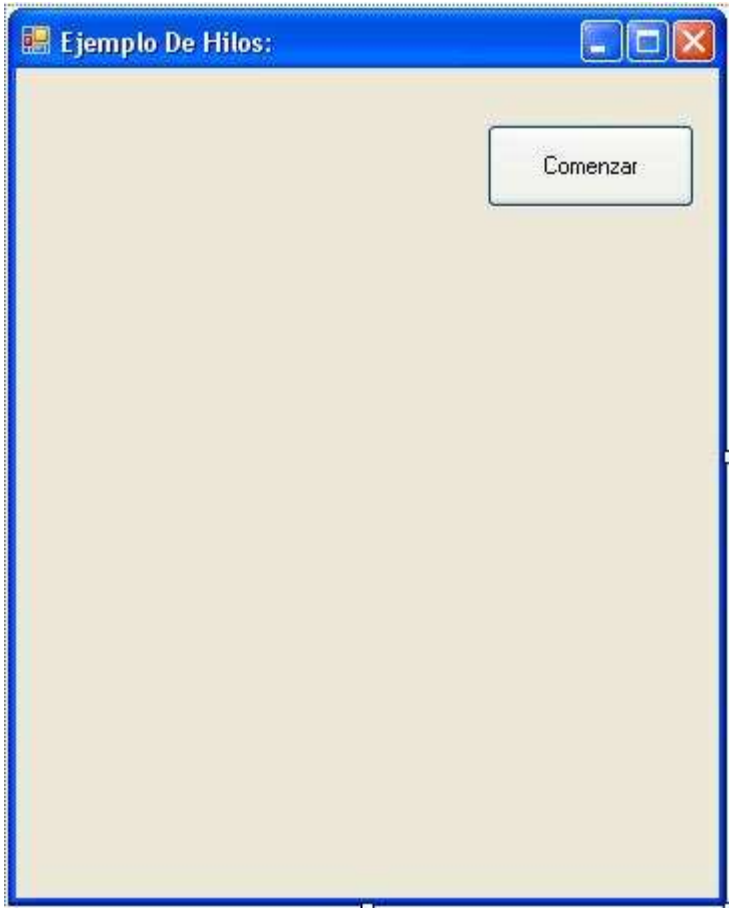
Dada esta introducción adentrémonos en el tema.

El lenguaje elegido esta vez es **C#.net** porque me parece que ya es hora de aprender un lenguaje complejo (dije complejo, no difícil) y altamente funcional.

Vamos a hacer una aplicación que cree por código múltiples barras de progreso y las valla llenando según lo indique una función.

Abrimos el Visual Studio y creamos un nuevo proyecto **C# Aplicación de Windows**.

En el formulario que se nos crea por defecto solo agregamos un botón en una esquina y ajustamos el tamaño de esta manera:



Ahora, escribimos el código del formulario para que quede así:

```
namespace EjemploHilos1
{
    public partial class Form1 : Form
    {
        int Posicion = 0;

        public Form1()
        {
            InitializeComponent();

        }

       private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void button1_Click(object sender, EventArgs e)
        {
            Posicion = Posicion + 30;

            CrearProgressBars(23,Posicion,23,200);
        }

        private void CrearProgressBars(int sX, int sY, int sAltura, int sAncho)
        {
            ProgressBar pb = new ProgressBar();
            int ContarHasta = new Random().Next(100000, 5000000);

            pb.SetBounds(sX, sY, sAncho, sAltura);
            pb.Parent = this;
            pb.CreateControl();

            for (int contador=1; contador <= 100; contador++)
            {
                pb.Value = contador;

                for (int CuentaNumeros = 0; CuentaNumeros <= ContarHasta ; CuentaNumeros++) ;
            }

        }

    }
}
```

Veamos la explicación:

