

TThread II

Nuestro punto de parada hoy: la implementación de la clase TThread en Delphi 6. Demos rienda suelta a nuestra curiosidad.

El primer artículo de esta pequeña serie sobre hilos ejecución, y más concretamente sobre la clase TThread en Delphi 6, nos servía como introducción al tema, presentándonos algunos de los protagonistas sobre los que nos iremos centrando a lo largo de éste y de posteriores capítulos. Comentábamos en aquel momento que no nos era posible profundizar sobre todos ellos como lógicamente podríamos desear, pero que en la medida de nuestras posibilidades, haríamos lo posible para tratar los conceptos con la mayor claridad, incorporando algunos ejemplos que nos pudieran servir de referencia. También vimos en esa introducción, el concepto de multitarea en nuestro sistema operativo y los problemas que la incorporación de nuevos hilos de ejecución iba a descubrirnos. Era el punto de encuentro hacia la sincronización de múltiples hilos, que nos llevaría a conocer la existencia de las funciones y objetos que nos facilita el Api de Windows: uso de Secciones Críticas, Semáforos, objetos Mútex, etc.

Y para finalizar, y tras introducir los primeros conceptos sobre la clase TThread, se abordó el ejemplo que incorpora Delphi en la carpeta “..\Demos\Threads”. Cerrando así ese capítulo primero de la serie.

Pero todo aquello fue... Y ahora disponemos de varias hojas en blanco que emborronar y muchos minutos por delante.

La idea en este segundo artículo, es profundizar en el código fuente que nos aporta la clase TThread. Pero quizás, paralelo al estudio y al análisis del mismo, nos puede interesar disponer de un ejemplo que pueda aportar un poco más de claridad en la creación y uso de hilos de ejecución. Y eso de los ejemplos, para las personas que tenemos tan poca imaginación, es algo complicado. ¡Intentémoslo!

Nos acompañamos de un ejemplo...

Pues sí. En realidad me parece muy oportuno hacerlo así, de forma que podamos, al tiempo que desmenuzamos el código fuente de la clase, verlo reflejado en un ejemplo sencillo y fácil de comprender. En ese sentido, el visto en el capítulo anterior, adolecía de claridad. Y es uno de los motivos que me puede haber llevado a plantearme algo mucho más sencillo.

Veamos. Esta es mi propuesta:

Vamos a crear un nuevo hilo de ejecución para hacer una tarea con bastante poco sentido, pero que resulta a la postre, artística. Dicho hilo dibujará sobre nuestro *form* una figura geométrica, dándole una ligera animación mediante el giro continuo de la misma. La figura geométrica a dibujar la tenéis en la **figura 1**. Queremos tener libertad para establecer su tamaño y su aspecto desde nuestra aplicación, así como la velocidad de giro. En este momento me parece importante no introducir ejemplos en los que se reproduzca situaciones de creación de

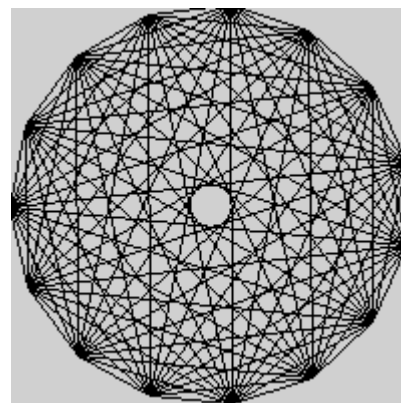


Figura 1 - Figura geométrica que deseamos pintar sobre el Form.

múltiples hilos concurrentes, dado que el objetivo marcado es en cierta forma estudiar el ciclo de vida del nuevo hilo creado. Para tal menester, incorporaremos a futuros artículos de la serie, un ejemplo de mayor complejidad y que voy a compartir con mi compañero **José Manuel Navarro**. Dicha aplicación, mucho más elaborada, tendrá como finalidad implementar un buscador de ficheros, en el que se crean y concurren un numero indeterminado de hilos. Y por cierto... ahora que nombro a mi compañero, os comento el interés de la serie que inició sobre el Api de Win32, y que le ha llevado en este número a tratar el tema de la estructura de memoria denominada habitualmente “montones”.

```

procedure FiguraGeometrica(Canvas: TCanvas; Origen: TPoint; Num_Vert, Radio: Integer;
Color: TColor; AnguloInicio: TGrados);
var
  Coordx1, Coordx2, Coordy1, Coordy2: Integer;
  IntBucle1, IntBucle2: Integer;
  Angulo, IncAngulo: Real;
  ColorTmp: TColor;
begin
  ColorTmp:= Canvas.Pen.Color;
  Canvas.Pen.Color:= Color;
  IncAngulo:= 2 * PI / Num_Vert;
  Angulo:= AnguloInicio;
  for IntBucle1:= 1 to Num_Vert do
    begin
      for IntBucle2:= 1 to Num_Vert - 1 do
        begin
          Coordx1:= Round(Origen.x + (Radio * cos (Angulo)));
          Coordy1:= Round(Origen.y + (Radio * sin(Angulo)));
          Coordx2:= Round(Origen.x + (Radio * cos (Angulo + (IncAngulo * IntBucle2))));
          Coordy2:= Round(Origen.y + (Radio * sin(Angulo + (IncAngulo * IntBucle2))));
          with canvas do
            begin
              moveto(Coordx1, Coordy1);
              lineto(Coordx2, Coordy2);
            end;
          end;
          Angulo:= Angulo + IncAngulo;
        end;
      Canvas.Pen.Color:= ColorTmp;
    end;
  end;

