Multihilos en .Net Compact Framework

Hilos en Windows CE

- Windows CE soporta múltiples *hilos* de ejecución para cada proceso.
- Un *hilo* es una unidad de ejecución dentro de un proceso. Cada proceso puede tener varios hilos. Los hilos reciben una porción del tiempo de CPU.
- Todos los hilos de un proceso comparten el mismo espacio de direccionamiento virtual, luego un hilo puede destruir los datos de otro hilo del mismo proceso.
- Consideraciones a tener en cuenta en Windows CE:
  - El máximo número de procesos que soporta Windows CE es 32. Algunos procesos del sistema están ejecutándose, por lo que hay que *ahorrar* al máximo en el número de procesos lanzados.
  - Cada proceso recibe un máximo de 32 MB de memoria.
  - Es mucho más económico (en consumo de recursos) crear un nuevo hilo que un nuevo proceso, ya que no se necesita un nuevo espacio de direccionamiento.
  - Aún así, crear hilos debería de ser considerado siempre *caro* en términos de recursos.

La clase `Thread`.

- La clase que encapsula toda la funcionalidad de los hilos es **`System.Threading.Thread`**.
- Los métodos de esta clase son:

Método	Descripción
`Thread`	Constructor
`Priority`	Pone o consulta la prioridad de un hilo
`Void Start`	Arranca un hilo recién creado o parado.
`Void Sleep (int tiempo)`	Suspende la ejecución de un hilo durante al menos `tiempo` milisegundos.
`Static Thread CurrentThread`	Devuelve el hilo que está en ejecución.

Creación e inicio de un hilo

1. Se crea una instancia de **`System.Threading.ThreadStart`**. Le pasamos el nombre del método (función) que queremos que se ejecute en el hilo. Este método debe ser `void` y no recibir parámetros de entrada.
2. Se crea una instancia de **`System.Threading.Thread`**. Se le pasa al constructor la clase Threadstart creada antes. Ahora tenemos una referencia al hilo que ejecutará el método indicado.
3. Se llama al método Thread.Start() en la referencia del hilo creada antes. Ahora el hilo comienza a ejecutarse.

```
System.Threading.ThreadStart mi_arrancahilos;
System.Threading.Thread mi_hilo;

mi_arrancahilos = new
        System.Threading.ThreadStart(lafuncion);
mi_hilo = new System.Threading.Thread(mi_arrancahilos);

mi_hilo.Start();
```

Parada de un hilo

- Para suspender temporalmente la ejecución de un hilo se invoca el método **Thread.Sleep()**. Este método detiene la ejecución del hilo en que es invocado.
- El método recibe como argumento el número de milisegundos que deberá pasar en la cola de espera de acceso a la CPU. El tiempo hasta su ejecución puede ser mayor. Un valor de 0 hace que el hilo abandone la CPU pero se coloque en la cola inmediatamente.

```
Thread.Sleep(200);
```

- Los controles no pueden ser actualizados desde hilos que no sean propietarios del control.
- Para modificar los controles desde los hilos es necesario crear un delegado para un método que realice las modificaciones. No admite parámetros. Se usa el método **Control.Invoke(delegado)**.

- Un aplicación no termina su ejecución hasta que todos los hilos que la componen han terminado.
- Hay que tener cuidado con hilos en bucles de ejecución que, aparentemente no hacen nada, pero impiden a una aplicación terminar.

Sincronización de hilos. `Mutex`

- El .Net Compact Framework incluye la clase **`System.Threading.Mutex`** que permite proteger áreas de código frente a múltiples hilos.
- El *cerrojo* permite que un único hilo lo adquiera. Si otro intenta adquirirlo, se bloquea hasta que quede libre. De esta manera, sólo un hilo ejecuta el código protegido.
- El modo de trabajo es el siguiente:
  1. Crear una instancia de la clase **`Mutex`** pasando **`false`** al constructor para que el código que lo crea no lo adquiera.
  2. Al principio del código, llamar al método **`Mutex.WaitOne()`** que permite que un único hilo ejecute el código.
  3. Al final del código, colocar una llamada **`Mutex.ReleaseMutex()`** para liberar el cerrojo.

Ejemplo:

```
Mi_cerrojo = new System.Threading.Mutex(false);

Mi_cerrojo.WaitOne();
// Este código sólo lo ejecutará un único hilo

Mi_cerrojo.ReleaseMutex();
```

Sincronización de hilos. `Monitor`

- El .Net Compact Framework también soporta el uso de monitores.
- La clase estática **System.Threading.Monitor** tiene dos métodos: **Enter()** y **Exit()** que podemos usar para sincronizar hilos.
- El monitor se utiliza para proteger objetos. Cuando un hilo accede al objeto protegido por el monitor, el planificador del sistema operativo no permite que otro hilo lo adquiera.
- El uso de la clase monitor es como sigue:
  1. Para acceder a la zona protegida, todos los hilos deben hacer una llamada al método **System.Threading.Monitor.Enter()**. Recibe como parámetro una referencia al objeto a proteger.
  2. El hilo que adquiere el acceso, accede al objeto. Debería de tener el bloqueo el menor tiempo posible, ya que los demás hilos permanecen bloqueados.
  3. Cuando el hilo termina, libera el objeto protegido haciendo una llamada a **System.Threading.Monitor.Exit()**. Hay que pasar como parámetro una referencia al objeto liberado.

Creación de un *Pool* de hilos

- Crear y destruir una gran cantidad de hilos para realizar pequeñas tareas es muy costoso en términos de recursos.
- Para evitarlo, se puede usar la clase **System.Threading.ThreadPool** que crea un almacén de hilos que reutiliza después de usados sin eliminarlos.
- La clase se utiliza de la siguiente manera:
  - Se escriben las funciones que queremos que se ejecuten por parte del almacén de hilos. Deben retornar `void` y, como mucho, recibir un parámetro de tipo `object`.
  - Se crea una instancia de la clase **System.Threading.WaitCallback** pasándole el nombre del método anterior como parámetro.
  - Se llama al método estático **System.Threading.ThreadPool.QueueUserWorkItem** pasándole la instancia `WaitCallback`, así como el parámetro que reciba el método a ejecutar (si lo tiene).

Ejemplo:

```
System.Threading.WaitCallback wb= new
                    System.Threading.WaitCallback(lallamada);

System.Threading.ThreadPool.QueueUserWorkItem(wb, (object)" Cadena
de          caracteres");

private void lallamada(object parametro)
```

Actualización segura de variables

- Para actualizar variables de manera segura utilizando hilos se puede usar la clase **System.Threading.Interlocked**.
- Cuando se usa esta clase, la actualización de la variable se realiza de manera atómica. La CPU no puede ser reclamada por ningún otro proceso o hilo mientras se realiza la actualización.
- Incrementar:
  - **System.Threading.Interlocked.Increment(ref lavariable)**;
- Decrementar:
  - **System.Threading.Interlocked.Decrement(ref lavariable)**;
- Establecer un valor:
  - **System.Threading.Interlocked.Exchange(ref lavariable, int valor)**;
- Comparar y cambiar:
  - **System.Threading.Interlocked.CompareExchange(ref lavariable, int valor, int comparador)**;

```
(if ( lavariable == comparador) lavariable= valor;)
```