Primero declaramos la variable **Posicion** y la inicializamos en cero. Esta variable se encargará de llevar la coordenada de altura que tendrá la nueva barra de progreso. Sin esta variable, todas las barras se crearían una arriba de la otra.

```
public Form1()
        {
            InitializeComponent();
        }
```

Estas líneas son propias de **Visual Studio** y se encargan de inicializar todos los componentes del formulario.

```
private void button1_Click(object sender, EventArgs e)
        {
            Posicion = Posicion + 30;

            CrearProgressBars(23,Posicion,23,200);
        }
```

Acá veremos el evento **Click** del botón. Acá hacemos que la variable global **Posicion** se incremente en 30. Las barras de progreso las crearemos con una altura de 23 unidades, por eso la nueva posición va a ser 30 unidades por debajo de la anterior barra (23 que ocupa la barra anterior y 7 de separación así no quedan pegaditas ^_^). Después llamamos a la función **CrearProgressBars()** y le pasamos por valor las coordenadas de posicionamiento en el **Form** mas el ancho y el alto.

```
private void CrearProgressBars(int sX, int sY, int sAltura, int sAncho)
        {
            ProgressBar pb = new ProgressBar();
            int ContarHasta = new Random().Next(100000, 5000000);

            pb.SetBounds(sX, sY, sAncho, sAltura);
            pb.Parent = this;
            pb.CreateControl();

            for (int contador=1; contador <= 100; contador++)
            {
                pb.Value = contador;

                for (int CuentaNumeros = 0; CuentaNumeros <= ContarHasta ; CuentaNumeros++) ;
            }

        }
```

Esta es nuestra función encargada de crear las barras. La función recibe los parámetros **sX** (coordenada X), **sY** (coordenadaY), **sAltura** y **sAncho**, todos de tipo entero.

Primero y principal declaramos e instanciamos **pb** de modo que nos quede un nuevo objeto de tipo **ProgressBar**. Luego utilizamos la propiedad **SetBounds** para establecer el posicionamiento en el formulario y le pasamos los parámetros de posicionamiento recibidos.

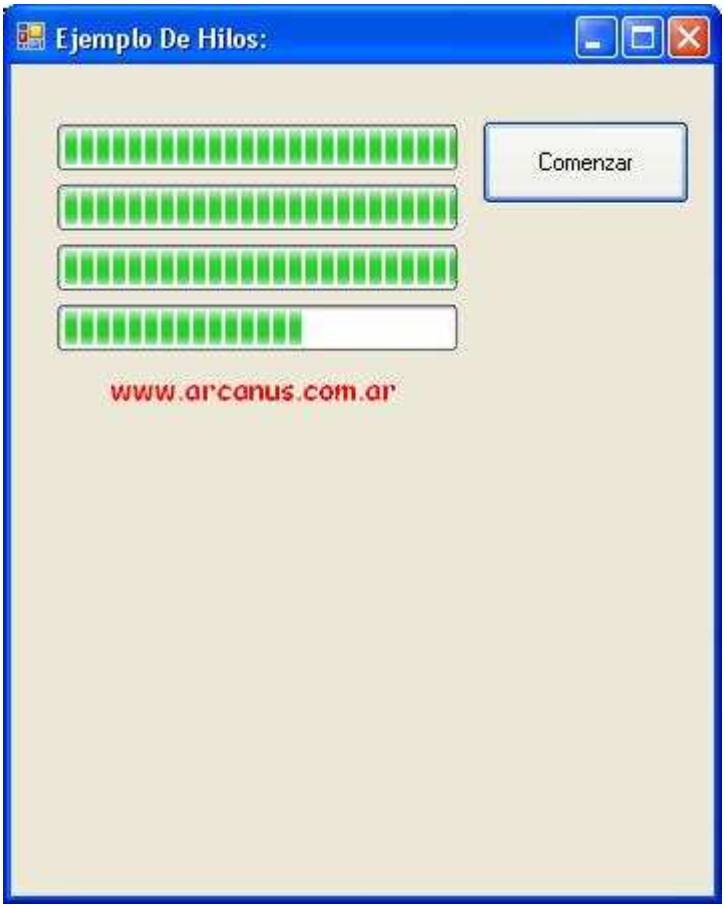
También declaramos la variable **ContarHasta** de tipo entero y le asignamos un número al azar entre cien mil y cinco millones.