```

Listado 1. Algoritmo que nos dibuja la figura geométrica.

En el **listado 1**, disponemos el algoritmo que hemos implementado para obtener la figura geométrica, obtenido de viejos apuntes de programación sobre un libro de Basic y ligeramente modificado.

Nos resta conocer la interfaz que declara la clase TDibujante, descendiente de la clase TThread, como podéis ver en las líneas inferiores. A través de ella será instanciado un objeto capaz de crear un nuevo hilo de ejecución.

Todo este código se encuentra en la carpeta de ficheros que acompaña al número actual de la Revista y que tiene como nombre “Thread2.zip”. En este caso, la clase TDibujante esta declarada en el mismo módulo que la aplicación que la utiliza y que tiene como nombre “hilos.pas”.

```

type
  TGrados = 0..359;
  TVelocidad = 100..2000;

  TDibujante = class(TThread)
  private
    FGradoActual : TGrados;
    FWidth       : Integer;
    FHeight      : Integer;
    FColor       : TColor;
    FBitmapTemp  : TBitmap;
    FVelocidad   : Integer;
  procedure DibujaFigura;
  procedure BorraFigura;
  public
    constructor Create(AWidth, AHeight: Integer; AColor: TColor; AVelocidad: TVelocidad);
  reintroduce; overload;
  destructor Destroy; override;
  procedure Execute; override;
  procedure AjustarVelocidad(NewVelocidad: TVelocidad);
  end;

```

Recordad como era muy fácil crear un nuevo descendiente de la clase TThread. En el artículo de introducción lo vimos. Bastaba elegir en el menú principal la opción <File | New > y tras emerger la ventana con los diferentes objetos que podemos crear, elegir “Thread Object” mediante un doble click sobre dicho icono. Y ya teníamos un nuevo módulo donde desarrollar nuestra nueva clase.

Pero en esta ocasión no lo hemos hecho así. Hemos creado un nuevo proyecto, que nos va a servir para el ejemplo, y en la declaración de tipos de su interfaz, en el módulo “Hilos.pas”, procedemos a añadir la clase TDibujante, con la idea de que los lectores menos iniciados se puedan percatar de como también es posible hacerlo así, aunque pueda técnicamente ser más correcta la creación en módulos separados. El por qué comento esto, es quizás para que se tenga constancia de que al hacerlo así, en un mismo módulo, perdemos algunas de las características que demanda la técnica de encapsulación de los objetos. Es decir, podemos desde cualquiera de nuestros métodos en *Tfrm_PruebaHilos* acceder a los campos y métodos del objeto *Dibujante*, privados o no privados, lo cual no debería ser técnicamente correcto, al menos en principio. Pueden dejar de tener sentido la separación {privado|protegido|publico} dada la relación de “amistad” entre ambas clases por el hecho de pertenecer al mismo módulo. Nosotros respetaremos las normas de “convivencia” y solo accederemos al objeto a través de los métodos que hace públicos, en ese intento de redimir un poco nuestra conciencia.

En la **figura 2** podéis ver la aplicación de ejemplo que nos servirá para curiosear y comprender el código fuente de la clase TThread, o al menos esa es la sana intención que se persigue.

Pero podemos volver sobre una idea que intentábamos dejar bien clara en el artículo anterior. La creación de un nuevo hilo de ejecución mediante la clase TThread es sin lugar a duda sencilla. El ciclo de vida de nuestro nuevo hilo comienza con la invocación del constructor del objeto TDibujante para lo cual nos dirigiremos al momento de su creación desde el *form* principal.

```

procedure Tfrm_PruebasHilos.FormCreate(Sender: TObject);
begin
  ... // creamos un nuevo hilo
  Dibujante:= TDibujante.Create(Width,
                                Height,
                                clBlack,
                                trb_MaxBucle.Position);
  Dibujante.Resume; // Lanzamos la ejecución del nuevo hilo
end;

```

Queda clara, pues la dinámica de trabajo que perseguimos. Ya tenemos un punto de partida para iniciar nuestro análisis, como es la creación de la instancia *Dibujante* de *TDibujante* en el *form* y que nosotros vamos a incluir en el método *FormCreate*, asignado al evento que se ejecuta previo a la creación del formulario. Un lugar como otro cualquiera, todo hay que decirlo...

