

## TThread IV

Ajustar el nivel de prioridad en la ejecución del hilo creado, declarar una variable como ThreadVar... Son algunas de las cosas que veremos en esta cuarta entrega de la serie.

Si habéis seguido con una pizca de atención el desarrollo de los tres artículos anteriores, casi con toda seguridad os encontraréis con suficientes conocimientos para incorporar el uso de hilos a vuestros desarrollos; a poco claro está, que tengáis una mínima dosis de paciencia. La idea que se perseguía al iniciar la serie, era poner sobre la mesa aquellas ideas y conceptos claves para la comprensión y su uso posterior. Si es así o no, sois vosotros, como lectores y destinatarios de estas líneas, quienes mejor podéis juzgarlo.

Nos sirven pues, estas primeras líneas del artículo, para hacer ese balance en el que sopesamos lo que se ha expuesto con anterioridad y lo que nos resta de camino. Lo que tenemos ya asimilado y lo que nos falta en este pequeño puzzle de ideas. Volvamos la vista hacia atrás:

La serie comienza con un artículo de introducción en el número 11 de Síntesis. Dicho artículo, con fecha de Octubre del 2002, nos sirve para presentar el concepto de Multitarea, y concretamente la que denominamos “Multitarea con derecho Preferente o Forzada”. También nos sirve dicho número para presentar los conceptos de Proceso, de Hilo o Thread, y para mostrarnos finalmente a los principales actores que participarán en lo que se conoce como “Sincronización” de los hilos. Extendíamos pues, sobre ese lienzo imaginario, las primeras pinceladas, como requiere básicamente un artículo de introducción al tema.

El segundo artículo de la serie, dos meses después, nos permitió bucear en la implementación que hace la clase TThread en Delphi 6, y seguimos con el máximo nivel de detalle, cómo eran encapsuladas las funciones que proporciona el API de Windows para la creación de nuevos hilos de ejecución. Desgranábamos, paso por paso, cada una de las llamadas, desde la creación del hilo hasta que este era destruido. Todo esto lo hacíamos desde un ejemplo muy sencillo en el que tan solo dibujábamos una figura geométrica en movimiento. Dada la extensión del artículo (12 páginas) y la densidad de algunos de los conceptos que abordábamos, decidí que aquellos aspectos vinculados a la sincronización quedaran para un momento posterior. Y creo que fue algo que sin duda agradecisteis.

Y llegamos al artículo anterior, de fecha de Febrero del presente año, y ubicado en el número 13 de Síntesis (¡crucemos los dedos!). Lo dicho... Era el momento para ver de cerca, como era posible la sincronización de los hilos que concurrían y para ello, como recordaréis, hicimos parada ante conceptos como “Mutex”, “Evento” o “Semáforo”, recreándonos incluso con pequeños ejemplos anecdóticos del uso de los mismos. Necesitábamos 17 páginas de nuestra revista para verlo con cierta profundidad.

¿Que nos espera en este número que ahora iniciamos? ¿Lo vemos...?

Para empezar, volveremos la vista hacia Delphi 5, y compararemos algunas de las diferencias que existen en la implementación de la clase TThread. No es algo realmente imprescindible, como bien se puede suponer. Es algo transparente para nosotros, como programadores, cuando nos limitamos a invocar los métodos del objeto sin conocimiento directo de cómo realmente lo hace. Sin embargo, cualquier intento de extender la funcionalidad de esta clase, y de querer hacerlo compatible con Delphi 6, puede encontrarse con dificultades... la implementación es distinta aunque ambas obtengan similares resultados.

También vamos a conocer la cláusula *ThreadVar*, que nos permitirá que una variable global sea “compartida” por múltiples hilos sin que su valor sea sobrescrito por cualquiera de ellos, como ocurriría de haber hecho uso de la cláusula *Var*.

Y finalmente, nos quedará la comprensión del concepto de “*prioridad*”, importante si queremos que una o varias tareas obtengan mayor tiempo de ejecución con respecto a otras que concurren dentro de la misma aplicación. Lo veremos con detalle y nos serviremos de un par de ejemplos sencillos donde se pueda ver todo esto con claridad.

## ¿Que ha cambiado con respecto a Delphi 5, por favor...?

```
constructor TThread.Create(CreateSuspended: Boolean);
var
  Flags: DWORD;
begin
  inherited Create;
  AddThread;
  FSuspended := CreateSuspended;
  Flags := 0;
  if CreateSuspended then Flags := CREATE_SUSPENDED;
  FHandle := BeginThread(nil, 0, @ThreadProc, Pointer(Self), Flags, FThreadID);
end;
```

Listado 1. Constructor de la clase TThread en Delphi 5.

Inicialmente, y hablo de cuando me planteaba por primera vez la serie, albergaba la feliz idea de centrarme en la implementación que hacía Delphi 5 de la clase TThread. Me resultaba mucho más sencillo, dado que trabajo habitualmente con esta versión del compilador, hacerlo así. También disponía de mayor cantidad de documentación para poder enfrentarme al artículo. El libro de Teixeira y Pacheco, “Guía de Desarrollo de Delphi 5”, dedica una buena cantidad de páginas a este tema, con profundidad y con algunos ejemplos útiles y didácticos.