Ahora le decimos a la propiedad **Parent** que el contenedor padre del control es este mismo formulario (**pb.Parent = this;**), y por último llamamos al método **CreateControl()** para que al fin nos cree el control en el formulario.

Lo siguiente es generar el retardo suficiente para llenar la barra de progreso. Hay muchas formas de hacer esto. Yo elegí hacer que el programa cuente de 0 hasta un número aleatorio alto (como mínimo 100 mil y como máximo 5 millones en mi caso), acción que según la capacidad de procesamiento de mi pobre computadora, es suficiente para que el proceso sea visible al ojo humano jejeje. Si ustedes tienen computadoras más rápidas, aumenten esta cantidad hasta que se ajuste a sus requerimientos.

Para esto hice dos bucles **For**. el primero declara una variable **contador** de tipo **entero**, la inicializa en uno y se va a repetir hasta que la variable llegue a 100 incrementándola de uno en uno. En cada repetición aumenta el valor de la propiedad **Value** de la barra, (que por defecto es cero) en 1, (Esto es lo que hará que vallan pasando las rayitas en la barra ^_^). El segundo bucle **For** es el que hace el trabajo sucio. Declara la variable **CuentaNumeros** de tipo **entero**, la inicializa en cero y se va a repetir hasta que la variable llegue al valor aleatorio que tomó la variable **ContarHasta** al principio de la función.

Bien, es hora de ejecutar el programa y ver los flamantes resultados. Ejecutemos y presionemos varias veces sobre el botón para ver que pasa.



Y???, ¿cómo les fue? Chan Chan!. La aplicación funciona, pero de una manera "**poco ética**" diría **Richard Stallman**. Si se fijaron bien, por más que ustedes hacían repetidos **clics** sobre el botón, la aplicación estaba como trabada, y solo creaba la nueva barra una vez que la anterior terminaba de llenarse y no instantáneamente cuando ustedes hacían el clic. Si no te diste cuenta, ejecutá de nuevo el programa y volvé a probar.

Esto sucede porque todo corre en un solo proceso: el del programa en si. Por tanto, el proceso está activo, pero trabajando. Es por eso que perdemos el control de la aplicación. solo cuando la función **CreateProgressBars()** finaliza, el programa retorna el control al usuario.

Este mismo escenario, trasladado al desarrollo real se suele ver muy frecuentemente en aplicaciones que utilizan **sockets** o que realizan grandes transacciones a servidores de bases de datos en donde los tiempos de red no son siempre los que uno quisiera. La mayoría de estos casos a gran escala suelen terminar en un cuelgue generalizado de la aplicación y la perdida de información que esto conlleva.

Para evitar todos estos dolores de cabeza es que se usan los **Hilos**. Y como dijo un buen amigo, "*El buen manejo de hilos es lo que separa a un Programador de un Aprendiz de coder*".

Quiero aclarar que esto de los hilos no es nada nuevo, ni tampoco es nada que **Microsoft** descubrió y que va a salvar al mundo ni nada de eso. La gente que programa en **C** o algunos de **C++**, vienen oyendo palabras como **Hilos**, **sub procesos**, **delegados**, etc desde hace mucho tiempo.

Lo novedoso, si se quiere, son las clases que el **framework.net** ofrece para el manejo de **hilos**, **invokes** y **delegados**. Están muy pulidas y constan de una gran abstracción, lo que facilita mucho el trabajo y el entendimiento de los mismos.