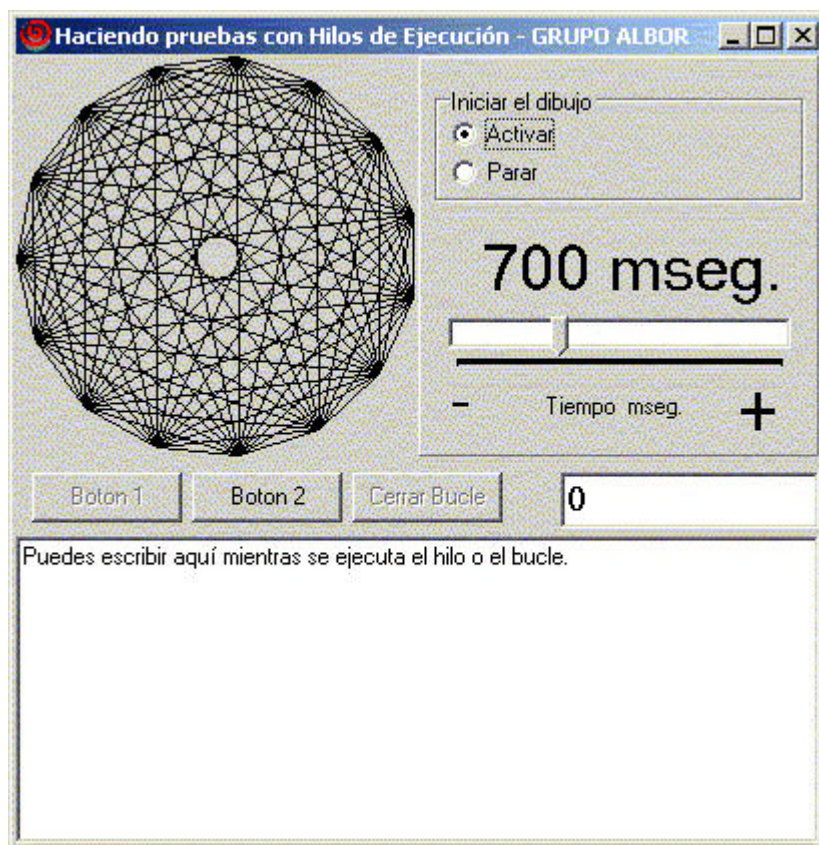


Figura 2 - Ejemplo que nos ayudará a introducirnos en la creación de hilos mediante la clase Tthread

Seguir la pista al nuevo hilo...

Nuestro siguiente punto de parada, toda vez ejecutada la rutina de creación del objeto *Dibujante*, `Dibujante:= TDibujante.Create(Width, Height, clBlack, trb_MaxBucle.Position);` es analizar el constructor de la clase, detallado a continuación.

```

constructor TDibujante.Create(AWidth, AHeight: Integer; AColor: TColor; AVelocidad:
TVelocidad);
begin
    inherited Create(True); //llamamos al constructor en la clase ascendente
    FWidth:= AWidth;        //guardamos el ancho de la imagen
    FHeight:= AHeight;      //guardamos la altura de la imagen
    FColor:= AColor;        //guardamos el color elegido para el pincel
    FVelocidad:= AVelocidad;
    FBitmapTemp:= TBitmap.Create; //creamos un bitmap dinámicamente para evitar parpadeos
    FBitmapTemp.Height:= 200;
    FBitmapTemp.Width:= 200;
    FBitmapTemp.Canvas.Brush.Color:= clBtnFace;
    Self.FreeOnTerminate:= True; //al finalizar se destruirá automáticamente
end;

```

Dicho constructor entrega como parámetros al objeto que se va a crear, los valores iniciales que afectan al tamaño del dibujo, el color y la velocidad con la que queremos que gire, y que eran parte de los requisitos que nos habíamos marcado. Podríamos, dado que podemos acceder al hilo principal de nuestra aplicación y en concreto a las variables y campos de nuestro form de forma segura a través del método `Synchronize()`, haber dejado para dicha ocasión la asignación de tales valores. Tampoco habría problema en ello, ni en considerarlo así de ser necesario, pero dado que no van a cambiar durante el tiempo de vida de nuestra aplicación, puede ser el lugar ideal nuestro constructor para la entrega de dichos parámetros, independientemente de que su acceso, mediante el constructor, se va a hacer en el contexto del *thread* primario, mientras que hacerlo desde el método `Execute` del objeto obligaría a la protección mediante el uso de Secciones Críticas, que al fin y al cabo es lo que internamente introduce el método `Synchronize()`.

