TThread III

A lo largo de esta tercera entrega en la que se estudia la clase TThread en la VCL, abordaremos el resto de su implementación en Windows, retomando el punto en que nos quedamos en el artículo anterior.

En nuestro artículo anterior, el segundo de la serie, hacíamos un punto de parada en el momento de la creación del nuevo hilo, entendiendo que la materia abordada hasta ese momento, y la densidad de algunos conceptos, nos hacía merecedores de ese pequeño descanso. Habíamos seguido, paso a paso, todas y cada una de las rutinas desde la creación del objeto descendiente de la clase TThread, que en nuestro ejemplo era la instancia de la clase TDibujante, hasta su lanzamiento mediante la invocación del método Resume, siendo nuestro objetivo principal observar que pasaba entre bastidores, internamente, analizando como eran invocadas aquellas funciones que proporcionaba el S.O. de Windows en la creación de nuevos hilos de ejecución, y de que forma se protegía el acceso a las variables globales que permitían la sincronización.

Los conceptos que involucraban conocimientos adicionales sobre la API de Windows eran abordados a medida que se hacían necesarios. De hecho, nuestra idea no era en si, abordar todas y cada una de las funciones que proporciona el API, ni explicar de forma detallada o exhaustiva cada una de ellas, sino aportar una serie de conocimientos básicos que nos dieran una buena idea de como eran encapsuladas, en la clase TThread, estas funciones y como, ciertamente, había sido hecho esto de forma sumamente eficiente, facilitándonos de forma sencilla la incorporación de técnicas de multihilado en nuestras aplicaciones.

Por eso, creo que sería conveniente antes de iniciar la lectura de este artículo, tener leído el anterior, pues éste da continuación a aquél, desde el mismo momento en que ha sido creado el hilo y se ha lanzado su ejecución. Y no dudo tampoco en recomendar que se compile y ejecute aquel sencillo ejemplo que se entregó junto a dicho artículo. Precisamente el hecho de su sencillez nos pueda hacer entender con mayor facilidad algunas de las ideas expuestas. En ocasiones, la complejidad de muchos desarrollos que se acompañan como ejemplos, ocultan ideas que resultan tan sencillas que uno se pierde entre detalles, que en la mayoría de los casos llegan a ser superfluos e innecesarios.

Vamos allá sin más dilación.

El nuevo hilo se crea... ¿y ahora qué?

Tras la creación del nuevo hilo, mediante la llamada que internamente se hacía a la función del API *CreateThread( )*, y que estuvo ampliamente explicada en la página 44 del número anterior, la clase TThread proporciona un método público para iniciar la ejecución del mismo: es el procedimiento *Resume*, que a fin de cuentas encapsula una llamada a la función *ResumeThread( )* también del API.

Veamos que hace realmente el procedimiento *Resume*.

```
{$IFDEF MSWINDOWS}
...
procedure TThread.Resume;
var
  SuspendCount: Integer;
begin
  SuspendCount := ResumeThread(FHandle);
```

```
  CheckThreadError(SuspendCount >= 0);
  if SuspendCount = 1 then
    FSuspended := False;
end;
{$ENDIF}
```

Como veis, declara localmente una variable de tipo entero cuya misión es obtener el valor del contador de suspensiones del hilo, **en el momento previo a la llamada a la función**, y que será el valor que nos retorne la llamada a la función *ResumeThread( )*. Esta función es incorporada por la librería KERNEL32.DLL, que básicamente recoge todas aquéllas necesarias a bajo nivel, y que se ocupan de temas tales como la administración de la memoria o la gestión de los recursos.

La sintaxis de esta función es la siguiente:

```
DWORD ResumeThread(

    HANDLE hThread
   );
```

Recibirá como parámetro el Handle o Manejador del hilo que se desea activar, y que almacenamos inicialmente en el campo FHandle. Como valor de retorno obtenemos el valor del contador de suspensiones de dicho hilo, en el momento actual y antes de ser decrementado en una unidad. Cuando el contador vale 0, después de hacer el decremento de dicha unidad, el hilo es ejecutado. En caso contrario sigue suspendido.

Así pues, se entiende que una vez hecha la llamada a la función y recuperado el valor en la variable local *SuspendCount*, se evalúe si ha existido algún error, que es básicamente lo que se pretende al invocar la rutina *CheckThreadError(SuspendCount >= 0)*, puesto que ésto, se traducirá finalmente en una llamada a *GetLastError* **solo** en el caso de que *SuspendCound* haya devuelto un valor menor que cero, concretamente **$FFFFFFFF**, valor de retorno de *ResumeThread* en el caso de error. ¿Queréis verlo con mayor detalle?

Nosotros entregaremos como parámetro de la función, la evaluación booleana de la condición,

```
SuspendCount >= 0
```

y en el caso de que se evalúe como falso, es decir que sea menor que cero, y es así como se interpretaría el valor hexadecimal $FFFFFFFF como Integer, (teniendo en cuenta el Complemento a Dos), procederemos a una nueva llamada al procedimiento, pero esta vez, limpiando el valor de error depositado por la función *ResumeThread( )*, que será el resultado de *GetLastError*

```
procedure TThread.CheckThreadError(Success: Boolean);
begin
  if not Success then
    CheckThreadError(GetLastError);
end;
```

Gracias a la sobrecarga de funciones, es elegida ante el nuevo parámetro, la versión del procedimiento *CheckThreadError( )* que recibe como parámetro un *Integer*. Y tan solo tenemos ya que formatear dicho error y generar la excepción.

```
procedure TThread.CheckThreadError(ErrCode: Integer);
begin
  if ErrCode <> 0 then
    raise EThread.CreateResFmt(@SThreadError, [SysErrorMessage(ErrCode), ErrCode]);
end;
```

Solo en el caso de haber obtenido éxito y de no haberse producido el error y la excepción correspondiente, procedemos a modificar el valor de la variable *FSuspensed*,

```
if SuspendCount = 1 then  FSuspended := False;
```

Si el valor del contador de suspensiones del hilo, es igual a 1, *FSuspended* tomará como valor False, puesto que en ese caso el hilo es lanzado, como explicamos una lineas más arriba.

Ésto, nos permite meternos de lleno en varios aspectos nuevos que acarrea esta situación, y que tienen que ver directamente con la sincronización de los hilos, y concretamente con el estudio del método *Synchronize( )* por un lado, y por otro el seguimiento de la función *ThreadProc( )*, que será realmente la que inicia la ejecución del hilo lanzado.

Empecemos por esta última.

La función ThreadProc: Se inicia la ejecución.

Hacemos un poco de memoria, rápidamente. Vamos a buscar la página 39 de nuestro anterior artículo. Recordad que en la creación del hilo intervenía la función *BeginThread( ),* que recibía en uno de sus parámetros ***@ThreadProc***, que no va a ser, ni mas ni menos, la dirección de esta función. El tercer parámetro, ***Pointer(Self),*** que representa al hilo actual que hace la llamada, será el parámetro que finalmente recibirá esta función para identificar a dicho objeto.

