

TThread VI: Un buscador de Archivos (y II)

Salvador Jover – s.jover@wanadoo.es

Vamos a la segunda y última entrega del buscador de Archivos. Aprender de los errores, que dicen los sabios, y rectificar... dos nuevas versiones del buscador de ficheros, ya funcionales.

A menudo escribimos todas estas páginas..., y pensamos que pueden ayudar a alguien mas que a nosotros mismos. Es algo que nos hace sentir bien..., podéis creerme. Suele ser una buena parte de las veces, cuando acabamos convencidos de que lo escrito está a la altura de ese listón imaginario, un listón que alzamos antes de sentarnos frente al equipo, y que nos sirve de medida, para saber si los objetivos que nos marcamos se han cumplido. Obviamente, ni se me ocurre pensar que fuera así. Nuestro artículo anterior es un punto de partida y posiblemente, éste también lo sea hacia otros objetivos. Raro es que no haya recibido de cualquiera de vosotros recriminación alguna: ¡majo!, ¡te pasaste! -podréis decirme, y no sin razón-. Sobretudo porque la lectura de esa primera parte del buscador de archivos, cuando se trasladaba a las fuentes y se revisaban con detenimiento, a uno le quedaba esa inseguridad de que algo andaba mal, como quien queda al borde de un precipicio en un puente de madera carcomido... Ejecutábamos la aplicación de ejemplo y quedábamos a la espera del mensaje de error si era cerrada la ventana. ¿Saldrá ahora...?

Pero era parte del desarrollo del artículo hacerlo así, con la idea de quedaran al descubierto algunos de los problemas que traían la incorporación de nuevos hilos de ejecución adicionales, en el uso de la clase TThread. Son la clase de cosas, los aspectos sucios que no se suelen ver cuando se ojea código fuente, al estilo de los galanes de Hollywood, que ni siquiera se despeinan tras una lucha encarnizada. ¡Suerte la suya!

En nuestro anterior artículo, nos quedamos en ese segundo borrador y apuntamos una serie de ideas al finalizar el mismo, sobre aquellos aspectos que se podían mejorar en una versión posterior. Así pues, ya os adelanto que las fuentes de éste, van a incluir un tercer borrador, dentro de la carpeta *3er_Borrador*. Y se incluyen también dos versiones “definitivas” distintas, si es que hay algo que se pueda dar como definitivo. Las encontraréis en *bficheros_jm* y *bficheros_sj*. Para facilitar que podáis probar los ejemplos, se ha buscado que todo el código fuente fuera compatible tanto con Delphi 5 como con Delphi 6, de forma que no presentara problemas de lectura del *form* al abrir el entorno, y no nos obligara a presentar dos versiones distintas, como hicimos en anteriores artículos. Por otro lado, no he añadido la rutina *Register*, que como sabéis es el procedimiento mediante el que añadimos nuestros componentes a la paleta de Componentes de Delphi. No lo he hecho en ninguno de los ejemplos que se acompañan. Lo haremos posteriormente, ya fuera de este artículo. La idea es, si no se pone algún inconveniente desde el grupo Albor, publicar estas dos versiones definitivas en la sección de componentes de su sede, y que podáis, si queréis, ir añadiendo mejoras a las mismas, depurando aquellos errores que podamos apreciar y dotándolas de funcionalidad real. Un componente que te muestre tan solo los archivos, ¿servirá de mucho? Podría hacer muchas cosas y mostrarnos mucha más información que tan solo el nombre del archivo encontrado. Creo que sería un buen estímulo tanto para la gente que participamos de este proyecto, Síntesis, como para los compañeros de Albor, que vuestra participación se convierta en algo vivo y real, que no se sepa realmente donde esta la frontera de quien lee el artículo y de quien lo escribe.

Como comentario, y antes de seguir, me gustaría que el lector vea estos dos artículos, el anterior y éste, como una oportunidad para recrear y observar sobre el terreno algunos pensamientos que se han ido abordando. Ambos se han escrito con la idea de que se recreen los ejemplos y el lector pueda poner puntos de parada y seguir -paso a paso- algunos puntos conflictivos que mencionamos o remarcamos como “terreno peligroso”. Si el lector analiza, hacemos una transición, desde un primer borrador, que le da la oportunidad de ver sobre el terreno, como se comporta una aplicación que crea un número indeterminado de hilos y el impacto real que supone al sistema. Hasta llegar a las versiones definitivas, donde la creación de múltiples

hilos se reduce a la demanda de búsquedas por directorios del usuario, claramente restringida. Esa es una de las ideas que quedan en el aire: darle al lector la oportunidad de sacar sus propias conclusiones.

