TThread: Introducción

Iniciamos una pequeña serie sobre la clase TThread, que nos va a permitir adentrarnos en lo que se viene a llamar por los especialistas más recurridos: El mundo de las "Aplicaciones multihilo".

Sirvan estas primeras lineas y nos ayuden a sentirnos un poco más cómodos; a romper el hielo en esta serie que iniciamos ahora y que nos introducirá en el mundo de las aplicaciones multihilo. Vamos a intentar, no lo dudéis, que el pequeño viaje que ahora comenzamos nos traiga nuevas inquietudes y nuevos deseos de curiosear, y sobretodo de aprender. ¿De mí...? No, no, por favor... Yo tengo las mismas inquietudes que vosotros y tan solo me limito a compartirlas desde esta humilde participación. Aprenderemos más de nosotros mismos si tras la lectura intentáramos extraer nuestras propias conclusiones, que con toda seguridad puedan ser más certeras que las mías. ¿Equivocarse...? Posiblemente... ¿y...? Errar sin duda forma parte del proceso del aprendizaje.

No hay mayor pretensión y quizás se pueda ver como un atrevimiento este deseo de haceos participes del devenir de la serie. Es más, os invito a que juntos hagamos el camino y que si así lo queréis podáis sugerir, comentar, criticar, apoyar cualquier cosa que aquí se pueda decir. Como ya tantas veces os he comentado no es mi artículo sino el vuestro. ¿Os atreveréis a participar?

Tras una interminable serie sobre los Objetos Auxiliares, objetos que en una gran mayoría procedían del estudio del módulo *Classes.pas*, volvemos para sumergirnos en otra de las clases pertenecientes al mismo módulo y que nos hace adivinar, a poco que curioseemos en él, su importancia dentro del conjunto que forma la librería de componentes visuales (VCL). Efectivamente, estamos hablando de la aparición de clases situadas en la parte alta del árbol jerárquico que construye la VCL, tales como las que implementan las listas más básicas, flujos de bits y persistencia, colecciones de objetos, ficheros, hilos de ejecución y el mismísimo TComponent, ascendiente de todas las clases que se integran en la paleta de componentes en el IDE. No puede pasar desapercibida, a todas luces, su importancia. Si hablamos de Delphi 5 y queremos localizar el código fuente nos dirigiremos al directorio de instalación de Delphi, concretamente dentro de la carpeta "..*Source**VCL*\". Si hablamos de Delphi 6, con la introducción de la CLX bajo el nuevo contexto del SO de Linux, la podremos localizar en "..*Source**Rtl**Common*\". En cierta forma esta nueva localización ya nos anuncian algunos cambios. Algunos serán perceptibles rápidamente:

```
unit Classes;

{$R-,T-,X+,H+,B-}

{$IFDEF MSWINDOWS}
{ ACTIVEX.HPP is not required by CLASSES.HPP }
(*$NOINCLUDE ActiveX*)
{$ENDIF}
{$IFDEF LINUX}
{$DEFINE _WIN32}
{$ENDIF}
{$IFDEF MSWINDOWS}
{$DEFINE _WIN32}
```

```
{$ENDIF}

interface

{$IFDEF MSWINDOWS}
uses Windows, Messages, SysUtils, Variants, TypInfo, ActiveX;
{$ENDIF}
{$IFDEF LINUX}
uses Libc, SysUtils, Variants, TypInfo, Types;
{$ENDIF}
...
```

La introducción de Linux como destino potencial del código que implementemos en nuestros desarrollos con Delphi 6, introduce en algunos de los módulos básicos, como puede ser *Classes.pas*, cambios sustanciales: sea ejemplo de esto la inserción de directivas de compilación que nos permitan hacer compatible el código fuente de algunos de los módulos para ser compilados por ambas plataformas y de esa forma hacer efectiva la promesa que Borland nos hace con la salida al mercado de Delphi 6 y Kylix. Pero no nos adelantemos. Si en un principio me movía la idea de ver la clase TThread desde la óptica de Delphi 5, quizás me parezca de mayor interés que pueda ser abordada desde la implementación que se hace en la versión 6 del compilador.