Veamos como está implementada la función. Tened en cuenta que omitimos aquellos fragmentos que afectan al S.O. de Linux para no liarnos innecesariamente en nuestros razonamientos.

```
function ThreadProc(Thread: TThread): Integer;
var
  FreeThread: Boolean;
begin
  try
    if not Thread.Terminated then
    try
      Thread.Execute;
    except
      Thread.FFatalException := AcquireExceptionObject;
    end;
  finally
    FreeThread := Thread.FFreeOnTerminate;
    Result := Thread.FReturnValue;
    Thread.FFinished := True;
    Thread.DoTerminate;
    if FreeThread then Thread.Free;
    EndThread(Result);
  end;
end;
```

Este será **TODO** el código que va a ejecutar nuestro nuevo hilo desde su nacimiento. De forma particular, nuestro escenario es el método ***Execute.*** Ya se puede entender también porque se hace necesario anular el método Execute en los descendientes de la clase TThread. Este, declarado como abstracto y virtual en esta clase, nos permite lanzar la ejecución de nuestro código. En nuestro ejemplo anterior simplemente dibujábamos una figura geométrica que giraba.

Veámoslo con detalle, línea a línea:

try

```
if not Thread.Terminated then
try
  Thread.Execute;
except
  Thread.FFatalException := AcquireExceptionObject;
end;
```

La propiedad *Terminated* tan solo lee el valor del campo *FTerminated*. Es una propiedad de solo lectura y lo hace directamente a través de dicho campo. Es decir, que se va a evaluar previamente si el hilo ha finalizado prematuramente antes de su ejecución, en cuyo caso se pasaría a ejecutar las rutinas que preceden a ***finally***, para garantizar la integridad en el estado final del hilo.

No es el caso habitual. Lo normal es que sea llamado el método *Execute* sobrescrito por el descendiente. En cualquier caso, ***finally*** nos garantiza que tras la ejecución de nuestro hilo se lleven a cabo una serie de acciones necesarias, tales como liberarlo, por poner el ejemplo mas claro, en el caso de que lo hallamos estipulado así.

Pero podemos resaltar algunos detalles que me parecen también interesantes y que se pueden comentar. Leemos que el campo de tipo TObject, *FFatalException*, va a apuntar (va a ser asignado) a algo “raro” que no nos recuerda para nada un tratamiento normal de las excepciones. Estamos habituados a que, tras la palabra reservada **except** se presenten todos aquellos manejadores del objeto excepción en una estructura:

on [Exception] **do** [tratamiento].

Sin embargo, la asignación se hace al valor de retorno de la función *AcquireExceptionObject,* declarada en el módulo System.pas. ¿Que nos devuelve esta función...?

Leamos que nos dice la ayuda en linea:

Use AcquireExceptionObject to keep the exception object after the except clause exits. AcquireExceptionObject increases the reference count on the exception object so that it is not automatically freed.

Es decir. Nos va a devolver el objeto Excepcion, la excepción misma. Y nos dice que se incrementa el contador de referencias para que no sea liberada automáticamente.

¿Lo veis un poco mas claro? En condiciones normales, la excepción sería destruida de forma automática toda vez que ha sido capturada por alguno de los manejadores pero en este caso no podemos saber que tratamiento se le puede dar, ni que clase de excepción va a ser generada, por lo que la estrategia que se sigue es impedir que sea liberado dicho objeto incrementado su contador de referencias y facilitando que posteriormente pueda ser tratada posteriormente por el “usuario” (nosotros). La propiedad pública:

property FatalException: TObject **read** FFatalException;

nos permitirá acceder al objeto excepción y chequear si ha existido o no, si ha finalizado con éxito la ejecución del hilo, o por el contrario se ha producido algún error de ejecución. Y esta comprobación se hace dentro del manejador del evento *OnTerminate*, invocado indirectamente a través de la llamada al método *DoTerminate* posterior.

```
finally
    FreeThread := Thread.FFreeOnTerminate;
```

Nos valíamos de esta propiedad booleana para saber si el hilo debía ser destruido tras finalizar su ejecución. Ese es el momento en que hacemos uso de nuestra anterior asignación para posteriormente, lineas más abajo y según el estado de la variable *FreeThread,* proceder a la liberación del objeto.

```
    Result := Thread.FReturnValue;
```

Entregamos como retorno de la función el valor de la propiedad *FreturnValue,* que puede ser modificado por nosotros en el descendiente, ya que se trata de una propiedad declarada en la zona ***protected*** (protegida) de nuestro objeto. Nos puede servir como valor esperado por otros hilos para la toma de decisiones.

```
    Thread.FFinished := True;
```

Llegado a este punto, fijamos el estado de finalización del hilo en el campo FFinished. Hay que tener en cuenta que esta variable es necesaria para saber si en el momento de la destrucción del objeto el hilo creado esta ejecutándose, o por el contrario ya ha finalizado. Lo veremos al analizar el método *Destroy*

```
    Thread.DoTerminate;
```

Damos una oportunidad para que nuestro usuario implemente el código necesario antes de proceder a la destrucción del objeto. Hacíamos referencia al evento OnTerminate como el lugar en donde podía ser chequeada la finalización con éxito de nuestro hilo y cualquier otro tipo de acción que nos se haga necesaria. Algunos autores

nos explican, que dicho evento se ejecuta en el contexto de nuestro hilo primario y dicho así, parece que hable de alguna otra cosa que nos sea desconocida. Esta es la implementación que hace este método:

```
procedure TThread.DoTerminate;
begin
  if Assigned(FOnTerminate) then Synchronize(CallOnTerminate);
end;

procedure TThread.CallOnTerminate;
begin
  if Assigned(FOnTerminate) then FOnTerminate(Self);
end;
```

Así se ve mucho más claro. Seguimos haciendo uso del método *Synchronize( )* y de la zonas críticas explicadas en el artículo anterior para acceder al contexto del hilo primario.

```
    if FreeThread then Thread.Free;
    EndThread(Result);
  end;
end
```

Y ya para finalizar, si se ha establecido así, procederíamos a destruir el hilo de ejecución mediante una llamada al método *Free*, que como es sabido, comprueba primero que el objeto existe (no apunta a **Nil**) y en ese caso es llamado el procedimiento **Destroy**.