Y llegamos al constructor heredado, invocado mediante

```
inherited Create(True);
```

y cuyo parámetro Booleano, con valor Verdadero, produce que el nuevo hilo no sea ejecutado inmediatamente sino que sea suspendido, en tanto no se invoque el método `Resume`. El motivo de hacerlo así es la necesidad de inicializar los campos privados de la clase `TDibujante` y la asignación mediante la cual decimos al objeto que se destruya toda vez que se ha ejecutado y finalizado el código implementado en `Execute`. Es la rutina:

```
Self.FreeOnTerminate:= True;
```

Si os surgen dudas sobre la palabra reservada **`inherited`**, o bien comprendiendo su concepto queréis conocer algún detalle más que os pueda pasar desapercibido puede ser un buen momento para leer un artículo aparecido en nuestro número anterior y que firmaba nuestro compañero Tavo Ibaceta, cuyo título es *Sobreescritura de propiedades*. Un buen artículo y que, sin lugar a duda, os ayudará a profundizar en conceptos tales como la declaración de un método como virtual y la anulación de métodos.

Pero podemos seguir el paso que nos demanda la ejecución del constructor en la clase ascendente, `TThread` y que responde a implementación que detallamos a continuación y en donde aparecen ya por primera vez, los primeros detalles en forma de directivas de compilación, que nos indican la consideración de un nuevo SO como Linux en nuestro código:

```

constructor TThread.Create(CreateSuspended: Boolean);
{ $IFDEF LINUX }
var
    ErrCode: Integer;
{ $ENDIF }
begin
    inherited Create;
    AddThread;

```

```

FSuspended := CreateSuspended;
FCreateSuspended := CreateSuspended;
{$IFDEF MSWINDOWS}
FHandle := BeginThread(nil, 0, @ThreadProc, Pointer(Self), CREATE_SUSPENDED, FThreadID);
if FHandle = 0 then
    raise EThread.CreateResFmt(@SThreadCreateError, [SysErrorMessage(GetLastError)]);
{$ENDIF}
{$IFDEF LINUX}
sem_init(&FCreateSuspendedSem, False, 0);
ErrCode := BeginThread(nil, @ThreadProc, Pointer(Self), FThreadID);
if ErrCode <> 0 then
    raise EThread.CreateResFmt(@SThreadCreateError, [SysErrorMessage(ErrCode)]);
{$ENDIF}
end;

```

Un poco de aventura en mi vida...

Os pido que de momento ignoremos el código que afecta a Linux, encerrado entre cada par de directivas de compilación “`{ $IFDEF LINUX}... <CODIGO> ...{ $ENDIF}`”. En principio y para no liarnos demasiado, nuestra perspectiva será la del S.O. de Windows y tan solo extenderemos nuestros comentarios cuando afecte a dicho sistema operativo y siempre que nos pueda ser realmente de interés. Mas adelante, retomaremos el código de Linux, intentando cumplir la promesa que nos hacíamos en el artículo de introducción respecto a realizar el análisis también desde la perspectiva de Linux. Como reza el título del apartado..., un poco de aventura en nuestra vida, que no nos falte.

La primera línea que nos va a interesar es la que ejecuta el procedimiento *AddThread*. Dicho procedimiento tiene como misión dos puntos claros y definidos: por un lado, creará una lista de punteros, un objeto de la clase TList, con ámbito global, *SynList*, cuya razón de ser es la de almacenar en cada uno de sus ítems un puntero a una estructura de tipo TSyncProc, cuya definición es como sigue:

```

TSyncProc = record
    Thread: TThread;
    {$IFDEF MSWINDOWS}
    Signal: THandle;
    {$ENDIF}
    ...

```

Dicha creación se llevará a cabo con el primero de los hilos creados, momento en que la variable toma el valor inicial **nil** y el contador de hilos **ThreadCount** vale 0.