Todas las clases y SubClases referentes a hilos se encuentran en el espacio de nombres **System.Threading**, por lo tanto debemos importarlo al principio del código.

```
using System.Threading;
```

El **FrameWor**k, también maneja los procesos y subProcesos de una forma especial, y no como simples procesos del sistema. Digo especial porque constan de restricciones de seguridad para evitar un mal uso de la sintaxis como sería puntear algo fuera de la memoria reservada del **framework** o acceder a un objeto situado en un proceso diferente. Pero bueno mejor no sigo que no quiero que se mareen. Sigamos. Lo único que me interesa que sepan es que para el **FrameWork**, los procesos "**Desprendidos**" del proceso principal son **SubProcesos** del mismo, (y no procesos independientes), y el sistema está enterado de esto. Cuando el proceso principal muera, morirán también todos sus **SubProcesos**, por más que se encuentren en el medio de la ejecución. Y si queda algo en memoria, ya se encargará el **GarbageCollector**. Ahora si, sigamos jejeje.

Ustedes me van a putear, pero después de pensarlo mucho decidí no usar el código antes visto para seguir explicando el uso de hilos, ya que decidí no meterme en temas muy profundos como son los **delegados**, los **handlers** y los **eventos** porque ya se perdería el enfoque de este tutorial. Mejor vamos a hacer un código nuevo y ver bien gráficamente el trabajo con hilos. Mejor usemos el código anterior para ver lo que sucede con un programa normal cuando ejecutamos múltiples acciones sobre el mismo proceso (se cuelga).

Lo que sucede es lo siguiente. Las barras de progreso son creadas desde el proceso principal, pero para ir llenándose necesitan ser accedidas desde el hilo que se encargue de esa tarea. Ahora el problema está en que a partir del **framework 3.0** un objeto solo puede ser accedido solo desde el mismo proceso que lo creó. Por tanto habría que crear un **delegado** que obtenga los datos del estado de la barra, se los pase a un **handler** y este los traiga a una función dentro del proceso principal que se encargue de actualizar la barra. No es que sea algo del otro mundo, pero si es algo demasiado avanzado para este tutorial, pero lo dejo pendiente para los próximos!!!

Bueno, vamos a crear una nueva aplicación de consola. Una vez que la tengamos le escribimos el siguiente código:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace EjemploHilos
{
    class Program
    {
        static void Main(string[] args)
        {
            Proceso oProceso = new Proceso();

            Console.WriteLine("Contando PARES en el proceso principal: ");
            Console.WriteLine("----------------------------------------");
            Console.WriteLine();

            oProceso.ContarNumerosPares();

            Console.WriteLine("Contando IMPARES en el proceso principal: ");
```

```
            Console.WriteLine("----------------------------------------");
            Console.WriteLine();

            oProceso.ContarNumerosImpares();

            Console.WriteLine("Contando PARES desde un hilo: ");
            Console.WriteLine("----------------------------");
            Console.WriteLine();

            Thread HiloPar = new Thread(oProceso.ContarNumerosPares);
            HiloPar.IsBackground = true;
            HiloPar.Start();
            HiloPar.Join();

            Console.WriteLine("Contando IMPARES desde un hilo: ");
            Console.WriteLine("------------------------------");
            Console.WriteLine();

            Thread HiloImpar = new Thread(oProceso.ContarNumerosImpares);
            HiloImpar.IsBackground = true;
            HiloImpar.Start();
            HiloImpar.Join();

            Console.WriteLine("Lanzando los dos hilos al mismo tiempo: ");
            Console.WriteLine("---------------------------------------");
            Console.WriteLine();

            Thread nuevoHiloPar = new Thread(oProceso.ContarNumerosPares);
            Thread nuevoHiloImpar = new Thread(oProceso.ContarNumerosImpares);

            nuevoHiloPar.Start();
            nuevoHiloImpar.Start();