Puede ser este, un buen momento para comentar el interés en la lectura del artículo de Jose Manuel Navarro en el número 10 de esta revista, “Los rincones del API de Win32. Memoria Virtual”. Entiendo que la serie que ha comenzado mi compañero en el número anterior, complementa el contenido de ésta y viceversa. Encontraremos en su artículo un acercamiento al modelo de memoria en Windows. El artículo pienso que os resultará ameno y sobretodo didáctico, que es al fin y al cabo lo que pretendemos al iniciarlos. Es inevitable al final, que se aborden algunos detalles inherentes al funcionamiento del SO desde nuestra perspectiva, si pretendemos la idea final de entender y saber valorar la incorporación de nuevos hilos de ejecución, paralelos al hilo principal de nuestra aplicación, en su justa medida, su problemática, sus ventajas y desventajas. Bueno... vayamos con calma y buenos alimentos, como diría mi padre.

Os comento aproximadamente cual será el contenido de la serie: Este primer artículo será breve y la idea que persigue es tan solo presentar los personajes que conforman el escenario. No podemos entrar, lógicamente en demasiadas profundidades y me parece al final suficiente enfocar algunas ideas importantes, dejando los matices para un momento posterior y siempre al hilo del desarrollo del artículo. Mas allá de esta presentación o introducción al tema perseguiríamos ver como se ha implementado la clase TThread en Delphi 6 y que, a buen seguro, será motivo de al menos un capítulo más. Esto nos obligará a entrar de lleno en el análisis del código, que intentaremos hacer con el máximo detalle, tal y como nos hemos comprometido. Buscaremos además no solo la visión desde el SO de Windows sino también desde la de Linux.

Nos quedará por ver todo aquello que hace referencia a la sincronización de múltiples hilos: semáforos, mútex, eventos, y demás objetos que el sistema operativo proporciona con motivo de la protección de los recursos. También sería una buena idea aderezarlos con algunos ejemplos. Y quizás ya para finalizar, intentar el desarrollo de algún pequeño programa, sencillo, abordando de una forma más real los conceptos estudiados. Si seguisteis alguno de vosotros la serie anterior sobre Objetos Auxiliares apreciaréis que mantiene cierto paralelismo con ella.

En fin. Tengamos en cuenta el guión y dejemos la suficiente libertad para acercarnos a vuestro interés y vuestras inquietudes.

Multitarea... las apariencias engañan.

Ejecutamos nuestro cliente de correo, vemos las noticias en nuestro navegador preferido mientras calculamos y acumulamos dígitos monetarios en la hoja de calculo, y resuena el altavoz con los sonidos de la última descarga de mp3... Ante nuestros ojos quedan abiertas varias ventanas que representan físicamente una serie de procesos que se ejecutan simultáneamente ante nuestros ojos. Vale... vale... ya sabéis por donde voy mas o menos. Apariencia o realidad...

Lejos están ya los tiempos en los que el pobre usuario se obligaba a finalizar la aplicación actual para dar inicio a una nueva ejecución. Era un paso obligado en la evolución, casi intuitivo. Alguien pensó que era posible ejecutar varios procesos simultáneos, compartiendo tiempo de una CPU que a fuerza de rapidez, nos daba una realidad inexistente: la mágica sensación de que muchas aplicaciones eran ejecutadas al tiempo. La multitarea se nos presenta en principio bajo dos formas distintas: multitarea cooperativa y multitarea con derecho preferente o preventiva. Windows 3.1 es un ejemplo de entorno de ***multitarea cooperativo***. En ese momento no existen los hilos de ejecución, cosa sin duda de interés si todavía se pretende un desarrollo compatible con dicho SO y que lógicamente deberemos de tener en cuenta.

Hablaremos siempre de equipos informáticos que se componen de un único procesador. La multitarea puede ser algo real si hablamos de ordenadores con mas de una CPU pero la idea queda ahí y en lo que respecta a nuestro artículo queda en un segundo plano: Existen sistemas operativos, recientes o no, "actuales" o de "moda" como Linux, o cualquiera de los Windows, que permiten con un solo procesador simular un entorno de multitarea. Y de entre estos, algunos como Windows NT, Windows 2000 o Linux, permiten el multiproceso real con la incorporación de dos o mas procesadores.

Pero ¿que significaba la denominación de entorno de ***multitarea cooperativo***? El término de cooperativo no es gratuito ciertamente. Todos los procesos en ejecución debían de colaborar cediendo tiempo al resto de procesos, que esperaban pacientes a que el planificador del SO entregara el control. Cada proceso debía de ceder el control de la CPU al Planificador del Sistema Operativo. Y este cederlo de nuevo al siguiente proceso, y así sucesivamente, colaborando todos en el reparto del tiempo de ejecución. Es decir, ¡cruda realidad! que si un proceso concreto, de forma egoísta quedaba muerto, podría inevitablemente hacer caer todo el Sistema. No deja de ser gracioso que tal requisito, saludable, pudiera hacer escribir con mayor cuidado el código de los desarrollos, dado que todos los procesos se veían obligados a colaborar o cooperar en la estabilidad del sistema.