Pero ¿cual es el motivo para hacerlo así? Adelantándonos un poco a los métodos que van a hacer uso de este record o registro, como puede ser tanto la función *CheckSynchronize*() como la función *ThreadProc*(), en dicha estructura citada, almacenaremos respectivamente en el primero de los campos la variable *Self*, que identifica al objeto mismo, y un segundo campo cuyo tipo THandle es el manejador devuelto al objeto de evento creado. Dicho objeto de evento es creado en la primera de las rutinas que implementa el método *Synchronize*(), es una función que nos facilita el Api de Windows dentro del grupo de funciones cuyo cometido es sincronizar los accesos concurrentes de múltiples hilos. Es decir... para que podamos empezar a tenerlo un poco más claro, resumiremos en varios puntos el ciclo de vida del nuevo hilo creado:

- *Primer punto* Creación del thread. Ejecución del método Resume si el parámetro CreateSuspended toma valor Verdadero.
- *Segundo punto* La creación lanzará la ejecución del método ThreadProc() Esto será consecuencia de la llamada a la función del Api CreateThread() que se produce dentro de la función BeginThread().

- *Tercer punto* Este última función invocará internamente a `Execute` que a su vez...
- *Cuarto punto* Puede invocar `Synchronize()` y éste mediante dicha lista de punteros invocar el método que accede al contexto del thread primario. Pudiendo hacer uso del manejador al evento para señalar el hilo ejecutado.
- *Quinto punto* `Execute` finaliza el resto de código pendiente y libera el objeto en caso de que asignemos a `True` la propiedad `FreeOnTerminate`.

Es un esquema bastante simplificado del orden en que se van a ejecutar los distintos métodos que intervienen en la creación de nuestro objeto Dibujante. ¿Os hacéis ya una idea -con más o menos detalle- de la necesidad de la variable global `SyncList`, que apuntará a una lista de estructuras, y que va a ser realmente una de las razones por las que se consigue sincronizar múltiples hilos de ejecución de forma concurrente?

Nos quedamos en el método `AddThread`:

```
procedure AddThread;
begin
  EnterCriticalSection(ThreadLock);
  try
    if (ThreadCount = 0) and (SyncList = nil) then
      SyncList := TList.Create;
    Inc(ThreadCount);
  finally
    LeaveCriticalSection(ThreadLock);
  end;
end;
```

Así pues, tras la creación de la lista, incrementamos el contador, una variable también global, de tipo entero, y a la que deberemos acceder mediante el uso de Secciones Críticas, protegiendo dicho contador de que pueda ser sobrescrito por el proceso de creación de un segundo hilo de ejecución que acceda simultáneamente.

Pero sigamos analizando el código del procedimiento de creación del objeto de la clase `TThread`, toda vez que ha sido ejecutado el método `AddThread`.

El campo `Fsuspended`, almacenará el valor de la propiedad `Suspended`, que nos permitirá mediante el método `SetSuspended()` suspender o activar el hilo de ejecución. Ignoramos también deliberadamente su uso en la función `ThreadProc()` para la compilación en el S.O. de Linux.

```
FSuspended := CreateSuspended;
```

`FcreateSuspended` representa el parámetro inicial de lanzamiento inmediato del hilo de ejecución o la suspensión indefinida hasta la invocación de `Resume`. Toma entonces el valor Verdadero/Falso introducido por `CreateSuspended` y que necesitamos almacenar por la razón que vamos a comentar líneas más abajo.

```
FCreateSuspended := CreateSuspended;
```

Y a la altura ya de este punto, deberíamos iniciar el análisis de la función `BeginThread()` pero dado que ya hemos citado el uso de Secciones Críticas, parece lo más lógico detenernos un momento y saber algo más sobre las mismas. Vamos a ver que nos dice el Api de Windows.

Secciones críticas: una parada para saber algo más sobre ellas...

Respecto a las Secciones Críticas, comentábamos en el artículo de introducción la necesidad de proteger los recursos globales cada vez que un hilo hiciese uso de ellos, impidiendo que otro hilo creado y activo intentase acceder a dicho recurso simultáneamente. Para ello, necesitamos de una función que marque el inicio de la Sección Crítica y otra en la que se abandone la misma, dejando el camino libre al segundo hilo para su acceso al recurso.

Veamos las funciones implicadas en el orden habitual de uso:

```
InitializeCriticalSection( var lpCriticalSection: TRTLCriticalSection);
EnterCriticalSection( var lpCriticalSection: TRTLCriticalSection);
LeaveCriticalSection( var lpCriticalSection: TRTLCriticalSection);
DeleteCriticalSection( var lpCriticalSection: TRTLCriticalSection);
```

Siempre que va a ser usada una Sección Crítica necesitaremos declarar una variable de tipo `TRTLCriticalSection`. Nuestra variable en el caso que nos ocupa es `ThreadLock`. Si tenéis curiosidad por saber como es esta estructura y que miembros la componen la podréis encontrar definida en el módulo `Windows.pas` y su definición es tal como sigue:

```
_RTL_CRITICAL_SECTION = record
  DebugInfo: PRTL_CRITICAL_SECTION_DEBUG;
  LockCount: Longint;
  RecursionCount: Longint;
  OwningThread: THandle;
  LockSemaphore: THandle;
  Reserved: DWORD;
end;
{$EXTERNALSYM _RTL_CRITICAL_SECTION}
TRTLCriticalSection = _RTL_CRITICAL_SECTION;
```