Es una buena hora para iniciarlo... luce el sol en lo alto, la gente pasea por la calle y tenemos todo el tiempo del mundo...

Tercer borrador del componente: urge solucionar problemas...

Las cosas no siempre salen como uno quisiera. Si se analiza seriamente nuestro segundo intento, el presentado en el artículo anterior, a poco que uno sea realista, acaba por reconocer que algo está fallando. Jose Manuel, en uno de sus correos me decía con cierto pudor: - ¡hombre! -exclamaba- parece que andes matando las moscas a cañonazos... y dejaba un icono sonriente haciendo referencia al uso del gancho. Ciertamente, proseguía, coincido contigo en que algo anda mal en la destrucción del componente, pero el uso del gancho me parece excesivo.

Yo también lo pensaba. Lo reconozco. Vale... la introducción del gancho era propia de un programador que se encontraba agobiado por el tiempo, consciente de que estaba aportando un elemento extraño, pero que en último extremo, necesitaba de esa idea que aportara un poco de luz, al motivo de por qué se seguían generando excepciones si era cerrada la ventana principal. El buscador se comportaba casi correctamente. Si no era interrumpida la búsqueda por el cierre de la ventana, era capaz de hacerlo sin generar excepción alguna, por lo que aparentemente se llegaba a la conclusión, tal y como razonábamos entonces, que éste era destruido antes de que algunos de sus hilos hubiesen concluido y que era precisamente el acceso a métodos del mismo, el que provocaba las excepciones. La invocación de un método de una instancia que ya no existía era razón más que suficiente. Esto lo comprobamos en numerosas ocasiones poniendo puntos de parada y visualizando las variables y los campos implicados.

Además, el gancho debía ser eliminado por otra razón de mucho más peso y que también introducíamos vagamente. Recordemos el código de la función gancho:

```
function WHCALLWNDPROC(nCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
var
  Estructura: ^TCWPStruct;
begin
  if nCode > -1 then begin
    if wParam = 0 then begin
      Estructura := Pointer(lParam);
      if Estructura.message = WM_CLOSE then varBuscador.DoCancel;
    end;
    Result := 0;
  end
  else Result := CallNextHookEx(WinHook, nCode, wParam, lParam);
end;
```

Nuestro razonamiento partía de poder capturar el mensaje WM_CLOSE con la idea de adelantarnos al cierre de la ventana y proceder a cancelar los hilos en ejecución. (Os adelanto que en las dos versiones finales del componente no existe ni gancho ni nada por el estilo, pero dejemos eso para más tarde). El problema grave que estaba apareciendo tenía que ver con el diseño del componente... Su diseño nos estaba conduciendo a una situación complicada, ya que era necesaria una variable global para acceder al ámbito de la función gancho (*varBuscador*) y a la postre, se impedía la existencia de más de un buscador. Si hubiéramos incluido más de un buscador en nuestro *Form* tan solo el segundo de ellos hubiera podido ejecutar la salida prematura, puesto que cada buscador, en su creación, sobrescribía dicha variable.

```
constructor TBuscador.Create(AOwner: TComponent);
var
  ListColumn: TListColumn;
```

```
begin
  inherited Create(AOwner);
  varBuscador:= Self;
  ...
```

Tenemos pues el primer error grave de diseño, pero hay mas... existe un segundo error de diseño que introduje en el código fuente y que debemos reparar:

¿Qué pasa con la lista de nodos? ¿Recordáis?. La lista de nodos era una variable de tipo TList, global al módulo, gestionada desde la instancia del componente TBuscador. Cuando un THiloBusqueda era creado tenía dos misiones bien distintas. La primera de ellas era encontrar alguna coincidencia en el directorio actual. La segunda añadir a la lista de nodos todos los directorios que contenía, susceptibles de ser buscados por nuevos THiloBusqueda creados tras la muerte de este. Un THiloBusqueda, extraería de la lista de nodos la nueva ruta y procedería a crearse de nuevo para la nueva ruta a buscar. ¿Os parece correcto esto...? Esto suponía ni más ni menos, que cada THiloBusqueda debería incluir un mecanismo para discriminar posteriormente si el nodo le pertenecía, pues podía, en ese supuesto no pertenecer a su buscador... ¡Demasiado complicado para una cosa tan sencilla!. ¡Queda a las claras que no podía ser una variable global!, pero ¿Donde la íbamos a situar?