Olvidándonos de los sistemas operativos de tiempo Real, nuestro punto de parada final son los entornos denominados de ***multitarea preventiva o forzada***, que vienen a coincidir con la inmensa mayoría que nos resultan familiares por su proximidad temporal. También he encontrado en otras documentaciones como denominación la de ***multitarea con derecho preferente***. Estos sistemas operativos coinciden con toda la familia de 32 bits de Windows (Windows 9x, Windows NT, W2000...) o Linux, por nombrar tan solo aquellos de los que vamos a centrar. Realmente Windows 9x es un entorno mixto 16/32 bits pero a nuestros efectos no es relevante y tan solo hacemos referencia a su dimensión como sistema operativo de 32 bits. Dejemos esos matices a un lado. Al igual que en los entornos de multitarea cooperativos los distintos procesos en ejecución comparten tiempo de CPU, pero en éstos finalmente es el propio Sistema Operativo el que distribuye y reparte de los ciclos de proceso en razón de determinados algoritmos que permiten optimizar el tiempo de ejecución. Es *forzado* porque el control lo toma el Sistema Operativo y es el quien decide en cada momento que proceso debe activar el Planificador y cual debe quedar a la espera de ser introducido nuevamente.

Esto ya nos acerca frente a algunos conceptos que abordaremos al centrarnos en los hilos de ejecución y que tienen relación precisamente en la prioridad asignada. Distintos niveles de prioridad nos permitirán que determinadas tareas se ejecuten con mayor celeridad que otras menos prioritarias. Es decir que aquellos procesos con mayor nivel de prioridad tendrán la oportunidad de disfrutar de mayor cantidad de ciclos de nuestra castigada CPU. Y con respecto a la creación de nuevas hebras o hilos de ejecución, han de disfrutar

o padecer similar situación y seremos capaces de asignar distintos niveles de prioridad a cada uno de los hilos creados.

Pero cuéntame algo más, por favor...

Ahora deberíamos empezar a introducirnos dentro de una de las claves que nos pueden hacer comprender el motivo por el que el modelo de hilado múltiple, aun siendo mas consistente con la realidad misma, presenta problemas; y un tipo de problemas especiales por ser éstos sensibles al tiempo. Este tipo de problemas tienen un mayor nivel de dificultad para ser depurados por cuanto perfectamente pueden quedar ocultados en un entorno o circunstancias y aparecer repentinamente en otro. La depuración es posible y Delphi ayudará en dicha labor, pero la complejidad mayor de este proceso se hace evidente.

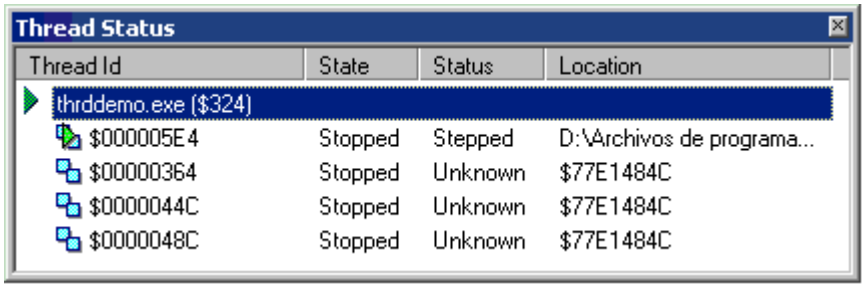

Figura 1 - Vista de la ventana Thread Status del depurador integrado de Delphi. Estado de los hilos de ejecución.

En la **figura 1** podéis ver una vista del depurador integrado, desde la ventana Thread Status, en la que se nos muestra información del estado de los distintos hilos creados. Otras vistas, como la Vista CPU nos entrega información del estado actual del procesador, como el valor de los registros en ese punto de parada, y en general, los aspectos más importantes de la CPU. Lo podéis ver en la **figura 2**.