Nosotros nos limitaremos a declarar simplemente la variable. Pero se hace necesario, lógicamente que en algún momento, sea inicializada dicha estructura. En el módulo `classes.pas` encontraréis dicha inicialización al final del mismo en la sección **initialization**

```
InitializeCriticalSection(ThreadLock);
```

A partir de ese momento, podemos hacer uso de la misma dentro del módulo mediante las funciones

```
EnterCriticalSection( ThreadLock);
LeaveCriticalSection( ThreadLock);
```

tal y como nos muestra el procedimiento `AddThread()`, haciendo uso de la primera para iniciar la sección crítica que protegerá nuestro recurso, y abandonándola mediante la segunda de ellas. No parece demasiado complicado.

Para finalizar, deberemos proceder siempre a liberar la memoria asociada a la estructura `TRTLCriticalSection` y para ello, el módulo `Classes.pas` hace uso de la sección **finalization** donde se invoca la función

```
DeleteCriticalSection( ThreadLock);
```

A partir de dicho momento, y me refiero concretamente a esta última función, ya no se puede hacer uso de la sección crítica si no es procediendo a inicializar de nuevo la estructura de la forma indicada. De haber sido invocada en otro punto cualquiera del módulo habría que seguir los pasos comentados.

Sin perder el hilo de lo que hacemos...

¿Dónde nos quedamos...?. Ah... Sí. Con tanto hilo nos hemos hecho un verdadero lío.

Hicimos un pequeño alto en el camino para comprender el uso de las Secciones Críticas y nos dejamos interrumpido el constructor de la clase TThread, justo en el momento en el que va a ser creado el nuevo hilo de ejecución. Veámoslo.

```
{ $IFDEF MSWINDOWS }
  FHandle := BeginThread(nil, 0, @ThreadProc, Pointer(Self), CREATE_SUSPENDED, FThreadID);
  if FHandle = 0 then
    raise EThread.CreateResFmt(@SThreadCreateError, [SysErrorMessage(GetLastError)]);
{ $ENDIF }
```

BeginThread es una función declarada en el módulo *System.pas* y que encapsula la creación de un nuevo hilo mediante la llamada al Api de Windows con la función `CreateThread()` que será quien realmente haga el trabajo de creación y lanzamiento del nuevo hilo dentro del proceso. El retorno de dicha función es un manejador al nuevo hilo, un THandle, que será almacenado por el objeto TDibujante en el campo FHandle del ascendiente.

La condición posterior (`FHandle = 0`) nos esta diciendo que ha fallado el proceso de creación del hilo y debe ser lanzada la excepción y limpiado el último error mediante *GetLasError*, tal y como indica la mecánica habitual de trabajo en Windows.

Analicemos el código que implementa la función:

```
function BeginThread(SecurityAttributes: Pointer; StackSize: LongWord;
  ThreadFunc: TThreadFunc; Parameter: Pointer; CreationFlags: LongWord;
  var ThreadId: LongWord): Integer;
var
  P: PThreadRec;
begin
  New(P);
  P.Func := ThreadFunc;
  P.Parameter := Parameter;
  IsMultiThread := TRUE;
  Result := CreateThread(SecurityAttributes, StackSize, @ThreadWrapper, P,
    CreationFlags, ThreadID);
end;
```

Lo primero que hace es declarar una variable puntero a una estructura TThreadRec.

```
PThreadRec = ^TThreadRec;
TThreadRec = record
  Func: TThreadFunc;
  Parameter: Pointer;
end;
```

Una vez declarada la variable, reserva memoria para dicha estructura entregando al primero de los campos un puntero a la función ThreadProc, que será el punto de inicio para la ejecución de todo el código en otro hilo distinto del primario. Al segundo campo se le entrega un puntero al objeto que crea el hilo mediante la variable *Self*. Recordemos la rutina en la que se produce todo esto:

```
FHandle := BeginThread(nil, 0, @ThreadProc, Pointer(Self), CREATE_SUSPENDED, FThreadID);
```

Se procede además a proteger el acceso a la memoria de los hilos creados al poner la variable *IsMultiThread* a True, informando al gestor de memoria que hay mas de un hilo activo antes de lanzar la creación mediante *CreateThread()*.

Y finalmente, es creado el nuevo hilo con su invocación

```
Result := CreateThread(SecurityAttributes, StackSize, @ThreadWrapper, P,  
CreationFlags, ThreadID);
```

Devolviendo como valor de retorno, como ya comentábamos, un manejador (THandle) al hilo creado.