Veamos como se resuelve esto en el tercer borrador de nuestro componente. Es un buen momento para que tengáis a mano el código fuente de esa versión. Vamos a comentar tan solo los cambios que hemos introducido, puesto que gran parte del código es el mismo.

Respecto al gancho... desapareció. Ahora hemos probado capturando el mensaje WM_DESTROY, que nos comunica la destrucción de TBuscador, y lo hacemos mediante el procedimiento de ventana:

```
procedure TBuscadorS1.SubClassWndProc(var Message: TMessage);
var
  indic_estado: TEstado;
begin
  if Message.Msg = WM_DESTROY then begin
    indic_estado:= fEstado;
    fEstado:= esDestruyendo;
    if indic_estado <> esDestruyendo then FinalizaThreads;
  end;
  WndProc(Message);
end;
```

Subclasificar el procedimiento de ventana lo podemos hacer en el mismo momento de creación del componente, en su constructor, tal que así:

```
constructor TBuscadorS1.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  WindowProc := SubClassWndProc;
  ...
```

Es decir, nos limitamos a escuchar los mensajes que recibe TBuscador y cuando detectamos que va a ser destruido, invocamos *FinalizaThreads*, que como su nombre indica, tiene la misión de ordenar la destrucción de los hilos creados. Es un forma de intentar asegurarnos que el buscador existe antes de que sean destruidos los hilos. Tenemos que tener en cuenta la existencia de mensajes de comunicación entre los hilos y el buscador, y que estos se hacen efectivos a través del handle del segundo.

```
procedure TBuscadorS1.FinalizaThreads;
var
  i: Integer;
begin
  for i:= 0 to fListaThreads.Count-1 do begin
    TThread(fListaThreads[i]).Terminate;
```

```

        TThread(fListaThreads[i]).WaitFor;
    end;
    fListaThreads.Clear;
    if Estado = esDestruyendo then begin
        fListaThreads.Free;
        fListaThreads:= nil;
    end;
end;

```

El procedimiento *FinalizaThreads*, recorrerá la lista de hilos y ordenará a cada uno de ellos que termine, forzándolo mediante el procedimiento *WaitFor*, que no retornará en tanto no haya finalizado dicho hilo. Luego, nos resta tan solo limpiar la lista y liberar la memoria reservada para la misma dinámicamente en el constructor del hilo, en el caso de que efectivamente esté siendo destruido el componente, dado que este método es invocado también al cancelar la búsqueda.

Respecto a la lista de nodos, introducimos un nuevo comportamiento en el modo en que se desarrolla el algoritmo de búsqueda. A partir de ahora ya no va a ser creado una instancia de THiloBusqueda para cada una de las rutas, sino que tan solo se va a crear un hilo para cada búsqueda inicial, y éste, será el mismo durante toda la ejecución de la búsqueda. Supongamos que TBuscador tiene en su lista de rutas iniciales, los directorios “c:\archivos\” y “d:\archivos\proprios\”. En su método de ejecución *Execute*, creará dos instancias de THiloBusqueda para cada una de estas rutas. La primera de estas instancias, por seguir el ejemplo, buscará en dicha carpeta, “c:\archivos\”, y en las interiores, y toda el desarrollo del algoritmo se producirá en el método *Execute* de TThread. No será necesario crear una nueva instancia para proseguir la búsqueda en una carpeta interior. Es decir, que introduciremos un bucle *While* que se repetirá mientras se den determinadas circunstancias.

```

procedure TBusquedaS1.Execute;
var
    Seguir: Boolean;
begin
    Seguir:= True;
    while (Not Terminated) and ((not(csDestroying in fBuscador.ComponentState))) and
    (fBuscador.Estado = esEjecucion) and Seguir do begin
        Buscar;
        Seguir:= PreparaNuevaBusqueda(fNuevaBusqueda);
        // para crear un nuevo objeto TBusquedaS1 que siga buscando y encontrando nuevos
        nodos
        if Seguir then begin
            with fNuevaBusqueda^ do FijaNuevaBusqueda(NArgumentos);
            // Le entregamos un puntero a su nodo para que al finalizar pueda destruirlo
            // destruimos la memoria reservada dinamicamente en el método execute
            Dispose(fNuevaBusqueda);
            fNuevaBusqueda:= nil;
        end;
    end;
end;