            Console.ReadLine();
        }
    }

    class Proceso
    {
        public void ContarNumerosPares()
        {
            for (int numero = 0; numero <= 100; numero = numero + 2)
            {
                Console.Write(numero + " ");
                for (int GranContador = 0; GranContador <= 20000000; GranContador++) ;
            }

            Console.WriteLine();
            Console.WriteLine();
        }
        public void ContarNumerosImpares()
        {
            for (int numero = 1; numero <= 100; numero = numero + 2)
            {
                Console.Write(numero + " ");
                for (int GranContador = 0; GranContador <= 20000000; GranContador++) ;
            }

            Console.WriteLine();
            Console.WriteLine();
        }
    }
}
```

Bueno, desarmemos esto. Fíjense que tenemos creadas dos clases: La principal (**class Program**) y la **clase Proceso** (**class Proceso**).

```
Proceso oProceso = new Proceso();

            Console.WriteLine("Contando PARES en el proceso principal: ");
            Console.WriteLine("---------------------------------------");
            Console.WriteLine();

            oProceso.ContarNumerosPares();
```

Primero creo un objeto llamado **oProceso** y lo instancio. Las siguientes 3 líneas escriben en pantalla lo que está entre comillas. Por último lanzo el método **ContarNumerosPares()**. Aquí estaremos lanzando el método desde el proceso principal.

```
            Console.WriteLine("Contando IMPARES en el proceso principal: ");
            Console.WriteLine("----------------------------------------");
            Console.WriteLine();
            oProceso.ContarNumerosImpares();
```

Lo mismo que antes pero contando números impares.

```
            Console.WriteLine("Contando PARES desde un hilo: ");
            Console.WriteLine("---------------------------");
            Console.WriteLine();

            Thread HiloPar = new Thread(oProceso.ContarNumerosPares);
            HiloPar.IsBackground = true;
            HiloPar.Start();
            HiloPar.Join();
```

Acá es donde comienza la fiesta. Primero volvemos a escribir en pantalla las cadenas de texto. Ahora procedemos a declarar e instanciar el hilo, llamado **HiloPar**. Para esto utilizo la clase **System.Threading.Thread**. Fíjense que cuando la instancio le paso por parámetro el nombre del método que voy a lanzar en el hilo. En **Vb.net** es lo mismo solo que le paso la dirección de memoria del método en cuestión usando **AddressOf**. Luego establezco la propiedad **IsBackground** a **True**. Esto hace que el hilo pase a trabajar en **segundo plano**. Después lanzo el hilo (**HiloPar.Start()**). El método **Start()**, permite pasar un parámetro siempre y cuando este sea de tipo **object**. Podemos pasar un número por ejemplo pero de tipo **objeto**, que luego habrá que **castear** a su tipo dentro del hilo. Pasar parámetros a un hilo es algo que también veremos en próximos tutórales. Finalmente, y con el hilo ya lanzado, ejecuto el método **Join()**. Esta acción hace que no se devuelva el control al proceso principal, hasta que el subproceso termine. Después prueben ejecutar el programa comentando todas las líneas **Join()** para que vean lo que sucede.

```
            Console.WriteLine("Contando IMPARES desde un hilo: ");
            Console.WriteLine("-----------------------------");
            Console.WriteLine();

            Thread HiloImpar = new Thread(oProceso.ContarNumerosImpares);
            HiloImpar.IsBackground = true;
            HiloImpar.Start();
            HiloImpar.Join();
```

Lo mismo que antes pero para el método **ContarNumerosImpares().**

Solo nos queda desarmar la clase Proceso. Veámosla de nuevo:

```
    class Proceso
    {
        public void ContarNumerosPares()
        {
            for (int numero = 0; numero <= 100; numero = numero + 2)
            {
                Console.Write(numero + " ");
                for (int GranContador = 0; GranContador <= 20000000;
GranContador++) ;
            }