Estamos ante otra de las funciones del Api de Windows.

Crear un nuevo hilo: más comentarios.

Quizás lo más oportuno sea iniciar la casa por los cimientos y centrarnos por unos momentos en el proceso de creación de nuevos hilos desde el Api de Windows. En el módulo *Windows.pas* es definida la función que traducirá los tipos originales de C a Pascal, con la sintaxis siguiente:

```
function CreateThread(lpThreadAttributes: Pointer;  
dwStackSize: DWORD;  
lpStartAddress: TFNThreadStartRoutine;  
lpParameter: Pointer;  
dwCreationFlags: DWORD;  
var lpThreadId: DWORD  
): THandle; stdcall;
```

Analicemos que parámetros son entregados a dicha función:

- `lpThreadAttributes = nil`

Este parámetro recoge un puntero hacia una estructura de registro con información de los atributos de seguridad. Como contiene el valor **nil** toma los valores por defecto.

- `DwStackSize = 0`

Es el tamaño de la pila asociada al nuevo hilo. El valor 0 nos indica que toma el tamaño de pila del hilo primario.

- `lpStartAddress = @ThreadWrapper`

Un puntero a una función implementada en ensamblador en el mismo módulo y que inicia la ejecución del hilo y enlaza con el método y el puntero al objeto en *PThreadRec*.

- `lpParameter = PThreadRec`

Para nosotros representa el código de la función que es ejecutada por el nuevo hilo y el puntero al objeto que lo ha creado.

- `dwCreationFlags = CREATE_SUSPENDED`

Le indicamos que queremos que el hilo, inicialmente sea suspendido. Si hubiera tomado valor 0 el nuevo hilo se ejecutaría inmediatamente pero el valor definido de dicha constante está recogido en el módulo *System.pas*, y concretamente toma valor \$00000004, distinto de 0.

- `lpThreadId = FThreadId`

Esta variable es modificada en el proceso de creación y recibirá el valor del identificador del nuevo hilo. Este identificador es único en todo el sistema. El método `Destroy`, destructor de la clase *TThread* hace uso de este parámetro para identificar el estado del hilo antes de destruirlo.

Un momento, por favor... pienso para mí mismo. Si de lo dicho se deduce que siempre que se crea un nuevo hilo este queda suspendido en tanto no sea invocado el método Resume de la clase TThread, ¿qué sentido tiene el parámetro *CreateSuspended* entregado en el constructor de la clase?

Confieso que yo también me perdí en ese punto. Y la mejor forma que encontré para averiguarlo fue la de establecer puntos de parada y observar que métodos eran llamados y el orden que seguían en su ejecución. Para esto, es necesario establecer la siguiente opción en <Project | Options | Compiler > y marcar la casilla **Use Debug DCUs**

El resultado de tales experimentos fue comprobar como tras la creación del objeto era ejecutado el método AfterConstruction, que anula o sobrescribe el del ascendiente. Si observamos el código:

```
procedure TThread.AfterConstruction;
begin
  if not FCreateSuspended then
    Resume;
end;
```

Si el valor del campo FcreateSuspended es False, el nuevo hilo será ejecutado inmediatamente después de su creación.

Pienso que es un buen punto este hacer un alto en el camino y finalizar nuestro análisis. En nuestro próximo artículo proseguiremos desde aquí, a partir del lanzamiento del hilo mediante *Resume* e iremos desmenuzando el resto de procedimientos y funciones hasta la destrucción de la clase, intentado seguir el ciclo de vida de nuestro ejemplo.

Acabemos de ver el ejemplo, por favor...

Dejábamos creada una nueva instancia de la clase TDibujante y con ella, toda vez que había sido invocado el método Resume del objeto, el nuevo hilo creado iniciará el código que sobrescribe el método Execute. Recordando todo lo comentado en éste y el anterior artículo, será este método sobre el que recaerá de forma directa todas y cada una de las acciones que deba efectuar nuestro hilo. En nuestro caso, deseamos que sea ejecutado de forma continua el dibujado del form mientras la condición de finalización no se cumpla. Y la condición de finalización viene ligada al estado de una variable booleana *Terminated* perteneciente a la clase TThread y que es manipulada por el método público *Terminate* de la misma clase.

```
procedure TDibujante.Execute;
begin
  repeat
    Synchronize(BorraFigura);
    Synchronize(DibujaFigura);
    Sleep(FVelocidad);
  until Terminated;
end;
```

No nos hace falta definir otra variable. El estado de ésta es mantenido tanto si es destruido el objeto de forma automática como si lo hacemos nosotros de forma manual mediante dicho método.