```

Antes de que veamos la implementación de este método, se hace necesaria una reflexión que a primera vista me parece importante. Posiblemente, uno de los errores implícitos en el diseño del componente es la forma misma en la que he concebido su desarrollo. Veamos: El diseño inicial parte de un desarrollo que en su momento llamamos “*1er borrador multifind*” y que no estaba pensado para convertirse en un componente. Jose Manuel nos lo entregó con la idea de escenificar el uso de hilos en un ejemplo relativamente sencillo. Al programador, y para andar por casa, le hubiera podido bastar implementar el evento *OnClose* de la ventana principal para destruir el buscador y evitar el problema que ya se presentaba en la destrucción del *form*... Nosotros buscábamos llegar un poco más allá, al convertir la clase TBuscador en un descendiente de TWinControl, e incorporar internamente algunos de los objetos que iban a ser utilizados en la ventana principal. Era un momento en el que debíamos haber reflexionado sobre si

realmente iba a ser necesario hacer estas incorporaciones. Quiero decir: podemos preguntarnos si nuestro TBuscador necesita de estos controles o bien puede prescindir de ellos y hacer uso de otras alternativas mejores. Estamos ligando el desarrollo del algoritmo, a la aparición de una instancia del componente TListView en la que depositar nuestros resultados, mientras que podemos hacerlo haciendo uso de una simple lista de cadenas y facilitando mediante un evento el acceso a cada uno de estos resultados, por cualquier componente que pueda manejarlos, sea un TMemo, un TListView, un TLabel o un cuadro de edición TEdit.

Con la barra de progreso, que nos indica la evolución de la búsqueda y el estado actual, pasa algo parecido. Puede también ser un objeto prescindible y nos valdremos de los eventos *OnMaxChange* y *OnPosChange*, indicadores del número de carpetas exploradas y de la posición actual sobre ese número. Cualquier componente que maneje y visualice el tipo Integer nos valdría para reflejar este porcentaje de exploración: una etiqueta (*pos/max*), una barra de progreso, etc.

Y por otro lado, la inclusión de los tres elementos: árbol, lista de resultados y barra de progreso, añadía una complejidad adicional a la depuración de los problemas que iban apareciendo, ya que aumentaban los accesos al hilo principal.

Se hacía necesario tomar una determinación, y ésta, iba en el sentido de prescindir en el tercer borrador de dichos componentes, de forma que nos permitiera localizar los errores y las excepciones que se generaban. Es otro de los cambios que vamos a encontrar. Hemos ignorado la existencia del árbol de resultados, y de momento y hasta la última versión del componente, no aparecerá éste.

Ahora podemos proseguir comentando la implementación del método *Execute*:

En el interior del bucle *while* lo primero que nos encontramos es una llamada a *Buscar*. Aunque la implementación es similar, por no decir igual, que la sufrida en el borrador anterior, intentamos refinar el procedimiento original para ganar en claridad, desmenuzando la implementación de *Execute* en varios procedimientos como *Buscar/Obtenercarpetas/PreparaNuevaBusqueda*, etc... donde, básicamente, vamos a verificar tanto las nuevas subcarpetas que deben ser añadidas a la lista de nodos, como si existe algún archivo en la carpeta actual que coincida con el patrón Token, e iremos devolviendo cada una de esas coincidencias al buscador asociado al hilo, mediante un mensaje *RE_FINDITEM*, y entregando en el parámetro *w_param* del mensaje un puntero a la cadena. En cualquier caso, antes de salir del bucle, debemos fijar la nueva ruta y lo hacemos tras evaluar la variable booleana *Seguir*, e invocando el procedimiento *FijaNuevaBusqueda()*, cuya misión principal es la de reemplazar la búsqueda en la carpeta actual, por la elegida anteriormente al extraer uno de los nodos de la lista. De esa forma, se vuelve a configurar un nuevo ciclo de búsqueda en una nueva carpeta o ruta.

Quedan por comentar algunos detalles. El procedimiento *OnEnd*, ligado a *OnTerminate* de TThread, siempre se va a ejecutar cuando haya finalizado el algoritmo de búsqueda, gracias a los nuevos cambios, y antes de ser destruido cada hilo, por lo que, se presenta como el lugar ideal para ir comunicando al buscador la finalización de los mismos. De esa forma, el buscador puede saber dos cosas: si el algoritmo de búsqueda finaliza correctamente, en cuyo caso deberá hacer las acciones oportunas a tal efecto, y por otro lado, la finalización de todos los hilos.