Nos queda tan solo una pregunta que hacernos y es ¿qué hace la rutina *EndThread( )* toda vez que se ha destruido nuestro objeto hilo? La respuesta es simplemente detener la ejecución del mismo. Recordemos que el objeto tan solo encapsula el uso de las funciones que hacen uso de los hilos en el S.O. de Windows. Es decir, una cosa es el objeto y otra el hilo de ejecución. La función *EndThread( )* llamará en su implementación a la función del S.O. *ExitThread ( ).* Y esta función presenta la siguiente sintaxis:

```
VOID ExitThread(

    DWORD dwExitCode    // código de salida para el hilo
   );
```

Como veis no retorna valor alguno y simplemente detendrá la ejecución del hilo, desalojando la pila creada para el mismo y dando por finalizada la vida del objeto Thread del S.O. de Windows. Una pena...

El método Destroy para finalizar...

Nos queda el análisis de qué es lo que sucede en el momento de la destrucción del objeto (recordad la llamada al método *Free*). Veamos paso por paso como es implementado *Destroy*, que os recuerdo es declarado como **override,** anulando así el método en el ascendiente.

Empezamos...

```
destructor TThread.Destroy;
begin
```

Lo primero es comprobar que la ejecución del hilo ha finalizado y nos valemos del valor booleano del campo *FFinished*, que era modificado a True si el método Execute había finalizado, con éxito o sin él. Luego lo primero que hay que hacer es, si no ha finalizado, finalizarlo. Es decir, se produce una llamada al método *Terminate* que simplemente modifica *FFinished* para que tome valor verdadero y marque el verdadero estado de nuestro objeto.

```
  if (FThreadID <> 0) and not FFinished then
  begin
    Terminate;
```

Lo dicho... *FFinished* nos dirá que el hilo ha finalizado.

```
    if FCreateSuspended then Resume;
```

```
    WaitFor;
  end;
```

En el caso de que el hilo esté en estado “Suspendido” habremos de iniciar su ejecución para poder identificarlo y esperar a que finalice. Esta es la misión del procedimiento *WaitFor,* el cual, no retornara en tanto no finalice el hilo.

```
  if FHandle <> 0 then CloseHandle(FHandle);
```

Liberamos el Handle de nuestro Hilo...

```
  inherited Destroy;
```

Se produce la llamada al destructor en la clase ascendiente...

```
  FfatalException.Free;
```

Destruimos ya el objeto Excepcion, capturado (si se ha producido algún error durante la ejecución del método Execute) en el tratamiento de las excepciones del método *ThreadProc( ),* como explicamos lineas más arriba.

```
  RemoveThread;
end;
```

Y por fin y para acabar ya, una última llamada para eliminar de nuestro contador de hilos creados al infeliz que hemos liquidado...

¿Que hace RemoveThread?

```
procedure RemoveThread;
begin
  EnterCriticalSection(ThreadLock);
  try
    Dec(ThreadCount);
  finally
    LeaveCriticalSection(ThreadLock);
  end;
end;
```

Es un buen momento para repasar el articulo anterior. Un momento... os facilito la página. Es la página 37 del número 12 de Síntesis, cuando hablamos del procedimiento *AddThread* y comentábamos como era creada una lista global y se protegía el acceso a la variable *ThreadCount*, (que representaba el número de hilos creados) mediante el uso de secciones críticas. Cada vez que era creado un nuevo hilo se procedía a incrementar el contador *ThreadCount* y en este caso, destruido el objeto, decrementamos en una unidad dicho contador.

Así pues, visto esto, y finalizada la ejecución y destruido el objeto, acabaría el estudio del ciclo vital del hilo creado, que era lo que pretendíamos al iniciar nuestro artículo anterior. De hecho, si no hiciéramos acceso al hilo primario en la ejecución de nuestro hilo se podría prescindir, por nuestra parte, del uso de Synchronize. Está claro, tal y como hemos comentado que la clase TThread sí que hace uso internamente de este método, pero centrándonos en nosotros, desde la redefinición del método *Execute* no es algo obligatorio. Como veremos más adelante, Windows nos provee también de otros objetos que nos facilitan la sincronización de hilos y pueden ser usados alternativamente a este.

¿Queréis que los veamos?. Vosotros mandáis...

Algunas cuestiones previas.

En el **listado 1**, tenéis la implementación que hace la VCL del método *Synchronize*, usado en la sincronización de los hilos que concurren. Antes de introducirnos en su análisis, se nos hace necesario ver un poco mas de cerca algunos conceptos que ya han salido, y que incluso fueron nombrados y usados en los métodos

descritos por nosotros. Era el caso de los objetos Eventos, que ya los anticipábamos en la página 36 de nuestro anterior artículo y segundo de la serie.

Estos objetos Eventos se nos presentan como algo misterioso y con cierto halo de sobrenatural y en cierta forma, a partir de este momento, y casi como anfitrión, asumo el papel de la presentación: Aquí el objeto Evento... Aquí unos amigos...

Bueno. Ya estáis presentados. Vamos a romper el hielo y empezaremos precisamente con este objeto, dejando para un segundo momento el semáforo binario o Mutex.

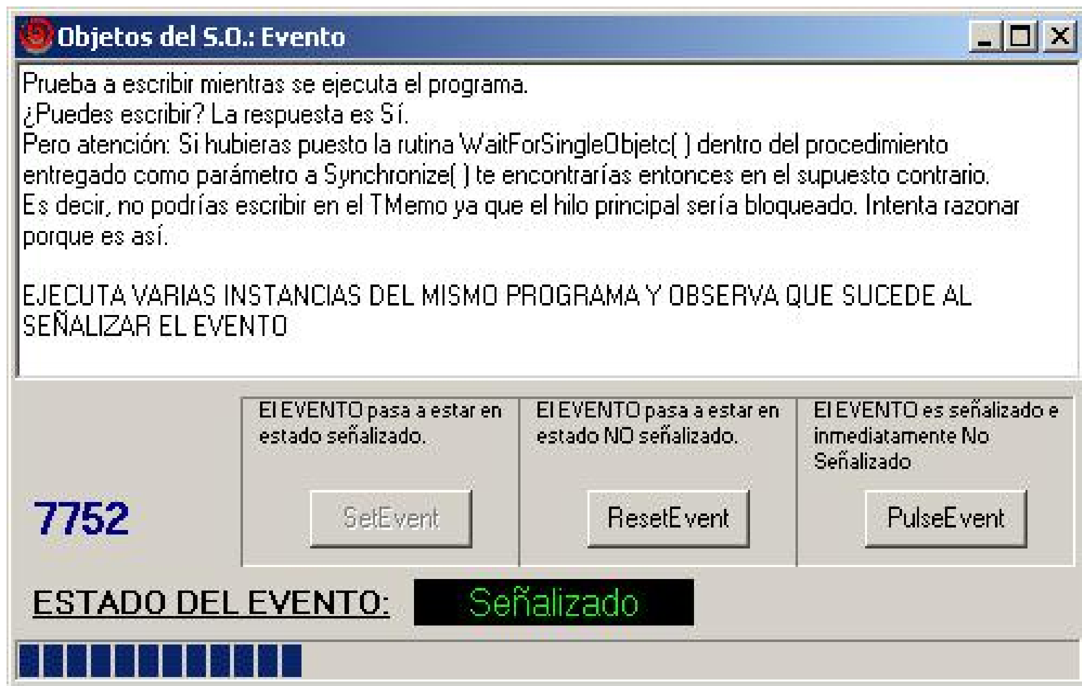

Figura 1.- Ejemplo muy básico sobre el uso de eventos que acompaña al artículo

Objetos Evento.