En primer lugar, procedemos a borrar el dibujo anterior, tras lo cual, iniciaremos el dibujo el actual, dejando tras estas dos operaciones, un breve espacio de tiempo que será el que nos marque la velocidad de giro del dibujo.

En el **listado 2** podemos ver el método mediante el que procedemos a dibujar sobre el *Form*, y será dicho método, al igual que anteriormente *BorraFigura*, el que accedería a las propiedades y variables del hilo primario mediante *Synchronize()*. Entregaremos a este último un puntero a nuestro método para que sean protegidos dichos

recursos, dado que la VCL, en su construcción no ha sido diseñada para la concurrencia de múltiples hilos. Quizás se pueda hacer el comentario de que el objeto *canvas*, sí que dispone de una propiedad que nos permite acceder al mismo con seguridad, *Lock* y *UnLock*, pero parecía más apropiado no hacer uso de ellas en referencia a las explicaciones que vamos a realizar en el próximo artículo sobre el método *Synchronize*().

Para finalizar, el destructor de la clase, que libera el bitmap temporal usado para no dibujar directamente sobre el form, en aras de evitar parpadeos de imagen.

```
destructor TDibujante.Destroy;
begin
  FBitmapTemp.Free;
  inherited;
end;
```

Así pues es un buen momento para compilar y ejecutar el ejemplo y sacar algunas conclusiones para finalizar ya este tiempo que estamos compartiendo.

```
procedure TDibujante.DibujaFigura;
var
  NuevoOrigen: TPoint;
  Rect: TRect;
begin
  if FGradoActual = 359 then FGradoActual:= 0
  else Inc(FGradoActual);

  With NuevoOrigen do
    begin
      x:= 100;
      y:= 100;
    end;
  with Rect do
    begin
      Left:= 0;
      Top:= 0;
      Right:= 200;
      Bottom:= 200;
    end;

  FBitmapTemp.Canvas.Brush.Color:= frm_PruebasHilos.Color;
  FBitmapTemp.Canvas.FillRect(Rect);

  FiguraGeometrica(FBitmapTemp.Canvas,
    NuevoOrigen,
    Num_Rad,
    DisRadio,
    FColor,
    FGradoActual);

  frm_PruebasHilos.Canvas.CopyRect(Rect, FBitmapTemp.Canvas, Rect);
end;
```

Listado 2. El método *DibujaFigura* de *TDibujante*, responsable del acceso al hilo primario.

Algunas conclusiones y acabamos.

Hay algún detalle que me parece importante que se pueda apreciar y que sin duda, aquellos compañeros que nunca han utilizado ni conocen el uso de los hilos, les podrá resultar cuando menos curioso. Nuestro *form* principal consta de una zona de dibujo, en la parte superior izquierda. La parte superior derecha conviven un objeto de clase TRadioGroup, para activar o parar el dibujo y que hacen uso de los métodos *Resume* y *Suspend* respectivamente, y un objeto de clase TTrackBar para modificar la velocidad de giro. En la parte inferior disponemos de tres botones y un objeto de clase TMemo. Y es en este punto donde deseo hacer el comentario que me parece de interés:

Suponiendo activo el dibujo, si procedemos a pulsar el botón 2, iniciaremos un bucle *for* que recorrerá el rango de los enteros y que se visualizará en un cuadro de texto a la derecha del botón 3. En ese momento, podemos escribir sobre el componente TMemo gracias a la inclusión del método *ProcessMessages* de *Application*, que vacía la cola de mensajes y permite que los mensajes sean entregados a sus destinatarios.

Hecho esto, debemos proceder a pulsar el botón 1 y emergerá una ventana modal con un texto explicativo. En dicho momento, y mientras no sea cerrada la ventana, el bucle *for* quedará congelado en espera que el hilo primario vuelva a activarse, dado que ha sido bloqueado por la ventana modal. Sin embargo nuestro dibujo sigue tan campante, dibujándose feliz, ignorante de la desgracia que padece nuestro hilo primario y nuestra ventana principal. Cerrada dicha ventana, el bucle restablece su ejecución en el punto en el que se detuvo.

Resulta curioso. Ahora imaginemos que en lugar del bucle *for* disponemos de un algoritmo sumamente complicado, que va a consumir una gran cantidad de tiempo. ¿No sería una buena idea que dicho algoritmo se pudiera ejecutar en un segundo plano, evitando los problemas y las interrupciones que puedan devenir por obra y gracia de nuestro usuario? A su finalización, entregaría el resultado de su cálculo al destinatario del mismo en el hilo primario. Hacerlo resulta sencillo. Ya lo habéis visto. Habría que adoptar ciertas precauciones adicionales que poco a poco, a lo largo de todos estos artículos vamos a intentar conocer.

Nos vemos en el número 13 de Síntesis,