Veamos su implementación:

```
procedure TBusquedaS1.OnEnd(Sender: TObject);
begin
  fBuscador.Perform(RE_DEL, 0, 0);
  if (not (csDestroying in fBuscador.ComponentState)) then
    SendMessage(fBuscador.handle, RE_FIN, 0, 0);
  fBuscador.Perform(HB_END, 0, 0);
end;
```

El mensaje *RE_DEL*, decrementará en el buscador el contador de hilos, *fContador*, cuyo valor, tras ser destruidos todos los hilos creados debe de ser cero. El último de los mensajes, *HB_END*, evalúa este valor y de ser cero, es decir, si han sido destruidos todos los hilos, permitirá al buscador enviarse a si mismo un mensaje de finalización.

¿Existe un mundo ideal...?

En este punto debería decir que el componente TBuscador funcionaba correctamente, asombrando a conocidos y vecinos del inmueble... ¿soy honrado si os digo que no era del todo así?. Tras hacer un pequeño ejemplo que introducía el componente, y forzando la ejecución a un numero poderoso de búsquedas, el componente TBuscador no generaba error alguno... en Delphi 5. Sin embargo en Delphi 6, ¡tendrá narices la cosa!, ni siquiera era capaz de finalizar. Se quedaba alelado, embobado con todos y cada uno de los hilos, y no acertaba a destruirlos. En definitiva, seguía sin ir, porque a fin de cuentas, no me veía comentándolo en el artículo: - por favor, utilizadlo solo en Delphi 5!. Todo apuntaba a que el problema estaba en aquel aspecto que abordamos anteriormente en la serie, y en donde hicimos referencia a la diferente implementación de la clase TThread en Delphi 5 y Delphi 6.

Para estos casos, lo mejor es recurrir al correo y en mi caso no dude en cruzar varios correos con Jose Manuel, en donde le comentaba todos los problemas que iba encontrando. Cambiamos impresiones e intentamos razonar el motivo, lo cual nos llevó a las dos versiones definitivas: la de Jose Manuel, partiendo de cero e incorporando algunos detalles muy interesantes, como el uso de una clase iteradora, y la mía, continuación del tercer borrador, y en donde finalmente parecen resolverse todos estos problemas. Ambas tienen un detalle en común que me parece muy relevante, y en donde puede estar la clave de gran parte de los problemas que pudimos observar: el uso de *Synchronize*... olvidado en todas las versiones anteriores.

¿Es casualidad que pueda prescindir en la versión definitiva de subclasificación alguna? Ahora ya no es necesario gancho alguno ni nada que pueda tener, ni por asomo, parecido. Ni siquiera es necesario forzar los hilos a finalizar. ¿casualidad? Estas dos versiones definitivas las tenéis en las carpetas *bficheros_sj* y *bficheros_jm*, que acompañan las fuentes.

¡Tanto hablar en los artículos anteriores del uso de este método, insistiendo y recalcando su uso, y si os fijáis, todos los borradores hacían uso de zonas críticas para sincronizar el acceso al hilo primario desde las instancias THiloBusqueda creadas. Realmente no nos hacía falta para nada dichas zonas críticas. ¡Es mas, y esto lo digo con toda la humildad de quien solamente es un aprendiz en todo esto, parecen ser las candidatas mas apropiadas sobre las que recaer el peso de la culpabilidad: la combinación del uso de zonas críticas juntamente con la creada por el método *Synchronize()* y que se hace efectiva por ejemplo cuando es destruido el hilo, en el evento *OnTerminate* de TThread. La forma en que se fuerza a finalizar a la clase TThread, priorizando la ejecución de los métodos sincronizados, quizá pueda ser un motivo de que alguno de los hilos quedaran atrapados en alguna de las zonas críticas de *Execute*, en donde no hacíamos uso de *Synchronize()*: puedo haber hecho ¿100 pruebas?, ¿200 ejecuciones?, y ni una sola me ha generado la esperada excepción, tanto en Delphi 5 como en Delphi 6, y esto es aplicable tanto al ejemplo de Jose Manuel como al mío. Es decir que una de las conclusiones que parecen desprenderse de todo esto, es que debemos hacer uso del método *Synchronize* y dejarnos de historias o gaitas... se diseñó expresamente para su uso en la clase TThread...