Podemos empezar distinguiendo el concepto de proceso frente al de hilo. No, no es lo mismo. Cada uno de los procesos que son lanzados, disponen de un hilo de ejecución al que se suele llamar habitualmente hilo primario. Cada proceso pues, es propietario de al menos un hilo. En ese punto, el programador puede solicitar si lo cree necesario al Sistema Operativo la creación de nuevos hilos o hebras que permitan la ejecución de tareas que pueden ser simultáneas o no a este hilo primario. Pero vamos a intentar dejar sobre este figurado tapete alguna de las ideas que pueden ser claves para entender la complejidad que encierra la concurrencia tanto en procesos como en hilos de ejecución. Al ejecutarse un programa cualquiera de nuestro equipo, un nuevo proceso es cargado en memoria. Este proceso consta básicamente de varias zonas de memoria: por un lado podemos encontrar un ***área de código***, área de solo lectura con las instrucciones que ha de procesar secuencialmente nuestro procesador, un ***área de datos*** que representa al espacio reservado a las variables globales de la aplicación y lo que se suele denominar como “montón” (heap) y finalmente la ***pila*** del programa, donde son acumulados temporalmente los puntos de entrada y salida a las funciones y variables temporales. Además de estas zonas que intuitivamente reconocemos nuestro sistema operativo mantiene otras ligadas a la ejecución del mismo, como los valores de los registros de nuestra CPU, tiempo de proceso, recursos, etc... y que generalmente se denomina ***contexto.*** Y en definitiva, todo esto en conjunto, y dependiendo básicamente del sistema operativo al que hagamos referencia, viene a constituir lo que podemos entender generalmente como ***proceso***.

Cada hilo adicional que es creado, obtiene su propia pila distinta a la del proceso y una cola de mensajes también propia. Sin embargo, al ejecutarse dentro del contexto del proceso comparte zonas de

memoria y dichas zonas de memoria son accesibles a ellos. Está idea será repetida algo más adelante, en razonamientos posteriores.

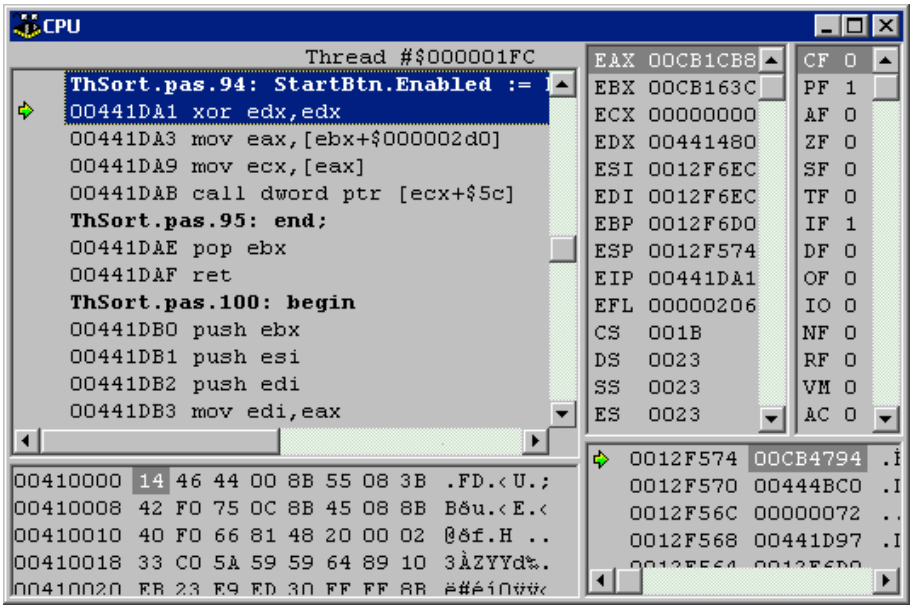

Figura 2 - Vista de la ventana CPU. Nos muestra información relevante del procesador en el punto de parada.

Aquí deberíamos hacer una pausa. Es un buen momento para leer el artículo de Jose Manuel Navarro de este mismo número y que trata precisamente sobre la "pila" en Windows y su funcionamiento, su implementación interna. El artículo os resultará de sumo interés y el autor, Jose Manuel, nos promete abordar otra de las estructuras de la memoria y que se corresponde a los montículos, ya nombrados en lineas anteriores.