Sin embargo, pensé que a vosotros, como lectores, os sería de mayor interés abordar este tema desde la perspectiva de Delphi 6, sobre el que existe -me parece- menor cantidad de documentación escrita. Al menos esa es mi impresión.

A medida que me sumergía en la lectura del código fuente de la clase *TThread*, en el módulo *Classes.pas*, y comparaba con el código facilitado en Delphi 5 para esta misma clase, apreciaba las primeras diferencias, que se justificaban en el hecho de querer hacer nuestro código compatible tanto para el S.O. de Windows como para Linux (Delphi & Kylix): adición de directivas para una compilación condicional y supresión de algunos elementos claves, propios de Windows. De las primeras ya hablamos en nuestros dos anteriores artículos. De los segundos lo haremos ahora, al mencionar la existencia de una ventana oculta *ThreadWindow*, mediante la que se pueden ejecutar de forma sincrónica cada uno de los hilos.

En el **listado 1** se muestra el constructor de la clase en Delphi 5. Aparentemente no existe ningún cambio relevante hasta que fijamos nuestra vista en el interior del procedimiento *AddThread*. Vamos a compararlos y para ello, los situaré, uno al lado del otro:

DELPHI 5

```

procedure AddThread;
begin
  EnterCriticalSection(ThreadLock);
  try
    if ThreadCount = 0 then
      ThreadWindow := AllocateWindow;
    Inc(ThreadCount);
  finally
    LeaveCriticalSection(ThreadLock);
  end;
end;

```

DELPHI 6

```

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

```

Aquí ya puedes observar la primera diferencia realmente importante. La he resaltado en color para que la visualices rápidamente. Delphi 6, como veíamos en nuestros dos anteriores artículos, hacía uso de una lista de punteros para almacenar una estructura de registro cuyo tipo es *TSyncProc*. El detalle de los campos de esta estructura, lo podéis ver con detalle en la página 18 del número 13 de Síntesis. Básicamente, se almacenan un puntero al método a ejecutar por el hilo, y una referencia al mismo hilo (Self).

Veamos donde nos lleva la llamada a la función *AllocateWindow* en Delphi 5. Os anticipo que la variable *ThreadWindow*, almacenará un “Handle” o manejador a la ventana creada (HWND):

```

function AllocateWindow: HWND;
var
  TempClass: TWndClass;
  ClassRegistered: Boolean;
begin
  ThreadWindowClass.hInstance := HInstance;
  ClassRegistered := GetClassInfo(HInstance, ThreadWindowClass.lpszClassName,
    TempClass);
  if not ClassRegistered or (TempClass.lpfWndProc <> @ThreadWndProc) then
    begin
      if ClassRegistered then
        Windows.UnregisterClass(ThreadWindowClass.lpszClassName, HInstance);
        Windows.RegisterClass(ThreadWindowClass);
      end;
      Result := CreateWindow(ThreadWindowClass.lpszClassName, '', 0,
        0, 0, 0, 0, 0, HInstance, nil);
    end;

```

Nos quedamos con la última línea de la función *AllocateWindow*, donde se produce la llamada a la función del API de Windows, tras proceder al registro de la Clase, *CreateWindow()*. El tercer parámetro (*dwStyle*) corresponde a las opciones de estilo de la ventana a crear y recibe como valor 0. Este parámetro se define en la función como *DWORD*, y este número de 32 bits, es el resultado de combinar las constantes de estilo tales como su visibilidad *WS\_VISIBLE*, borde exterior *WS\_BORDER*, etc... Un ejemplo de una combinación cualquiera podría ser *WS\_VISIBLE or WS\_VSCROLL or WS\_BORDER or ...*

Nos vamos a valer pues de esta ventana oculta para sincronizar la ejecución de cada uno de los hilos que concurren; y el modo es similar también a cómo es gestionado en Delphi 6: cada vez que tiene lugar la creación de un nuevo hilo se comprobará que el contador de hilos creados (*ThreadCount*) vale 0, en cuyo caso, debemos crear el objeto que nos ayudará a sincronizar. En Delphi 5, por los motivos ya explicados, hacemos uso de una ventana oculta y será esta ventana la que recibirá el mensaje *CM\_EXEPROC*, mensaje que contiene en su campo *lparam* una referencia al hilo que se debe ejecutar, a través de la que se podrá acceder al método mismo. Estos mensajes serán despachados de forma serializada en la cola que gestiona los mensajes de nuestra ventana oculta. Hay que tener en cuenta que, dado que esta ventana recibe en su creación, en el penúltimo de los parámetros, *HInstance*, es decir, la instancia del módulo que la crea, la ejecución de los métodos a través de *Synchronize()*, tienen lugar dentro del contexto del hilo primario, garantizando así la concurrencia.