Decíamos en el primer artículo de la serie, que un Evento era un objeto del S.O. y que tenía la peculiaridad de que su estado era manipulable por el programador. Estamos haciendo referencia a la página 72 del Número 11 de Síntesis. Dicho objeto, global a todos los procesos e hilos en ejecución, mantenía dos estados posibles: o bien "señalizado" o bien "no señalizado" y el programador se servirá de funciones como *SetEvent*, *PulseEvent* o *ResetEvent* para modificar el estado del objeto.

¿Qué significa esto realmente?

Realmente esto no significa nada hasta que no aparece el tercero en discordia: una función como *WaitForSingleObject( ),* cuya única misión es detener la ejecución del hilo en tanto no se den dos condiciones:

1ª Que el objeto sea señalizado.

2ª Si no se establece el parámetro INFINITE y se limita el tiempo de parada del hilo a X Milisegs., y se vence dicho tiempo.

Así pues, pon esta función delante de cada uno de los hilos y tras crear el objeto evento tendrás una oportunidad de detenerlos.

```
procedure TThread.Synchronize(Method: TThreadMethod);
var
  SyncProc: TSyncProc;
begin
{$IFDEF MSWINDOWS}
  SyncProc.Signal := CreateEvent(nil, True, False, nil);
  try
{$ENDIF}
{$IFDEF LINUX}
    FillChar(SyncProc, SizeOf(SyncProc), 0);
{$ENDIF}
    EnterCriticalSection(ThreadLock);
    try
      FSynchronizeException := nil;
      FMethod := Method;
      SyncProc.Thread := Self;
      SyncList.Add(@SyncProc);
      ProcPosted := True;
      if Assigned(WakeMainThread) then
        WakeMainThread(Self);
{$IFDEF MSWINDOWS}
      LeaveCriticalSection(ThreadLock);
      try
        WaitForSingleObject(SyncProc.Signal, INFINITE);
      finally
        EnterCriticalSection(ThreadLock);
      end;
{$ENDIF}
{$IFDEF LINUX}
      pthread_cond_wait(SyncProc.Signal, ThreadLock);
{$ENDIF}
    finally
      LeaveCriticalSection(ThreadLock);
    end;
{$IFDEF MSWINDOWS}
  finally
    CloseHandle(SyncProc.Signal);
  end;
{$ENDIF}
  if Assigned(FSynchronizeException) then raise
FSynchronizeException;
end;
```

Listado 1: Implementación del método Synchronize( )

Dicho esto de este modo, se nos queda algo abstracto y realmente uno se queda como esperando algo más, sobretodo si se enfrenta por primera vez a esta serie de conceptos. Lo mejor es que veamos un ejemplo y para ello os pido que busquéis entre las fuentes que acompañan a la revista y dentro de las carpetas *d6_obj_evento* o *d5_obj_evento.* Si usáis Delphi 6 abrid la primera de ellas y en caso de Delphi 5 o inferior la segunda. Dentro de ellas hay que seleccionar el archivo de proyecto *pro_event.dpr.* Es un ejemplo muy sencillo y en el que tan solo involucramos un hilo. Así pues, si lo tenéis a la vista, yo tan solo comentaré aquellos trozos sobre los que podamos reflexionar y ampliar conceptos. En la **Figura 1** podéis ver el ejemplo en ejecución.

La idea que se persigue es crear un hilo que haga algo muy sencillo, que en este caso es incrementar la posición de la barra de progreso o TProgressBar, desde el valor cero (0) hasta alcanzar el máximo fijado en la propiedad *Max* de dicha componente. Durante esa ejecución el hilo podrá ser detenido gracias a que existe una llamada a la función *WaitForSingleObject( )* dentro del bucle **while** ... **do.**

Veamos primero como es creado el Evento. En este caso nos valemos del evento de creación del formulario, pero podríamos haber elegido cualquier otro lugar con la condición deseable y lógica de que tan solo se ejecute una sola vez.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //Creamos el evento
  Marcador:= CreateEvent(nil, True, False, 'MiSeñal');

  MiBarra:= TMiBarra.Create(Barra.Max); //creación del hilo
  MiBarra.Resume;   //lanzamiento del mismo
end;
```

Esta es la sintaxis de la función *CreateEvent( )*:

```
HANDLE CreateEvent(

    LPSECURITY_ATTRIBUTES lpEventAttributes,
    BOOL bManualReset,
    BOOL bInitialState,
    LPCTSTR lpName
   );
```

El primero de los parámetros, *lpEventAttributes,* es un puntero al registro TSegurityAttributes, que contiene la información sobre los atributos de seguridad y que como veis, en la creación del evento se ha entregado el valor **nil,** indicándole que sea considerados los atributos de seguridad por defecto.

El segundo de los parámetros, *bManualReset,* nos ayudará a que el estado no señalizado sea repuesto de forma automática por Windows o bien por nosotros. En el caso actual, le damos valor *True.* Le estamos diciendo al sistema operativo que el "reset" se hará de forma manual, lo haremos nosotros.

El tercer parámetro hace referencia al estado inicial del objeto Evento. ¿En que estado se inicia? ¿Señalizado? ¿No señalizado?. Al entregar el valor de False le estamos diciendo que no se señalice el objeto y por lo tanto, el hilo quedará a la espera, congelado al llegar a la función WaitForSingleObjetc en espera de que dicho objeto cambie a estado señalizado (o vía libre).

Y ya el cuarto de los parámetros, *lpName,* un PChar que contiene el nombre del objeto Evento, limitado en longitud a no mas de MAX_PATH caracteres.

El valor de retorno de esta función será el Handle o Manejador al objeto Evento creado, de haber tenido éxito la creación y en caso contrario el valor cero (0), devolviendo la información del error a través de GetLastError, tal y como nos tiene acostumbrados Windows.

Si por ejemplo ya se hubiera creado con anterioridad el Evento, deberíamos en principio hacer uso de la función *OpenEvent( )* para acceder al mismo desde otro proceso, ya que desde el mismo, nos bastaría con guardar en una variable el Handle al Manejador del Evento y hacer uso posteriormente como ahora veremos.

Ya sabemos como se comportará nuestro evento y podemos pasar directamente a observar que hace nuestro método *Execute* en la instancia de la clase TMiBarra:

```
procedure TMiBarra.Execute;
begin
  FreeOnTerminate:= True;
  {Primer punto de espera que introducimos para razonar sobre el
   uso de los eventos. Cuando el hilo llega a esta rutina espera
   de forma indefinida a que sea señalizado el objeto evento. Si
   lo señalizamos mediante SetEvent() el hilo prosigue la ejecución
   mientras no sea "no señalizado" de nuevo, por ejemplo mediante
   ResetEvent()}
  WaitForSingleObject(Marcador, INFINITE);
  while (fPos < fMax) and Not Terminated do //mientras no llegemos al max de la barra
    begin
    {Accederemos de forma segura a los objetos en Form1 a traves
    del método Synchronize( )}
    Synchronize(IncrementaPosicion);
    {Esperamos un máximo de 10 segundos a que sea señalizado el
     objeto evento. Si en ese plazo de tiempo no es señalizado
     de forma automática pasa a estado señalizado e inmediatamente
```

```
     despues a no señalizado de nuevo.}
    WaitForSingleObject(Marcador, 10000);
    end;
end;
```

He incluido dos llamadas a la función que se encuentran remarcadas en color amarillo. Marcador es el Handle al objeto Evento. El segundo parámetro INFINITE, obliga a que no se prosiga la ejecución en tanto el objeto no sea señalizado. En la segunda llamada, el segundo parámetro, obliga a esperar al menos 10 segundos antes de que sea señalizado, y si cumplido el plazo de tiempo no es modificado el estado del objeto, el S.O. libera el hilo tras señalizar el objeto y cambia de nuevo su estado a no señalizado.

Para que podáis probar y entender bien el comportamiento y las diferencias entre *SetEvent* y *PulseEvent* tan solo tenéis que pulsar repetidamente sobre los tres botones, señalizando el objeto Evento que comporta la ejecución del hilo, y viceversa.

No puede ser más sencillo:

- SetEvent(Marcador)

  Señaliza el objeto Evento. Vía libre a su ejecución.

- ResetEvent(Marcador)

  El objeto pasa a estado No Señalizado y por lo tanto se detendrá si encuentra una nueva llamada a la función *WaitForSingleObject( )*

- PulseEvent(Marcador)

  El objeto pasa de estado Señalizado a estado No Señalizado de forma instantánea.

Como curiosidad, y para que veáis el carácter global del objeto Evento, podéis ejecutar dos instancias del ejecutable tras compilarlo, y observar como al señalizar el objeto, ambas aplicaciones liberan la ejecución del hilo retenido.

Es un buen momento para jugar durante un par de minutos con el ejemplo. Intentad reflexionar y sacar cada uno sus propias deducciones. Algunas reflexiones son inmediatas, y ya nos ayudan a adivinar de que forma es posible garantizar que tan solo uno de los hilos, en un momento determinado, tenga acceso al hilo principal. Pero ésto queda para un poco después, tras estudiar e intentar comprender que es un Mutex, y como también nos puede ayudar en la sincronización de múltiples hilos concurrentes.

Pues vamos a eso. Nuestro próximo apartado: los semáforos binarios o Mutex.

Objetos Mutex

Pensaba que era necesario también acompañar este subapartado con algún pequeño ejemplo sobre el que pudiéramos reflexionar y que a su vez fuera sencillo. Situación ideal para castigar mi pobre imaginación, que harta de buscar, al final se decidió por un ejemplo mucho más absurdo aun si cabe que el anterior. Se que sabréis perdonarlo. En la **Figura 2** se puede apreciar el bochornoso ejemplo.

Lo podéis encontrar en las carpetas *d6_obj_mutex* y *d5_obj_mutex*, y las mismos comentarios hechos en el ejemplo anterior nos van a servir. Si vuestro entorno es Delphi 6 elegid el proyecto *pro_mutex.dpr* de la primera de las carpetas. Y si es Delphi 5 o inferior, el archivo de proyecto de la segunda de ellas.

En este caso nuestra aplicación creará tres hilos pertenecientes a distintos descendientes de la clase TThread. Cada uno de ellos hacen pues cosas totalmente distintas. Ese es nuestro supuesto. Para simplificar la implementación del método Execute en las tres instancias es similar y la diferencia entre uno y otro tan solo radica

en el color con el que pintan cada uno de los paneles. Sin embargo, la idea que encierra esto, es que cada uno de los hilos creados van a ejecutar acciones distintas. Ejecutadlo tras la compilación y si queréis podemos empezar a ver que ideas podemos resaltar.

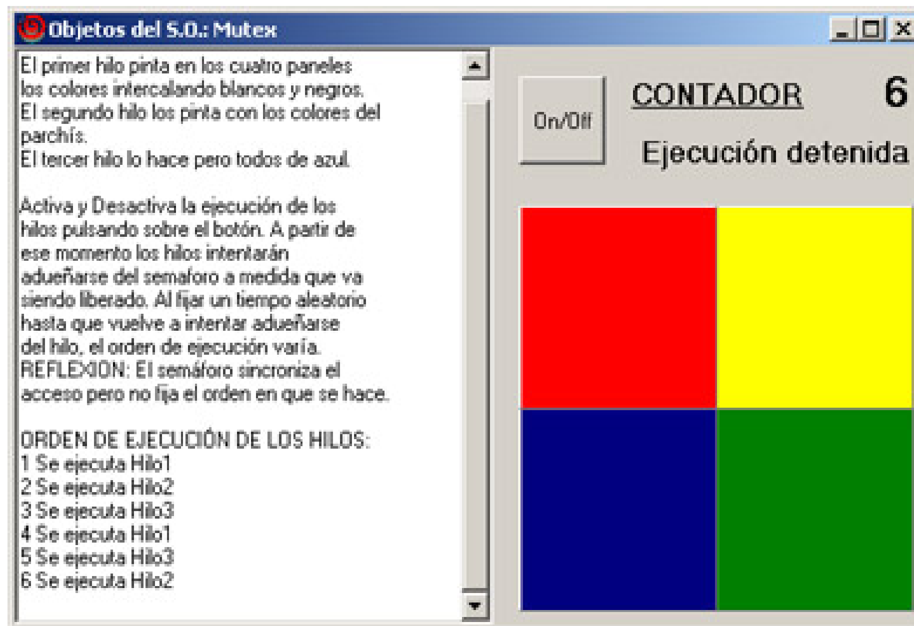

Figura 2 .- Ejemplo muy básico sobre el uso de objetos mutex que acompaña al artículo

Respecto al proceso de creación del Mutex es similar a lo comentado al hablar del evento.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  //Creamos el mutex
  obj_Mutex:= CreateMutex(nil, True, 'MiMutex');
  MiHilo1:= THilo1.Create(True); //creamos los hilos de ejecución
  MiHilo2:= THilo2.Create(True);
  MiHilo3:= THilo3.Create(True);
  ReleaseMutex(obj_Mutex);        //liberamos el mutex
end;
```

La sintaxis de la función *CreateMutex( )* es la siguiente:

```
HANDLE CreateMutex(

    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,
    LPCTSTR lpName
   );
```

Como primer parámetro recibe un puntero a un registro TSecurityAttributes. Nos vale lo comentado respeto al objeto Evento. Cuando este puntero apunta a **Nil** el objeto Mutex creado tiene los atributos de seguridad que se

establecen por defecto. Pero hay que tener en cuenta que este parámetro es ignorado en Windows 9x y tan solo se considera en NT o 2000.