Y llegados a este punto, introducimos una primera idea importante: cada vez que nuestro sistema operativo "detiene" la ejecución de un proceso para dar paso a otro, en ese ciclo de planificación del que hablábamos en lineas mas arriba , dispone de la información necesaria que le permita salvar el estado actual de ejecución del proceso saliente y reponer la del nuevo, reiniciando la ejecución en el justo momento en el que había sido detenido. Y lo más gracioso de todo es que ninguno de los dos tiene conciencia de que haya sido detenido ni de que haya sido relanzado. Ambos, tanto el proceso que sale como el que entra, piensan que están siendo ejecutados sin interrupción. El tema lógicamente es tan amplio como complejo y se nos escapa de las pretensiones del artículo. Nuestro enfoque es superficial. Podríamos hablar de como las áreas de cada proceso son protegidas por el sistema operativo. Cada proceso se ejecuta "solo", y tiene tan solo acceso a aquellas zonas que le pertenecen. Es lo que se denomina ejecución en modo usuario frente a la ejecución en modo supervisor que caracteriza al sistema operativo y que resulta francamente lógica, si tenemos en cuenta que el sistema operativo esta en un nivel superior y que, a toda costa se debe proteger de otra aplicación asuma su papel. Hablamos de los típicos mensajes, preludio en algunos casos de males venideros, donde se advierte que se ha producido una excepción... Todos nos hemos visto afectados por ellos alguna que otra vez.

Pero bueno... ¿entonces los hilos...? ¿los procesos...?. Podemos imaginar que el cambio de contexto necesario para iniciar y detener los procesos ha de consumir un tiempo x, indeterminado. Es el tiempo necesario para reponer el estado de un proceso que abandona la inactividad y que vuelve a ser repuesto en memoria para ser ejecutado... En ese momento es donde empiezan a cobrar sentido los hilos de ejecución. El mecanismo es llamado en ocasiones multiproceso ligero o hilos y básicamente consiste en dotar al proceso original de la capacidad de iniciar un nuevos flujos de ejecución que comparten zonas de memoria respecto al hilo primario de ejecución. De tal forma, la generación y ejecución de múltiples tareas se puede realizar sin llevar a cabo un cambio de contexto total, ya que parte de la información es compartida y por lo tanto, se

aligera una tarea que de ser mantenida mediante procesos independientes sería costosa. De esa forma, los procesos pueden seguir manteniendo su independencia, sin duda uno de los aspectos más característicos de esta arquitectura.

En cierta forma, nos damos cuenta de que mientras el cambio de procesos es transparente al desarrollador en tanto el sistema operativo tiene control absoluto sobre la interacción de los distintos procesos, la posibilidad de generación de nuevos hilos abre una puerta a la ejecución de múltiples tareas de acuerdo con una planificación y de una forma muy poco gravosa, dado que todos los hilos pueden interactuar libremente dentro del mismo espacio de direcciones, compartiendo zonas de memoria, variables y un hilo primario eje de la ejecución.

¿Problemas...?

Pero vamos a pensar en un momento en los recursos de que puede disponer una aplicación que se ejecute, en este mismo momento, en nuestro equipo: son limitados, discretos... y existe la necesidad de que sean compartidos. Un archivo, el CD ROM, etc... se nos pueden ocurrir muchos más. Y si hacemos el paralelismo a nivel de hilos de ejecución podríamos mencionar a las variables globales.

Imaginemos de forma gráfica una situación en el ámbito humano que sea metáfora de esto: nos hallamos tres compañeros uno frente a otro, con la sana intención de tomar café. Yo, educado, le ofrezco mi taza al primero de ellos para que me sirva de una hermosa cafetera de loza... En ese momento quedo sumido en un profundo y misterioso sueño. El primero, despierta para proceder al servicio del mismo, pero ay ... cuando iba a proceder a verter el café también queda dormido. El problema es que inesperadamente interviene el tercero de los invitados que, tras ver la taza de café (y no viéndome a mi) procede a hacerla suya para que el primer invitado le sirva.

El resultado lógico es que el primero de los participantes acabe por derramar todo el café en mis manos... quemándome si al despertar inicia el vertido del mismo...