            Console.WriteLine();
            Console.WriteLine();
        }
```

Acá definimos el método **ContarNumerosPares()**. Dentro de el tenemos nuestros queridos bucles **For**. Al igual que en el código anterior, uno cuenta de 0 a 100 y en cada repetición escribe el valor en pantalla y después ejecuta al bucle que hace el trabajo sucio: Contar de 0 a 20000000. De esta forma relentizamos el programa así podemos verlo. De lo contrario no alcanzaríamos a ver nada, salvo que estuviéramos con una commodore64, pero si estuviéramos con una commodore64 no estaríamos programando en C#.net, uffff ya se me saltó la chapa. (Ida de bola de ArcanuS .....). Mejor sigamos.

Las dos líneas Console.WriteLine(); escriben un salto de línea en la consola.

```
        public void ContarNumerosImpares()
        {
            for (int numero = 1; numero <= 100; numero = numero + 2)
            {
                Console.Write(numero + " ");
                for (int GranContador = 0; GranContador <= 20000000;
GranContador++) ;
            }

            Console.WriteLine();
            Console.WriteLine();
        }
    }
```

Lo mismo, solo que es el método **ContarNumerosImpares()**.

```
            Console.WriteLine("Lanzando los dos hilos al mismo tiempo: ");
            Console.WriteLine("---------------------------------------");
            Console.WriteLine();

            Thread nuevoHiloPar = new Thread(oProceso.ContarNumerosPares);
            Thread nuevoHiloImpar = new Thread(oProceso.ContarNumerosImpares);

            nuevoHiloPar.Start();
            nuevoHiloImpar.Start();