Veamos el método *Synchronize()*:

```

procedure TThread.Synchronize(Method: TThreadMethod);
begin
  FSynchronizeException := nil;
  FMethod := Method;
  SendMessage(ThreadWindow, CM_EXECPROC, 0, Longint(Self));
  if Assigned(FSynchronizeException) then raise FSynchronizeException;
end;

```

Ahora, quizás, la pregunta que os estéis haciendo es precisamente qué sucede en el método que gestiona los mensajes recibidos por la ventana. Lo podemos ver:

En el módulo *Classes.pas*, dentro de la sección de implementación, es declarada la variable global *ThreadWindowClass*. Antes, unas líneas más arriba, en la función *AllocateWindow( )*, haremos uso de ella para registrar la clase de la ventana.

```

var
  ThreadWindowClass: TWndClass = (
    style: 0;
    lpfnWndProc: @ThreadWndProc;
    cbClsExtra: 0;
    cbWndExtra: 0;
    hInstance: 0;
    hIcon: 0;
    hCursor: 0;
    hbrBackground: 0;
    lpszMenuName: nil;
    lpszClassName: 'TThreadWindow');

```

He resaltado *@ThreadWndProc*, porque será la función que gestionará los mensajes que reciba la ventana y que básicamente se reducen a la captura de dos mensajes propios:

```

const
  CM_EXECPROC = $8FFF;
  CM_DESTROYWINDOW = $8FFE;

```

Estos dos mensajes serán procesados por la función *ThreadWndProc( )*, lo que normalmente podemos conocer como procedimiento de ventana:

```

function ThreadWndProc(Window: HWND; Message, wParam, lParam: Longint): Longint; stdcall;
begin
  case Message of
    CM_EXECPROC:
      with TThread(lParam) do
        begin
          Result := 0;
          try
            FSynchronizeException := nil;
            FMethod;
          except
            if RaiseList <> nil then
              begin
                FSynchronizeException := PRaiseFrame(RaiseList)^.ExceptObject;
                PRaiseFrame(RaiseList)^.ExceptObject := nil;
              end;
            end;
          end;
    CM_DESTROYWINDOW:
      begin
        EnterCriticalSection(ThreadLock);
        try
          Dec(ThreadCount);
          if ThreadCount = 0 then
            FreeThreadWindow;
        end;
      end;

```

```
        finally
            LeaveCriticalSection(ThreadLock);
        end;
        Result := 0;
    end;
else
    Result := DefWindowProc(Window, Message, wParam, lParam);
end;
end;
```

Destacamos dos cosas: tras la recepción del mensaje *CM\_EXECPROC*, vamos a intentar ejecutar el método de nuestro hilo a través del puntero a método *FMethod*. Y por otro lado, tras recibir el mensaje *CM\_DESTROYWINDOW*, se decrementa el contador de hilos. Este contador, tomaría valor 0 si el mensaje proviniese del único hilo que quedaba en ejecución y que resulta removido, por lo que, dado que ya no quedan hilos que gestionar, podemos destruir nuestra ventana oculta. La llamada a *FreeThreadWindow* encierra la destrucción de esta ventana, como se puede ver:

```
procedure FreeThreadWindow;
begin
    if ThreadWindow <> 0 then
        begin
            DestroyWindow(ThreadWindow);
            ThreadWindow := 0;
        end;
end;
```

Si veíamos que el mensaje *CM\_EXECPROC* es enviado en cada una de las llamadas al método *Synchronize* (), será el procedimiento *RemoveThread*, invocado desde el destructor de la clase *TThread*, el que gestione el envío del mensaje de destrucción de la ventana, *CM\_DESTROYWINDOW*.

```
procedure RemoveThread;
begin
    EnterCriticalSection(ThreadLock);
    try
        if ThreadCount = 1 then
            PostMessage(ThreadWindow, CM_DESTROYWINDOW, 0, 0);
        finally
            LeaveCriticalSection(ThreadLock);
        end;
    end;
end;
```

Podemos resumir... Si se tiene en mente reutilizar código fuente en el que extendamos la funcionalidad de la clase *TThread* ,y que afecte a esta ventana oculta, ya intuimos que no va a funcionar, puesto que esta ventana va a ser sustituida en Delphi 6 por una lista de punteros, global, a la que accedemos mediante una zona crítica. Tendremos que modificar nuestro código y adaptarlo a la nueva implementación, que hemos visto con detalle en los dos artículos anteriores. Y saber eso ya no solo se hace conveniente, sino totalmente necesario, como bien se puede imaginar.

## ¿Que es una ThreadVar...?

Antes de iniciar cualquier explicación al concepto que encierra la cláusula **ThreadVar**, que nos permitirá que una variable global tenga unas características un tanto especiales, nos vamos a proponer un pequeño ejemplo que nos resulte significativo. Veamos que es lo que queremos...

```
TMiHilo = class(TThread)
private
  MiNumero: Integer;
protected
  procedure Execute; override;
  procedure MiMetodo;
public
  constructor Create(Valor: Integer);
end;
```

La nueva clase TMiHilo, ha declarado la variable global *NumeroGlobal* como *ThreadVar*.

```
threadvar
NumeroGlobal: Integer;
```

La idea que se persigue es algo muy sencillo: que el *Caption* de nuestro *Form*, visualice el mensaje “Mi número vale + [el valor de dicha variable]”.