Los hilos de ejecución no están exentos de todos estos problemas derivados de la concurrencia. Si seguimos la lectura de William Stallings en uno de sus muchos libros sobre sistemas operativos cuyo título es “Sistemas Operativos”, libro de cabecera en algunas universidades y en los primeros cursos de las Ingenierías en Informática, al hablar de los problemas creados por la multiprogramación apela al hecho de que la velocidad relativa de los procesos no pueden predecirse, tanto en sistemas monoprocesador como multiprocesador. Al referirse a las dificultades afirma literalmente:

- `La compartición de recursos globales está llena de riesgos. Por ejemplo, si dos procesos hacen uso al mismo tiempo de la misma variable global y ambos llevan a cabo tanto lecturas como escrituras sobre la variable, el orden en que se ejecuten las lecturas y escrituras es crítico.`

Y algo más adelante:

- `Resulta difícil localizar un error de programación porque los resultados no son normalmente reproducibles.`

Hilos y procesos, como podemos apreciar comparten parecidos problemas y cuando iniciemos un desarrollo en que haga uso de hilos de ejecución habremos de extremar el cuidado. Respecto a los procesos, las mismas soluciones aportadas para mitigar la competencia son generadoras de nuevos males que pueden aparecer de forma inesperada. Estamos hablando concretamente de la introducción de lo que se pueden llamar zonas de exclusión mutua (secciones críticas). Nuevos remedios, nuevos males: Interbloqueos y lo que se viene a denominar Muerte por Inanición.

En el plano de los hilos, y ya centrándonos un poco en el tema que nos ocupa, algunos problemas reales, como lo que se viene a llamar Deadlock o “abrazo mortal”, donde un hilo queda suspendido de forma indefinida o dicho de otra manera, dos hilos se esperan mutuamente. O problemas propios de la sincronización de varios hilos en el desarrollo de una tarea en la que se interrelacionan. Si el lanzamiento de un nuevo hilo se llevara a cabo como un proceso totalmente independiente del resto de tareas ejecutadas... no, no suele ser el caso habitual. En buena parte de los casos una tarea depende del resultado de otra y obliga a dicha sincronización.

En todos estos contextos, podemos entender que los Sistemas Operativos incorporen una serie de objetos propios, directamente relacionados con el concepto de sincronización: Secciones críticas, Semáforos, objetos de exclusión mutua o Mútex o Eventos...

Los objetos relacionados con la sincronización.

Podemos empezar este apartado hablando de las ***Secciones críticas*** y comentando brevemente en que consisten. Una sección crítica es un fragmento de código que tiene carácter de unidad atómica, y por atómica entendemos que es comprendido todo el fragmento como una unidad. Dicho fragmento, perteneciente a uno de los hilos en ejecución se mantiene así, activo, en tanto no haya abandonado la sección y el resto de los hilos quedan a la espera de su salida de dicha sección crítica para activarse de nuevo. El inicio y el final de la sección crítica viene marcado por las llamadas a las funciones *EnterCriticalSection* y *LeaveCriticalSection*. El código comprendido entre la llamada inicial y la llamada final es el fragmento de código “crítico”. Este objeto nos ayudará por ejemplo a proteger determinados recursos para que no puedan ser accedidos por varios hilos de ejecución simultáneamente.

El API de Windows nos entrega varias funciones:

- InitializeCriticalSection( )
- EnterCriticalSection( )
- LeaveCriticalSection( )

Tal y como comentábamos, entraremos en detalles en capítulos posteriores. Nos interesa en este momento tener una idea global al menos. Pero sigamos avanzando.

Cuando hablamos del concepto de ***Mutex*** intuitivamente lo denominamos como semáforo binario. Su nombre, ¡ya, ya, algo raro! proviene de **MUT**ual **EX**clusion y su concepto es muy próximo al de Sección crítica, y nos ayudará a sincronizar el acceso a los recursos por un solo hilo. Steve Texeira y Xavier Pacheco en un maravilloso libro sobre Delphi “Guía de Desarrollo Delphi 5” y que me ha servido de gran ayuda para documentarme sobre el tema junto con otra documentación, comentan la utilidad de éstas para “sincronizar los threads entre fronteras de procesos”.

Comentan además, la diferencia del objeto Mútex frente a la Sección Critica en términos de rendimiento, teniendo esta última, mejor rendimiento.

Se iniciará el fragmento de código crítico con la llamada a *CreateMutex*, función que nos entregara un handle al objeto creado. Este handle nos permitirá con la llamada a la función *CloseHandle* darlo por finalizado.

Algunas de las funciones relacionadas con el uso de Mutex son:

- CreateMutex( )
- ReleaseMutex( )
- CloseHandle( )

- OpenMutex( )
- WaitForSingleObject( )

Otro paso adelante... los ***semáforos***. Intuitivamente ya nos imaginamos en que consisten. También van a poder actuar dentro de las fronteras de los procesos. En el momento de proceder a la creación de este objeto, determinamos el número de hilos que podrán acceder a una sección que entendemos como crítica, con el resultado del bloqueo del mismo y el conteo de cada uno de los accesos. Nos basta de momento con saber poco más.