El segundo de los parámetros hace referencia a si el hilo llamante desea convertirse en el propietario del objeto Mutex a crear. En nuestro caso, al tomar como valor True es nuestro hilo principal quien se adueña del objeto creado y no podrá ser obtenido por ninguno de los tres hilos en tanto no sea liberado. Esto se produce un par de rutinas más abajo al invocar la llamada a la función *ReleaseMutex( ).*

Y por fin, el tercer parámetro, al igual que en el ejemplo anterior, hace referencia a un PChar donde se fijará el nombre del Mutex y que como el anterior, será global y único a todos los procesos e hilos que se están ejecutando. Dijimos antes que, como máximo, este nombre podía contener MAX_PATH caracteres. Y olvidamos comentar que sí que se tienen en cuenta las mayúsculas y las minúsculas, o que puede contener cualquier carácter excepto la barra invertida "\".

Respecto al valor de retorno, será, en el caso de tener éxito la creación del objeto, un Handle o Manejador a dicho objeto y en caso contrario cero (0).

Aquí tenemos que tener en cuenta algunas situaciones que nos pueden ser de interés. Si por ejemplo procedemos a su creación y el nombre elegido para el objeto ya existe ¿que sucede?... Pues bueno... Si dicho nombre hiciera referencia a un objeto distinto de un Mutex, tal como un Evento, un Semáforo o un Archivo mapeado en memoria, la función fallaría irremisiblemente, devolviendo la función *GetLastError* el valor *ERROR_INVALID_HANDLE*. La ayuda nos muestra la razón: todos estos objetos comparten el mismo espacio de nombres.

En el caso de que coincida con un objeto Mutex creado anteriormente, la función devolverá un manejador al objeto creado y no tendrá lugar la creación de uno nuevo. Pero se exige que el Mutex al que vamos a acceder tenga acceso de tipo MUTEX_ALL_ACCESS para que esto tenga éxito. De todas formas, lo más habitual es que intentemos acceder a un objeto Mutex ya creado, utilizando la función OpenMutex( ).

Hay un último caso, también aplicable al objeto Evento. ¿Puede ser nulo el nombre del objeto?

Sí. En ambos casos, tanto en el caso del Evento como en del Mutex, pueden ser creados sin nombre, anónimos y como veremos al profundizar en la implementación del método Synchronize( ), es creado un Evento anónimo cada vez que se inicia la ejecución del método, y destruido al salir de el.

Una vez que hemos visto el modo en que son creados los objetos Mutex, podemos pasar directamente a ver una cualquiera de las implementaciones que hemos hecho en el método Execute. Esta es la que hace el segundo de los hilos. Omitimos aquellas rutinas que no son relevantes.

```
procedure THilo2.Execute;
begin
  FreeOnTerminate:= True;
  while Not Terminated do
    begin
    //esperamos a que el mutex sea señalizado
    WaitForSingleObject(obj_Mutex, INFINITE);
    with Form1 do
      begin
      ...
      //grupo de rutinas que acceden a los paneles para cambiar el color
      ...
      end;
    //liberamos el semáforo
    ReleaseMutex(obj_Mutex);
    //esperamos un segundo y menos de tres, al menos antes de volver a iniciar el bucle
    Sleep(Random(2000)+1000);
    end;
end;
```

Estamos en un caso similar al ejemplo que pusimos en nuestro artículo anterior, cuando veíamos como era dibujada una figura geométrica en tanto no fuera finalizado el hilo mediante una llamada al método Terminate. El bucle **while**.. **do** es ejecutado indefinidamente mientras la propiedad *Terminated* no devuelva True. Esto no tiene mayor importancia pero nos va a ayudar a ver la siguiente reflexión.

Tras la creación del formulario y de los propios hilos, el hilo principal libera mediante ReleaseMutex( ) nuestro semáforo. Hay que darse cuenta, que antes de producirse esto, cada uno de los hilos ya se encuentran esperando a que dicho objeto sea liberado, por lo que esta nueva situación hace que el primero de los hilos creados sea el nuevo propietario del Mutex. La espera se produce al llegar a la llamada a la función *WaitForSingleObject( ).* Así pues, tenemos la seguridad, mientras accedemos a los objetos de nuestro form que el resto de hilos, o están ejecutando otras rutinas que no interfieren, o bien van a encontrar una función de espera antes de acceder a una zona "crítica". Podemos prescindir pues de el método *Synchronize( )* ya que mediante este nuevo objeto y las funciones de espera que nos proporciona el API garantizamos que no es sobrescrita cualquier variable, campo de un objeto, etc...

El uso de la función Sleep( ) es anecdótico en este caso ya que tan solo se busca tras la liberación, congelar aleatoriamente dicho hilo de forma que el objeto Mutex sea hecho propietario por cualquiera de ellos, y en un orden también aleatorio. Y esto es una idea que podemos resaltar y que en algún caso puede inducir a error, ya que podemos llegar a pensar que al hablar de Sincronización y de estos objetos, puede estar inducido algún tipo de orden en obtención de la propiedad del semáforo.

Es más, mientras que los objetos Evento pueden cambiar de estado a voluntad del programador, los objetos Mutex son señalizados para el hilo que se convierte en propietario del mismo y el resto de hilos quedan congelados en la espera de la señalización al llegar a la función de espera. Resumiendo: se produce un lucha entre los distintos hilos que acceden al objeto por hacerse propietarios del mismo, y el que lo consigue es ejecutado hasta la llamada a *ReleaseMutex( ).*

Existe una función de espera, *SignalObjectAndWait( )*, que permite señalizar un objeto de tipo Semáforo, Evento o Mutex pero su uso se restringe a NT.

Más ideas y reflexiones al hilo de los dos ejemplos...

Me quedan algunas ideas que pueden ser interesantes. Un par al menos, que voy a intentar razonar porque me parece que merecen resaltarse. De hecho, cuando pensé en los dos ejemplos, eran aspectos que ya tenía en cuenta comentar, pero que he dejado olvidados en el hilo narrativo del artículo.