Para ello y a través del constructor del hilo, recibiremos y guardaremos el valor a visualizar,

```
constructor TMiHilo.Create(Valor: Integer);
begin
  MiNumero:= Valor;
  inherited Create(False);
end;
```

depositando finalmente ese valor temporal en la variable *NumeroGlobal*, dentro del procedimiento *Execute*, y asignándola a nuestro *Caption* en el método invocado a través de *Synchronize*( ), como lo requiere el acceso a objetos pertenecientes al hilo primario de nuestra aplicación.

```
procedure TMiHilo.Execute;
begin
  FreeOnTerminate:= True;
  NumeroGlobal:= MiNumero;
  Synchronize(MiMetodo);
end;

procedure TMiHilo.MiMetodo;
begin
  Form1.Caption:= 'Mi Numero vale ' + IntToStr(NumeroGlobal);
end;
```

Tras hacer todo esto, añadimos un botón a nuestro *Form* que nos permita ejecutar la creación del hilo de la siguiente forma:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  NumeroGlobal:= 10;
  Hilo1:= TMiHilo.Create(5);
end;
```

Ejecutamos y sorprendentemente, nuestra ventana nos muestra en su *Caption* “Mi número vale 10”. Así ha primera vista, yo hubiera jurado que se debería haber sobrescrito la variable *NumeroGlobal*, que inicialmente vale 10, y que tras la creación del hilo *Hilo1*, tomará el nuevo valor asignado (5)...

Ummm... ¿Se ve...?

El motivo por el que la variable **ThreadVar** *NumeroGlobal* toma finalmente el valor 10 es porque nunca es sobrescrita, ya que Windows, nos permitirá mediante un mecanismo de almacenamiento local, y eso lo hace Delphi por nosotros, una copia local a cada hilo de la variable, y por lo tanto, y dado que estamos accediendo a

*NumeroGlobal*, desde el contexto del hilo primario, accedemos al valor inicial que es 10, siendo solo modificada por el hilo la copia que obtiene de la variable y no la que pertenece al contexto del hilo primario.

Creo que el ejemplo expuesto muestra claramente la existencia de ambas variables y hace comprensible el concepto. Este ejemplo lo podéis encontrar en las fuentes que acompañan a la Revista en la carpetas *d6\_threadvar* y *d5\_threadvar*, respectivamente para una y otra versión del compilador. El motivo de hacerlo así es por salvar los problemas que se producen en la lectura de los valores del formulario y la existencia en ocasiones de propiedades publicadas, inexistentes en Delphi 5 y que sí existen en Delphi 6, con los consiguientes errores de lectura.

De haber utilizado la cláusula **Var** para declarar la variable global, irremisiblemente hubiera sido sobrescrita por la ejecución del hilo y nuestro *Form* mostraría un *Caption* tal que: “Mi numero es 5”.

De no existir esta cláusula, cualquier variable global accedida por múltiples hilos, nos obligaría a su protección, mediante los mecanismos propios de la sincronización que ya hemos estudiado anteriormente, si deseamos evitar que ésta pueda ser accidentalmente sobrescrita. Intentaremos hacer uso de este tipo de variables en los próximos ejemplos donde vamos a abordar la asignación de prioridad en la ejecución de hilos.

IDLE_PRIORITY_CLASS	Prioridad muy baja. Un proceso con este tipo de prioridad, esperará pacientemente ante procesos con una prioridad mayor.
NORMAL_PRIORITY_CLASS	Prioridad normal. La prioridad asignada por defecto a todos los procesos.
HIGH_PRIORITY_CLASS	Prioridad alta. Muchos de los procesos del S.O. son buenos candidatos a esta asignación. Si la utilizamos debemos ser prudentes en su utilización, ya que los procesos con esta prioridad, harán uso de una buena parte de los ciclos de la CPU.
REALTIME_PRIORITY CLASS	La prioridad más alta posible. No es aconsejable, si no se hace realmente necesaria, ya que podría afectar a la ejecución de procesos del S.O. con menor nivel de prioridad.

Tabla 1. Valores de prioridad de un Proceso.

## Quando nos llevan las prisas...

Las prisas nunca son buenas pero en ocasiones se hacen ¿necesarias?. En un mundo perfecto el desarrollo y ejecución de múltiples tareas posiblemente nunca exigirían que unas tuviesen prioridad sobre las otras, pero gracias a Dios, nuestro mundo es imperfecto, y nuestro deseo de reflejar esa realidad y esa necesidad de que “algo” sea prioritario sobre otro “algo”, también se traslada a la programación. Decíamos al principio de esta serie, en ese capítulo de introducción al tema, cómo la concurrencia de múltiples procesos era posible gracias a compartir, todos ellos, tiempo de la CPU, que “magistralmente” administraba nuestro sistema operativo. Teníamos la apariencia engañosa de que todos esos procesos eran ejecutados simultáneamente, aun cuando no era realmente así. Al asignar una determinada prioridad a nuestros hilos, concedemos mayor o menor porcentaje de uso de nuestra CPU, como vamos a ver.

Pero vamos a empezar por el principio, tal y como hace la mayoría de la documentación de que dispongo. Y el principio, es la misma creación de cualquiera de los procesos que concurren. La sintaxis de la función *CreateProcess()*, de la que hacemos uso entre bastidores es la siguiente:

```

BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPCTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPVOID lpReserved,
    LPVOID lpReserved,
    LPVOID lpReserved)

```

```

LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory,
LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation
);

```

Nos interesa el parámetro *dwCreationFlags*, que puede contener cualquiera de los cuatro siguientes valores, representados en la **tabla 1**:

El API de Windows nos proporciona dos funciones para obtener en tiempo de ejecución, el valor actual de la prioridad asignada a un proceso *GetPriorityClass()*, y para modificarla *SetPriorityClass()*. Pero realmente esto es algo que no nos interesaría a nivel de Threads o Hilos, si no fuera porque la prioridad general de cada hilo vendrá determinada por la conjunción de ambas prioridades: la del proceso en la que el hilo es creado y la del propio hilo, que puede tomar cualquiera de los siete valores indicados en la **tabla 2**.

CONSTANTE	TThreadPriority	Valor
THREAD_PRIORITY_IDLE	tpIdle	-15
THREAD_PRIORITY_LOWEST	tpLowest	-2
THREAD_PRIORITY_BELOW_NORMAL	tpLower	-1
THREAD_PRIORITY_NORMAL	tpNormal	0
THREAD_PRIORITY_ABOVE_NORMAL	tpHigher	1
THREAD_PRIORITY_HIGHEST	tpHighest	2
THREAD_PRIORITY_TIME_CRITICAL	tpTimeCritical	15

Tabla 2. Valores de prioridad asignadas al Hilo.

Todos estos valores vais a poder encontrarlos definidos en el módulo “*Windows.pas*”, ubicado en “*..\Source\Rtl\Win*” del directorio de instalación de Delphi.

Así que, lo que nos interesa es saber como la clase TThread nos permite asignar la prioridad al hilo para así poder hacer uso de esta característica de ser necesario. Algo fácil, sin duda:

```

MiHilo.Priority:= tpHigher;

```

¿Queréis que veamos que es lo que sucede realmente cuando se produce la asignación?:

La clase TThread define el tipo TThreadPriority, como un tipo enumerado donde sus valores representan los siete posibles niveles de prioridad.

```

{$IFDEF MSWINDOWS}
TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest,
tpTimeCritical);
{$ENDIF}

```

Por otro lado se define la siguiente matriz que almacenan los valores de las constantes que Windows asigna a cada prioridad relativa (de hilo):

```

{$IFDEF MSWINDOWS}
const
Priorities: array [TThreadPriority] of Integer =
(THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL,
THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL,
THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL);

```

Y nos serviremos de esta matriz y del conjunto de tipos enumerados, para obtener y fijar los valores a través de las llamadas al API.

Para la lectura de la prioridad del hilo, el método *GetPriority*( ):

```
function TThread.GetPriority: TThreadPriority;
var
  P: Integer;
  I: TThreadPriority;
begin
  P := GetThreadPriority(FHandle);
  CheckThreadError(P <> THREAD_PRIORITY_ERROR_RETURN);
  Result := tpNormal;
  for I := Low(TThreadPriority) to High(TThreadPriority) do
    if Priorities[I] = P then Result := I;
  end;
```

Obtenemos la prioridad del hilo mediante la invocación de *GetThreadPriority*( ), que recibe como parámetro el identificador del hilo, y como se puede recordar éste era almacenado en el campo *FHandle* del objeto. Luego nos basta recorrer el array e ir comparándolo con este valor y devolver aquel índice (*TThreadPriority*) que representa a la constante encontrada.

Y la escritura, la asignación de la nueva prioridad, resulta si cabe más sencillo.