Veamos algunas de las funciones del API relacionadas con los Semáforos:

- CreateSemaphore( )
- ReleaseSemaphore( )
- OpenSemaphore( )
- WaitForSingleObject( )

Y por último los ***Eventos***. Respecto a este tema ando más escaso de documentación y me remito a lo que me ofrece el *Nucleo del Api Win32* en la colección de los Tomos de Delphi y que edita en español la editorial DanySoft. Muchas veces he comentado que me parece un magnífico libro para los programadores que nos hemos acercado al API con conocimientos demasiado básicos para enfrentarnos a una documentación en Ingles, dura y árida. Y al hilo de todo esto me pregunto: ¿cuando tendremos un Delphi en castellano...? ¡Una documentación a la altura de las circunstancias...!

Vale, vale... ya me conocéis que me pierdo con las ramas. Volvamos a lo nuestro.

Los eventos son objetos que permitirán al programador dirigir el control de la etapa de sincronización ya que el marcado como señalizado o no señalizado del objeto que sincroniza le efectuara en forma de llamadas a las funciones *SetEvent*, *ResetEvent* o *PulseEvent*. En este punto puede aparecer uno de los problemas mencionados anteriormente y que hacían referencia a los DeadLock o “abrazo mortal” entre hilos de ejecución. La intervención del programador sin duda posibilita esto.

Algunas de las funciones que abordaremos:

- CreateEvent( )
- SetEvent( )
- ResetEvent( )
- PulseEvent( )
- OpenEvent( )

Pienso que podemos cerrar este apartado aun cuando han quedado en el tintero algunos comentarios que dejaremos para un momento posterior. Supongo que tendremos tiempo para ir abordando todo con bastante detenimiento.

Bla, bla, bla... y no comentas nada de los Threads...

¡Venga!, ¡No seáis impacientes! Estamos introduciéndonos en el tema y sentando las bases para abordarlo con cierto detalle. La idea que hasta ahora tenemos, y que nos aportan los apartados anteriores es que nos encontramos ante una especie de monstruo de dificultad extrema y nada más lejos de la realidad...

Delphi encapsula en la clase TThread la funcionalidad de los threads del API de Windows y lo hace de forma sencilla, increíblemente sencilla. Crear un nuevo hilo será tan fácil como sobreescribir en un descendiente de la clase TThread el método *Execute*, definido como abstracto en la interfaz de la clase, y comprender la mecánica y ciertos detalles. En el **listado 1**, al final de este artículo, podéis visualizar la declaración de métodos que hace públicos la clase TThread en su interfaz.

¿No os lo creéis, eh?. Si pensáis que resulta difícil haced lo siguiente: abrid el entorno de Delphi 5 o Delphi 6. En el menú superior en Delphi 6 pulsamos <File | New | Other>, en Delphi 5 tan solo <File |New> y tras emerger una ventana que pone a nuestra disposición la diferentes opciones de objetos a crear, en la pestaña New deberemos encontrar el objeto “Thread Object”. Haced doble click sobre él y ya está. Tan solo tenéis que entregar en la ventana que emerge el nombre de la nueva clase descendiente de TThread y se generará de forma automática un código similar a:

```
unit Unit1;
interface
uses
  Classes;
type
  Mi_Thread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
procedure Mi_Thread.Execute;
begin
  { Place thread code here | Código del hilo}
end;
end.
```

Cuando sea creada una nueva instancia de la clase Mi_Thread, será el método *Execute* el que contenga el código que ha de ejecutar el nuevo hilo creado. Entraremos en detalles en los siguientes artículos de esta serie. No os preocupéis si ahora no queda del todo claro porque lo veremos todo con detalle.

En realidad hemos estado hablando de la multitarea pero realmente no hemos puesto ningún ejemplo, y quizás este puede ser un buen momento para hacerlo. Cualquiera nos puede valer: una búsqueda secuencial en un texto, un cálculo de saldos para un balance en una aplicación de gestión, determinados procesos en una aplicación gráfica, todos ellos susceptibles de poder quedar en un segundo plano y liberar al hilo primario de la carga. Delphi nos trae un ejemplo en la carpeta “..\Demos\Threads” del directorio en que instalamos el compilador. Es un ejemplo sencillo en el que vamos a ejecutar tres algoritmos de ordenación de un vector de enteros simultáneamente, cada uno de ellos sobre un vector que contiene los mismos elementos. Son tres algoritmos de sobra conocidos: Bubble (Burbuja), Selección y QuickSort. Sobre este último, comentar que ya fue objeto de nuestro análisis en la serie sobre Objetos Auxiliares.