Veamos el primero. Al analizar el método Execute en el ejemplo que hacemos referencia al Evento, figura tras la llamada a la función de espera, la invocación al método *Synchronize(IncrementaPosicion)*. En un principio y mientras jugaba con las llamadas a una y otra función, tuve la sana ocurrencia de desplazar una de las funciones de espera dentro de la implementación del método *IncrementaPosicion,* que es precisamente el método ejecutado de forma segura. ¿Que pasó? Lo que tenía que pasar... lo que denominamos un bloqueo.

El por qué se produce el bloqueo en esas circunstancias no es difícil de entender: La función de espera *WaitForSingleObject( )* espera a que sea señalizado el objeto Evento, por lo que congela dicho hilo de ejecución, al que llamaremos B (por poner un nombre), y que circunstancialmente se halla dentro de una zona crítica... esto implica el hilo primario no puede ser ejecutado en tanto no se produzca la salida de la zona crítica de B y por lo tanto no podemos entregar la señal desde los botones creados a tal efecto. Es lo que podemos llamar un bloqueo mortal, tonto al fin y al cabo pero mortal de necesidad, ¡digo yo...!

La otra reflexión o pensamiento hace referencia a una idea ya comentada anteriormente, pero que vale la pena verla en funcionamiento. El uso del objeto Mutex nos permitía acceder de forma segura al hilo principal sin hacer uso de las zonas críticas. Sin él, o sin otro objeto que garantice la sincronización y la concurrencia de los hilos, cualquier resultado puede ser esperado. Nos basta para verlo en el ejemplo del Mutex, hacer lo siguiente: comentad las funciones de espera y las de liberación del Mutex, comentad también las *Sleep( )* para que se aprecie claramente lo que queremos ver. Al lanzar la ejecución los tres hilos accederán de forma concurrente al *caption* de

la etiqueta2, que nos sirve de contador, para incrementarlo. Y tras hacerlo, salvarán su valor en una linea del componente *Memo1*, de la clase TMemo. Mirad las lineas del TMemo tras detener su ejecución o bien en el fichero que se salva al final: la conclusión es que presentan presentan saltos en el contador. Y esto se produce porque no hay nada que garantice la concurrencia, tal y como imagináis.

Respecto al objeto Semáforo, del que todavía no hemos hablado, y que básicamente lo podéis entender como un Mutex, en donde se permite el acceso a la propiedad del objeto de mas de un hilo, lo podemos dejar si os parece para el final de la serie. Esta característica, que se pueda acceder por mas de un hilo, hace que tenga connotaciones distintas en cuanto su uso, puesto que parece más adecuado en la interacción de los procesos que en el uso que ahora estamos dando a dichos objetos. Lo intentaremos ver con detenimiento.

Y ya por fin el dichoso Synchronize( )...

¿Hay algo más odiado para mí en estos momentos que este método...?. No lo dudéis. Tras varias horas maldiciéndolo una y otra vez, porque no acertaba a ver claramente qué es lo que estaba sucediendo entre bastidores, salvo lo que aparentemente dejaba entrever...

Y al final se encendió esa lucecita que tanto nos alegra en las noches de insomnio. En esos momentos, se puede decir de uno que es feliz...

No dejemos escapar esa gota de inspiración y vamos a intentar comprender que es lo que está pasando y porque algunos autores nos comentan que el método *Synchronize( )* es ejecutado en el contexto del hilo primario. Empecemos desde el mismo momento en que éste es llamado:

```
procedure TThread.Synchronize(Method: TThreadMethod);
var
  SyncProc: TSyncProc;
begin
  SyncProc.Signal := CreateEvent(nil, True, False, nil);
```

Este es nuestro primer protagonista. Estamos frente a la creación de un objeto Evento de carácter anónimo, como nos manifiesta nuestro cuarto parámetro, con los atributos de seguridad por defecto. El segundo parámetro nos indica que será desactivado automáticamente por el S.O. y por fin, el tercero, que se inicia en un estado no señalizado y por lo tanto, la ejecución del hilo, en principio no se producirá hasta que el objeto sea señalizado mediante *SetEvent( )* o *PulseEvent( )*.

Primer punto importante. El Handle o Manejador del Evento es guardado en el segundo campo del registro *SyncProc* cuyo tipo es TSynProc.

```
             TSyncProc = record
                 Thread: TThread;
             {$IFDEF MSWINDOWS}
                 Signal: THandle;
             {$ENDIF}
             {$IFDEF LINUX}
                 Signal: TCondVar;
             {$ENDIF}
               end;
               PSyncProc = ^TSyncProc;
  try
    EnterCriticalSection(ThreadLock);
```

Entramos en la primera de las zonas críticas y por lo tanto, mientras estemos en ella, el resto de hilos quedarán esperando que este salga de ella.

```
try
  FSynchronizeException := nil;
```

Inicializamos el campo *FSynchronizeException*,

```
  FMethod := Method;
```

La asignación del parámetro *Method* al campo *FMethod* del hilo es el primero de los pasos que nos ayudan a encaminarnos hacia el resultado final. Vamos a guardar en este campo la dirección del método que debe ser ejecutado y que entregamos como parámetro dentro de *Synchronize( )*. Veremos después cómo se hace uso de esto, pero os adelanto que es precisamente esta dirección, la que nos va a ayudar a ejecutar el método en el contexto del hilo principal.

```
  SyncProc.Thread := Self;
  SyncList.Add(@SyncProc);
```

Y este es el segundo punto importante. Asignamos el primer campo del registro SynProc, toda vez que ya tenemos almacenada la dirección del método a ejecutar. Este campo, Thread, se corresponderá como se puede pensar con nuestro objeto hilo actual (Self).

¿Qué se ha conseguido al llegar a este punto?

Llegamos a la idea central y debemos entenderla bien. Daos cuenta de que para poder ejecutar desde el contexto del hilo principal, necesitamos que éste sea capaz de encontrar la dirección del método y por lo tanto, ya cobra sentido la creación de la lista de punteros SyncList (TList). Al ser este objeto, global a los dos hilos, cada uno de los hilos creados se sirve de ella para entregar la dirección de dicho método, y será nuestro objeto Application el que acceda al puntero y proceda a la ejecución del método.

```
  ProcPosted := True;
  if Assigned(WakeMainThread) then
    WakeMainThread(Self);
```

He aquí la madre del cordero... que me quitó varias horas de sueño. Os puedo asegurar que me tuvo en vilo varias veces sobre la conveniencia de haber iniciado la serie; entre otras cosas porque no veía la relación entre el método *Synchronize( )* y *CheckSynchronize,( )*. Es fácil cuando se tiene amplia documentación, o se es un "genio" (y yo ni una cosa ni la otra), adivinar que narices está pasando en este punto de la ejecución del programa. La ayuda que nos ofrece Delphi no parece demasiado clara y tampoco aparece anotación alguna en las fuentes que nos indiquen que es lo que se pretende. Dice la ayuda:

Represents a method (event handler) that is forced into the main thread's queue.