```
procedure TThread.SetPriority(Value: TThreadPriority);
begin
  CheckThreadError(SetThreadPriority(FHandle, Priorities[Value]));
end;
{$SENDIF }
```

Como comentario final a este tema, al menos en su parte teórica, comentan Teixeira y Pacheco en su libro “Guía de Desarrollo de Delphi 5”, que la prioridad general de un hilo es un valor puede ser cualquier valor comprendido entre 1 y 31, resultado de la combinación de la prioridad del proceso y la prioridad relativa del hilo. Las prioridades *tpIdle* y *tpTimeCritical*, según el autor, recibirán un trato especial por parte del sistema operativo ya que la asignación del primero de estos dos valores, con independencia de la prioridad del proceso, establecerá como valor de prioridad general de

hilo un valor de 1. En el otro extremo, el segundo valor, también tiene un tratamiento especial: si el proceso que genera el hilo tiene el nivel de prioridad mas baja, el hilo creado con prioridad *tpTimeCritical*, obtendrá un valor

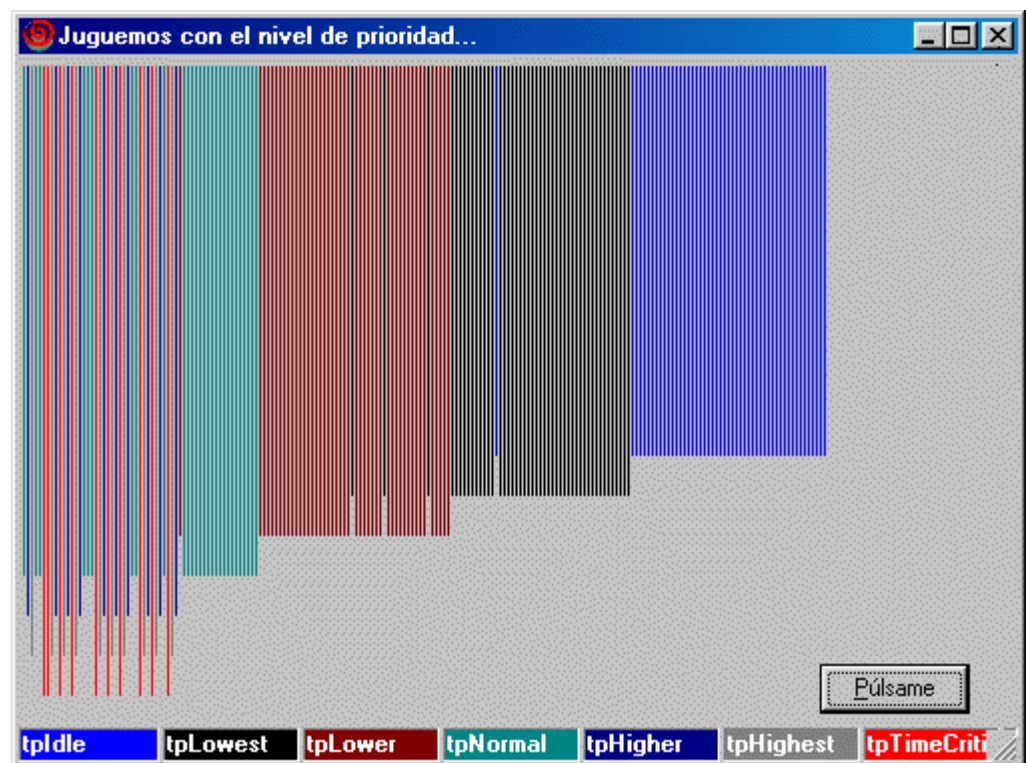


figura 1.- Instante de la ejecución del primer ejemplo sobre prioridades “Proprior.exe”

general de 15. Y si son combinados el nivel *REALTIME\_PRIORITY\_CLASS* del proceso con éste, se asignará el valor máximo de prioridad general que es 31. No he podido comprobar por otros autores el modo en el que son combinados y obtenidos estos valores, pero los reflejo por si les fuera de alguna ayuda al lector.

### ¿No tendrás a mano un ejemplo, majo...?

Pues he preparado dos, para que no se me queje nadie... Ambos, se van a poder localizar en las carpetas *d6\_prioridades2* y *d6\_prioridades*, acompañadas de otras similares, propias para Delphi 5. Acompañan a la revista y se descargan junto a la misma. Del primer ejemplo nos vamos a limitar a comentar tan solo qué se busca observar. Nos puede servir un poco de reflexión, a pesar de la sencillez del código que implementa. Abrid la carpeta *d6\_prioridades*. Ejecutad, tras compilar el ejemplo, y aparecerá una ventana prácticamente vacía, con tan solo un botón que nos invita a pulsarlo.



figura 2.- Ejecución de múltiples hilos con el mismo nivel de prioridad.

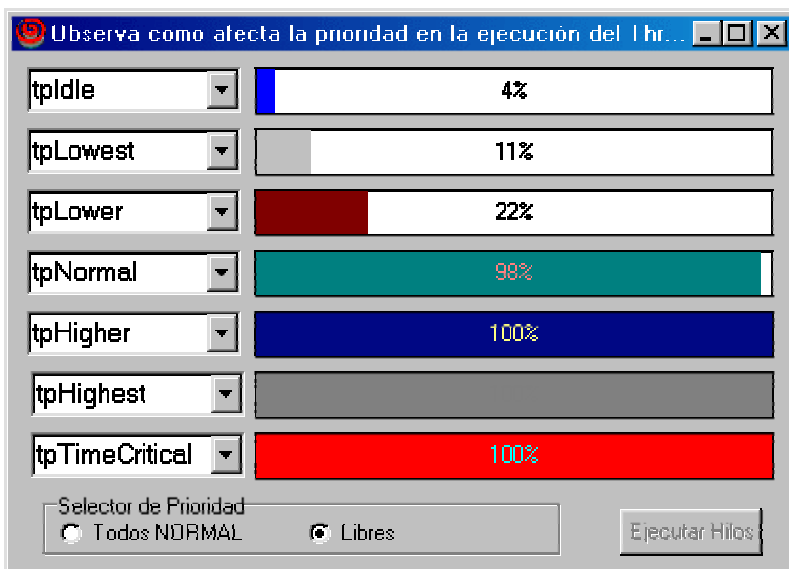


figura 3.- Ejecución de múltiples hilos con progresivo nivel de prioridad.

prioridad más bajos. Al soltar el botón, el diagrama resultante es similar a la **figura 1**, donde se puede apreciar que a medida que baja la prioridad, mayor número de suspensiones ha sufrido: la superficie pintada con los colores

En la parte inferior, he colocado un *TStatusBar*, y cada uno de los paneles representan con colores los siete posibles niveles de prioridad relativa. La idea que se persigue es que, al pulsar el botón, sean ejecutados siete hilos concurriendo con distintos niveles de prioridad relativa. Cada uno de ellos, accederá a nuestro *form* y dibujará sobre él 10 líneas, con el color correspondiente a la asignación establecida en la barra de estado.

Pulemos una sola vez: se debe obtener un diagrama escalonado pero relativamente proporcional. Ahora bien, si mantenemos pulsado el botón una serie indeterminada de segundos, la creación de nuevos grupos de hilos van siendo suspendidos progresivamente, ejecutándose tan solo aquellos que se crean con niveles de prioridad relativamente altos e ignorándose los que poseen niveles de

correspondientes a las prioridades más bajas es mayor. No se me ocurría en el momento en que escribo estas líneas, una forma mejor de expresar esa reflexión: ¿será esa la razón de que un componente como TTimer pierda la exactitud en el disparo del evento *OnTimer*, y que sea poco recomendable para usos que requieran precisión? Dejo ahí esta reflexión que me hago...

El segundo ejemplo resulta totalmente intuitivo y a su vez, repetitivo. Cualquiera que se empeña en explicar de forma visual todos estos conceptos, llega a la conclusión de que es ilustrativo la ejecución de múltiples hilos, simultáneamente, y actuando cada uno de ellos con distinto nivel de prioridad. Y casi todo el mundo acaba por implementar ejemplos similares, valiéndose de barras de progreso o cualquier componente con una función parecida. Las **figuras 2 y 3** nos lo muestran.

Veamos la declaración de la clase TMiHilo:

```
TMiHilo = class(TThread)
private
  FGauge: TGauge;
  Continuar: Boolean;
  ContadorLocal: Integer;
  FId: Integer;
  procedure SetId(const Value: Integer);
protected
  procedure HiloPrincipal; virtual;
  procedure Execute; override;
public
  Property Id: Integer read FId write SetId;
  Property Gauge: TGauge read FGauge;
  Constructor Create(MiGauge: TGauge);
  Destructor Destroy; Override;
end;
```

No puede ser más sencilla. En el constructor, almacenaremos en el campo *FGauge*, una referencia a la barra de progreso asociada al hilo. Asimismo, el destructor nos servirá para romper este vínculo, asignándolo a *nil*.

```
constructor TMiHilo.Create(MiGauge: TGauge);
begin
  FGauge:= MiGauge;
  Continuar:= True;
  inherited Create(True);
end;
```

```
destructor TMiHilo.Destroy;
begin
  FGauge:= Nil;
  inherited Destroy;
end;
```

Nuestro hilo, se limitará tan solo a incrementar sucesivamente en una unidad el valor de la propiedad *Progress*, del componente TGauge.

```
procedure TMiHilo.Execute;
begin
  Continuar:= True;
  FreeOnTerminate:= True;
  repeat
    ContadorLocal:= Contador;
    Synchronize(HiloPrincipal);
    Inc(Contador);
  until Continuar;
end;
```

```
procedure TmiHilo.HiloPrincipal;
begin
  Continuar:= (ContadorLocal >= FGauge.MaxValue) or Terminated;
  if Continuar then Dec(Contador);
  FGauge.Progress:= ContadorLocal;
end;
```

A simple vista, puede parecer un poco extraño como se combinan los valores de *ContadorLocal* y *Contador*, pero si tenemos en cuenta que esta segunda se declara como *ThreadVar*:

```
ThreadVar
  Contador: Integer;
```

La variable contador, dentro del contexto del hilo principal, nos sirve para mantener un contador del número de hilos ejecutándose. El hilo principal se vale del evento *OnTerminate* de cada hilo para verificar según el valor de dicho contador, si se debe activar el botón de lanzamiento y el panel selector. El procedimiento *OnFinish* es asignado a este evento.

```
procedure TfrmPrincipal.OnFinish(Sender: TObject);
begin
  //Es necesario liberar la referencia para que el evento OnChange
  //no genere una excepción en el caso que se intente cambiar la prioridad
  //tras la finalización del hilo
  MisBarras[TmiHilo(Sender).Id].Hilo:= Nil;
  //nos valemos del contador para evaluar el estado del boton
  //y del cuadro selector
  but_Lanzar.Enabled:= Contador = 0;
  rgp_Selector.Enabled:= but_Lanzar.Enabled;
end;
```

Sin embargo, dentro del contexto de cada uno de los hilos, la variable contador representa el número máximo de ejecuciones, siendo destruido al ser alcanzado éste, que coincide con el valor máximo que puede tomar cada barra de progreso o TGauge.

El resto es un poco más anecdótico. Hago uso de un registro en el que almaceno una referencia a cada uno de los siguientes objetos, y que me permiten mediante bucles la manipulación de los mismos, tarea esta mucho más liviana que de no poder hacerlo así.