Como comentario a dicho ejemplo, y adelantándonos al contenido de un posterior artículo, resaltar como se hace necesario acceder a la interfaz del usuario tan solo desde el contexto del hilo primario. El motivo es porque la VCL no soporta el acceso concurrente de múltiples hilos. El remedio viene de la mano del método Synchronize( ), que recibe como parámetro un puntero a método. Vamos a verlo:

El método Execute invocará a *Sort( )*. Cada uno de los tres descendientes ha redefinido dicho método que se declara como abstracto en el ascendiente, efectuando las operaciones de manipulación del vector que le sean oportunas. El uso de *Slice( )*, si os presenta duda, tendrá como fin obtener un subvector que espera como parámetro Sort( ).

```
procedure TSortThread.Execute;
begin
  Sort(Slice(FSortArray^, FSize));
end;
```

Si visualizáis cualquiera de los tres métodos *Sort( )* de la unidad *SortThds.pas* podremos observar como se produce la invocación del método *VisualSwap( )* y que este, a su vez, a través de *Synchronize( )* y un puntero al método *DoVisualSwap*, accedería al elementos del interfaz dentro del contexto del hilo primario.

```
procedure TSortThread.VisualSwap(A, B, I, J: Integer);
begin
  FA := A;
  FB := B;
  FI := I;
  FJ := J;
  Synchronize(DoVisualSwap);
end;

procedure TSortThread.DoVisualSwap;
begin
  with FBox do
  begin
    Canvas.Pen.Color := clBtnFace;
    PaintLine(Canvas, FI, FA);
    PaintLine(Canvas, FJ, FB);
    Canvas.Pen.Color := clRed;
    PaintLine(Canvas, FI, FB);
    PaintLine(Canvas, FJ, FA);
  end;
end;
```

El resultado final de la ejecución de la demo que nos proporciona Delphi, es la ordenación de los tres vectores de enteros mediante tres algoritmos distintos, simultáneamente; algo imposible de satisfacer si consideramos tan solo el uso del hilo primario, dado el carácter secuencial de la ejecución del código.

Buscando el final...

Hemos llegado al final y de alguna forma se espera una conclusión que resuma lo dicho. Es pronto para hacerla y la dejamos para un poco más adelante. Intentaremos entonces tener criterio suficiente para valorar si puede ser positivo incorporar a un desarrollo concreto nuevos hilos de ejecución. Veremos que resulta sencillo pero que, como todo, tienes sus pros y sus contras, sus ventajas y sus inconvenientes. Nos queda mucho camino pero también muchas ganas de recorrerlo.

Recibid un saludo y hasta el próximo número.

```
  TThread = class
  private
    ...
    ...
    ...
    procedure CheckThreadError(ErrCode: Integer); overload;
    procedure CheckThreadError(Success: Boolean); overload;
    procedure CallOnTerminate;
{$IFDEF MSWINDOWS}
    function GetPriority: TThreadPriority;
    procedure SetPriority(Value: TThreadPriority);
    procedure SetSuspended(Value: Boolean);
{$ENDIF}
{$IFDEF LINUX}
    function GetPriority: Integer;
    procedure SetPriority(Value: Integer);
    function GetPolicy: Integer;
    procedure SetPolicy(Value: Integer);
    procedure SetSuspended(Value: Boolean);
{$ENDIF}
  protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue write FReturnValue;
    property Terminated: Boolean read FTerminated;
  public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure AfterConstruction; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    property FatalException: TObject read FFatalException;
    property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
    property Handle: THandle read FHandle;
{$IFDEF MSWINDOWS}
    property Priority: TThreadPriority read GetPriority write SetPriority;
{$ENDIF}
{$IFDEF LINUX}
    property Priority: Integer read GetPriority write SetPriority;
    property Policy: Integer read GetPolicy write SetPolicy;
{$ENDIF}
    property Suspended: Boolean read FSuspended write SetSuspended;
{$IFDEF MSWINDOWS}
    property ThreadID: THandle read FThreadID;
{$ENDIF}
{$IFDEF LINUX}
    property ThreadID: Cardinal read FThreadID;
{$ENDIF}
    property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
  end;
```

Listado 1 – Definición de la Clase TThread. (Nota: En el listado no aparecen los campos privados).