Posiblemente para una persona que hable y lea correctamente el idioma inglés puedan ser fácil las interpretaciones, pero para un español, con conocimientos limitados de la lengua inglesa, se convierte en una pequeña hazaña, donde lo más fácil es quedarse con una interpretación muy personal y subjetiva de lo que se cree entender... ¡Lo que nos lleva a recordar a los señores de Borland lo necesario de tener la ayuda en linea en el propio idioma!

Así que, como se suele decir, tenemos que buscarnos la vida dado que la ayuda en linea no resulta de mucha ayuda en este momento.

Vamos a poner un punto de parada para ver donde nos lleva. Atentos...

```
procedure TApplication.WakeMainThread(Sender: TObject);
begin
  PostMessage(Handle, WM_NULL, 0, 0);
end;
```

¿Que quiere decir esto?

Vamos a ver. La misión es enviar un mensaje y tras el análisis y la observación directa del valor Handle se nos indica que estamos encolando un mensaje cuyo destinatario es el objeto TApplication, el proceso principal. Le estamos enviando el mensaje WM_NULL.

Así pues tras varios intentos, acabaremos encontrando que el mensaje WM_NULL es procesado por el procedimiento *WndProc(var Message: Tmessage)* del objeto Application, uno de los centros neurálgicos de mayor actividad, donde se procesan una buena parte de los mensajes que se reciben. Esto se produce para vuestra curiosidad en el módulo *Forms.pas.*

Y éste llama al método *CheckSynchronize,* que realmente se ejecuta en el contexto del hilo primario como veremos. Podemos ya salir de la zona crítica y...

```
      LeaveCriticalSection(ThreadLock);
      try
        WaitForSingleObject(SyncProc.Signal, INFINITE);
```

dejar nuestro hilo actual, esperando que sea señalizado.

```
      finally
        EnterCriticalSection(ThreadLock);
      end;
```

Momento en el que vuelve a entrar en una zona critica (es decir, cuando el hilo principal ha ejecutado el método desde *CheckSynchonize*)

```
    finally
      LeaveCriticalSection(ThreadLock);
    end;
```

Para salir de la misma al finalizar su ejecución.

```
  finally
    CloseHandle(SyncProc.Signal);
```

Finalmente, la llamada a CloseHandle, eliminará el Handle al objeto Evento, puesto que este es destruido automáticamente por el S.O.

```
  end;
  if Assigned(FSynchronizeException) then raise FSynchronizeException;
end;
```

Al final del proceso, si se ha producido alguna excepción será lanzada para su captura.

Ahora y tras ver que encierra el método CheckSynchronize daríamos por cerrado el ciclo de sincronización. Nos quedan cuatro lineas... y nos vamos con viento fresco.

La primero que hay que tener en cuenta es que este método es ejecutado por el objeto Application tras recibir y procesar el mensaje WM_NULL. En este momento ya estamos en el contexto del hilo principal.

Veamos la implementación del método:

```
function CheckSynchronize: Boolean;
var
  SyncProc: PSyncProc;
begin
  if GetCurrentThreadID <> MainThreadID then
    raise EThread.CreateResFmt(@SCheckSynchronizeError, [GetCurrentThreadID]);
  if ProcPosted then
  begin
```

Tras confirmar que la condición *ProcPosted* es verdadera, que nos indica que Synchronize puso el mensaje y debe ser ejecutado el método...

```
    EnterCriticalSection(ThreadLock);
```

Dado que vamos a acceder a la lista de punteros, a la que tienen acceso el resto de hilos, debemos proteger dicho acceso mediante una zona crítica.

```
try
  Result := (SyncList <> nil) and (SyncList.Count > 0);
```

Si se ha creado el objeto SyncList y tiene algún item *(count > 0)*

```
  if Result then
  begin
    while SyncList.Count > 0 do
    begin
      SyncProc := SyncList[0];
      SyncList.Delete(0);
      try
        SyncProc.Thread.FMethod;
      except
        SyncProc.Thread.FSynchronizeException := AcquireExceptionObject;
      end;
      SetEvent(SyncProc.signal);
    end;
```

Iniciamos un bucle en el que serán ejecutados cada uno de los métodos cuya dirección de memoria fue guardada en el campo *FMethod* del Thread y que ahora encuentra el objeto Application desde el primer campo de la estructura *SynProc.* La ejecución se produce al señalizar el evento con SetEvent, dado que es señalizado el objeto Evento de cada uno de los hilos que esperan la señal.

Al mismo tiempo y una vez hecho esto, es eliminado cada item, por lo que, el bucle finaliza cuando ha ejecutado todos ellos y queda la lista vacía.

```
    ProcPosted := False;
```

Ponemos a false la condición y

```
  end;
finally
  LeaveCriticalSection(ThreadLock);
```

nos salimos de la zona crítica con el trabajo bien hecho y cumplido...

```
    end;
  end else Result := False;
end;
```

Resumiendo para finalizar.

Con este último apartado que hemos visto, ya tenemos una buena idea de como encapsula la clase TThread las funciones del Api sobre hilos de ejecución, dado que hemos recorrido paso a paso, y pienso que con un buen nivel de detalle, desde la creación del objeto hasta su destrucción. Os dejo para vosotros, el análisis de lo que sucede al ejecutarse el método *WaitFor* de nuestro hilo, si es que tenéis algo de curiosidad y más ganas que yo ahora mismo... Y si alguien se decide a verlo desde luego no le voy a enviar un jamón, os lo garantizo, pero casi con toda seguridad encontraremos algún hueco en los próximos artículos para reflejarlo y exponerlo. ¿Os animáis?

¿Qué nos espera en el número 14 de Síntesis?

Nuestra próxima parada, si Dios quiere, nos llevará a hablar de las prioridad en la ejecución de hilos y procesos, algo necesario si pretendemos que no todas las tareas desempeñadas por ellos sean o tengan la misma importancia. Así que cuidaos mucho y hasta entonces.

Notas Bibliográficas:

La documentación sobre la que me he basado en la preparación de los artículos que hasta este momento componen la serie, es básicamente el código fuente del módulo classes.pas de la VCL en Delphi 6 y los ficheros de ayuda en línea sobre el API. En el artículo de introducción me fue de mucha ayuda la lectura de "La Guia del Programador para el uso de la Api de Win32" de Dan Appleman y en el resto, he agradecido la lectura de "El Nucleo del Api de Win 32" en la colección de los tomos de Delphi y editado en España por DanySoft, así como la del libro de Texeira y Pacheco "Guía de Desarrollo Delphi5" o algunas de las ideas extraídas de "Delphi 5" de Francisco Charte, en los apartados que se refieren a la creación de aplicaciones multihilo. Mi más sincero agradecimiento y admiración.