```
Par = Record
  Gauge: TGauge;
  Combo: TComboBox;
  Hilo : TmiHilo;
end;
```

Ejecutaremos el procedimiento *LanzarHilos* desde la pulsación del botón:

```
procedure TfrmPrincipal.but_LanzarClick(Sender: TObject);
begin
  LanzarHilos;
  but_Lanzar.Enabled:= False;
end;
```

Este método, inicializa el contador de hilos y recorre la matriz, procediendo respectivamente a la creación y lanzamiento de los hilos. En cada nueva ejecución se incrementa el contador, como se puede apreciar.

```
procedure TfrmPrincipal.LanzarHilos;
var
  xIndice: integer;
begin
  Contador:= 0; //inicializamos el contador de hilos
  for xIndice:= 0 to 6 do
    begin
```

```

with MisBarras[xIndice] do
begin
  Gauge.Progress:= 0;           //Inicializamos la barra de progreso
  Hilo:= TMiHilo.Create(Gauge); //Creamos el hilo [estado suspendido]
  Hilo.Id:= xIndice;           //Almacenamos el indice de la matriz en el hilo
  Hilo.OnTerminate:= OnFinish; //Asignamos el evento de finalización
  //Ajustamos la prioridad según el indice del combobox correspondiente
  case Combo.ItemIndex of
    0: Hilo.Priority:= tpIdle;
    1: Hilo.Priority:= tpLowest;
    2: Hilo.Priority:= tpLower;
    3: Hilo.Priority:= tpNormal;
    4: Hilo.Priority:= tpHigher;
    5: Hilo.Priority:= tpHighest;
    6: Hilo.Priority:= tpTimeCritical;
  else
    Hilo.Priority:= tpNormal;
  end;
  Inc(Contador); //incrementamos un contador de hilos
  Hilo.Resume;   //y lanzamos la ejecución del hilo
end;
end;
end;

```

Y al destruirse nuestra ventana, debemos proceder a finalizar la ejecución de cada uno de los hilos, si es que estuvieran en estado de ejecución, y a desasignar el vínculo entre la barra de progreso y el campo del registro, así como hacer lo propio entre cada TComboBox y el último de los campos del mencionado registro.

```

procedure TfrmPrincipal.FormDestroy(Sender: TObject);
var
  xIndice: integer;
begin
  for xIndice:= 0 to 6 do
  begin
    with MisBarras[xIndice] do
    begin
      Gauge:= Nil;
      Combo:= Nil;
      if Hilo <> Nil then Hilo.Terminate;
    end;
  end;
end;

```

Y por último, por ejemplo, podemos responder al evento *OnChange* de nuestros TComboBox, para que al ser elegido otro ítem se modifique también la prioridad del hilo al que hace referencia, durante la ejecución

```

procedure TfrmPrincipal.OnChangeGeneral(Sender: TObject);
var
  prioridad: TThreadPriority;
begin
  if TComboBox(Sender).itemindex <> -1 then
    prioridad:= TThreadPriority(TComboBox(Sender).itemindex)
  else
    prioridad:= tpNormal;
  //nos aseguramos de que el objeto existe
  if MisBarras[TComboBox(Sender).tag].Hilo <> nil then
    if MisBarras[TComboBox(Sender).tag].Hilo.Gauge <> nil then
      MisBarras[TComboBox(Sender).tag].Hilo.Priority:= prioridad;
end;

```

Y poco más... estamos acabando.

## Para ampliar... por si te aburres.

La comunidad de Borland suele ser un buen sitio para husmear, a pesar del problema que nos supone el no encontrar nada en castellano. Muchos de nosotros lo vencemos, en un buen número de casos, apoyándonos en algún que otro diccionario. Sobre este tema que nos ocupa, encontré recientemente la publicación del fallo de un concurso cuyo tema central eran las aplicaciones multihilo. Lo podéis encontrar en <http://community.borland.com/article/0,1410,29786,00.html>

Este artículo de John Kaster, recoge los ganadores del concurso en el año anterior, y desde él podéis acceder a la descarga del código fuente, que sin duda no solo os resultará interesante sino incluso práctico y útil. De los 7 ganadores, 5 de ellos lo hacen con el compilador de Delphi. Particularmente, y concretando en estos cinco, pueden resultar muy instructivos el “Virtual Thread Manager” de Guinther de Bitencourt Pauli, y el “Building\_pyramids...” de Denis Portilho, para comprender mejor el funcionamiento de los hilos. Y respecto a ser útil, sin duda el “Databasecopy...” de Wayne Sepega, donde se puede exportar una tabla a CDS, sea en binario o en XML. Muy interesante.

Y para el número próximo, el último de esta serie, intentaremos poner en práctica todo lo que hemos aprendido, y lo haremos junto a nuestro compañero Jose Manuel Navarro, con el que vamos a compartir un ejemplo de uso de hilos con mayor complejidad. Gracias, Jose.

Me resta despedirme, con el deseo de que os haya gustado y de que sigáis compartiendo estos ratos con nosotros. Y como siempre os comento: no es mi artículo, sino el vuestro... así que podéis hacer cuantas sugerencias os parezcan oportunas, matizar y completar con vuestros comentarios su contenido. Incluso enviar algún ejemplo que sea representativo y que os apetezca compartirlo.