            Console.ReadLine();
```

Ahora vamos a lanzar los dos hilos al mismo tiempo y veremos como es que funciona esto de la multitarea. volvemos a crear 2 hilos nuevos apuntando a los métodos **ContarNumerosPares()** y **ContarNumerosImpares()**. Te preguntarás por qué volvemos a crear los hilos si ya están creados e instanciados. Como estamos en una aplicación de consola, la ejecución aquí es puramente lineal, y como algunos sabrán, un proceso una vez que muere no se puede reiniciar. Una vez muerto se descarga de la memoria y hasta la vista proceso. Si estuviéramos en una aplicación **Windows Forms**, y tuviéramos la

creación de los hilos dentro del evento **click** de algún botón, no habría problema con usar los mismos hilos, ya que cada vez que se presione el botón se estarían creando nuevos hilos (por tanto en realidad no estaríamos usando "los mismos hilos" [pequeños engaños del framework xDDD]). En fin, volvemos a crear e instanciar los hilos. Luego los lanzamos uno atrás de otro. Por último escribimos **Console.ReadLine()**; para que la consola no se cierre hasta que no presionemos una tecla.

Ahora si, sentate, ponete cómodo, preparate el mate y ejecutá el programa. La salida debería ser similar a esta:

Y con esto, estaríamos terminando este tutorial sobre hilos y multitarea en C#.net, no sin antes aclarar algunas cosillas que quedaron en el tintero.

La MultiTarea que obtenemos en este caso no es una Multitarea real, al menos en mi caso ya que tengo una computadora con un solo procesador. Salvo que ustedes tengan una maquina con mas de un procesador, obtendrán una multitarea simulada.

Permítanme aquí hablar un poquito de un tema avanzado pero que es bueno saberlo. Quiero aclarar que para lo próximo que van a leer, me basé en las opiniones vertidas por **Tom Archer** en su libro "**A fondo C#**" de **Microsoft Press**.

Multitarea y Alternancia de Contexto (Context Switching):

Al principio de este turorial dijimos que un Hilo es simplemente un subproceso de un proceso principal o creador. Permítanme ahora, que sabemos un poco mas del tema, modificar un poco esta definición.

Un hilo de ejecución es una **unidad de procesamiento**, y la multitarea consiste en la ejecución de múltiples hilos en simultáneo. Esta posee dos ramas principales: **Multitarea Cooperativa** y **Multitarea Preferente**.

Si recordamos versiones antiguas de **Windows** como **Windows 3.11**, veremos que este poseía **Multitarea Cooperativa**, en donde cada hilo de ejecución era el responsable de renunciar al control del procesador para que este pudiera atender a otros hilos. En mi experiencia personal, la primera vez en mi vida que vi. multitarea preferente fue hace mucho tiempo, cuando tenia al rededor de 8 o 9 años y empecé a usar **OS/2** y a hacer mis primeras aplicaciones en **REXX** (**Lenguaje de scripting del sistema Os/2**). Después ya vino **Windows NT**, **95**, **98**, **2000**, etc que traían la misma multitarea preferente de **Os/2** (raro Microsoft afanando algo no?).

Con la multitarea preferente, el procesador es el encargado de asignar al proceso una cierta cantidad de tiempo en la que ejecutarse. Un **'quantum'** diría mi profesor de Procesamiento Distribuido, pero yo, porfiado como soy, prefiero llamarle **TimeSlice**.

Entonces, el procesador se encargará de darle a cada proceso su **timeslice** de forma alternativa, así el programador no tiene que preocuparse por cuando recibe el control el proceso y cuando lo pierde ni de cuando ceder el control para que el resto de los hilos puedan ejecutarse. Vale aclarar que **.NET** solo puede ejecutarse en sistemas con **Multitarea Preferente**. Creo que ya me entendieron por qué hablo de una multitarea "simulada". Si nuestra aplicación corre sobre una maquina con un solo procesador, este está simplemente alternando los hilos en fracciones de milisegundos, por lo cual tenemos una sensación de multitarea. Si queremos tener una multitarea de verdad, necesitaremos desarrollar y correr nuestra aplicación en una maquina con múltiples procesadores.

La **Alternancia De Contexto** es algo que siempre está ahí, pero que no se ve, y a su vez es un concepto un poco difícil de asimilar para algunos.

El procesador, utiliza un temporizador de hardware para saber cuando ha terminado una fracción de tiempo (**TimeSlice**) para un proceso determinado. Cuando el temporizador lanza la interrupción, el procesador salva en la pila el estado de los registros del hilo de ejecución en curso. Una vez hecho esto, mueve esos datos a una estructura de datos denominada **estructura CONTEXT**. Cuando el procesador quiere retornar a un hilo de ejecución previamente ejecutado, deshace la operación anterior y recupera los valores existentes en la **estructura CONTEXT**, depositándolos nuevamente en los registros. El conjunto de todas estas operaciones se llama: **Alternancia de Contexto**.

En fin, no sigo mas porque si no los voy a matar del aburrimiento. Con esto me despido y espero que este tutorial le halla servido a alguien. Por favor, dejen sus comentarios en este post.

Para este tutorial que me llevó dos días armar, quiero agradecer a dos personas en especial.

Gracias a **Néstor V**. Un compañero de trabajo de mi madre quien inconscientemente sembró en mi la curiosidad por la programación. fue él además quien me prestó el CD de **Visual Basic 5** para que yo lo instale en mi computadora y comience a jugar con el lenguaje a la edad de mas o menos 12 años (allá por el año 1998).

La segunda persona a la que quiero agradecer es a **Melina T**. fue mi profesora de programación en lenguaje **Vb 6** del único curso de programación que tomé en mi vida. Fue por el año 2000 mas o menos. **Melina** fue como una musa inspiradora, (hablando siempre de programación, aunque también era muy muy linda), durante los 11 meses que duró el curso. Gracias Meli por ayudarme a sentar las bases en ese lenguaje y en mi cabeza, bases validas para todo tipo de lenguajes, y por enseñarme a razonar de forma lógica y matemática.

Solo resta agradecerles a ustedes por dejarme expresarme en estos tutoriales. El código fuente de los dos ejemplos lo dejo acá. Y también dejo la version en pdf de este tutorial.

Hasta la próxima.

by ArcanuS