El único problema que se me ocurre es que la declaración del parámetro que recibe el método *Synchronize*, y que es del tipo TThreadMethod

```
TThreadMethod = procedure of object;
```

Un puntero a método, un procedimiento sin parámetros. Esto puede obligar a la declaración de variables que nos permitan la sustitución de ese parámetro que se hace necesario.

```

procedure TsjBusqueda.EncuentraItem;
begin
    if (not (csDestroying in fBuscador.ComponentState)) and
        (fBuscador.Estado <> esInactivo) and
        (fBuscador.Estado <> esCancelando) then
        fBuscador.Perform(RE_FINDITEM, Integer(fItem), 0);
end;

```

Un ejemplo de esto lo podéis ver en el código anterior en la variable *fItem*, improvisada tan solo para poder trasladar un valor que estaba declarado originalmente en una variable local al procedimiento (variable *ultimo* en el procedimiento *Buscar* del borrador tercero). En ocasiones parece que se hace necesario hacer uso de parámetros desde el interior del método que envuelve Synchronize y esto parece una pequeña limitación o un inconveniente menor.

La versión definitiva de Jose Manuel.

En este apartado hubiera agradecido enormemente que Jose Manuel nos comentara su código. No es broma. En ocasiones parece tener mayor dificultad entender la lógica que se ha seguido en el diseño de un algoritmo, que hacerlo propiamente. En mi pueblo dicen algo así como “el que la lleva la entiende...” En este caso pasa algo parecido... Busco la lista de nodos en su código (os recuerdo que esta contenido en la carpeta *bficheros_jm*), y no veo lista alguna. Prescinde de ella. Si analizamos con un mayor detenimiento los primeros movimientos, las primeras rutinas de código del método *Execute* de *TJMHiloBusqueda*, empezaremos a entender que responde a un planteamiento puramente recursivo.

Para entender este planteamiento uno tiene que apartar de su vista muchas líneas que resultan accesorias para su comprensión, y fijarse en los tres o cuatro puntos claves que originan la recursividad y que garantizan que se explore el árbol de directorios totalmente. Intentaré remarcar cuales me parecen esos puntos claves y los comentaremos.

Vamos a suponer que deseamos iniciar una búsqueda cualquiera. En ocasiones resultaría interesante crear tres o cuatro carpetas y un par de archivos en el interior de ellas para simular ésta, y hacer un seguimiento desde Delphi, paso por paso, en la exploración de este pequeño árbol. Yo lo he hecho así. Puse un punto de parada justo en la línea que invoca *Execute*, y voy avanzando paso a paso mediante la pulsación de F7, siguiendo en un la ventana de código el valor de algunas de las variables. Supongamos que lo hacemos así:

```

procedure TJMBuscador.Execute;
begin
    fCountRes:= 0;
    if FRutas.count = 0 then
        raise ESinRutas.Create('No hay rutas de búsqueda configuradas. ');

    if FEstado in [ebPausado, ebBuscando] then
        raise EBuscando.Create('La búsqueda ya está activa. ');

    FEstado := ebPausado;
    FResultado.Clear;

    CrearBusquedas;

    Pausado := false;
end;

```

fCountRes representa al total de coincidencias encontradas por el buscador. Tras inicializar este valor, y comprobar que existen rutas asignadas y que el componente no se haya ya en estado de búsqueda o pausado,

inicializa también la lista de resultados y procede a crear cada uno de los hilos necesarios en *CrearBusquedas*. Hecho esto, puede activar la ejecución del buscador. Un detalle que puede resultar de interés para los compañeros que se inician, es observar como la misma propiedad, nos puede ayudar a desencadenar acciones a través de su escritura.

```
FEstado := ebPausado;
...
Pausado := false;
```

Juega Jose Manuel modificando directamente el valor de la variable *fEstado*, que almacena el estado real del buscador, mientras que en un momento posterior, líneas mas abajo, lo hace invocando a la propiedad *Pausado*, que no solo incidirá sobre la misma variable, sino que, como efecto colateral y tras varias rutinas de código, invocara finalmente al método *Resume* de cada uno de los hilos.

Nos podemos adelantar al momento en que se inicia la ejecución de uno de los hilos creados en *CrearBusquedas*, y lanzados tras la asignación de *Pausado* a *false*. El método *Execute* del hilo consta básicamente de una sola línea de código:

```
ReturnValue := BuscarEnCarpeta(FRuta);
```

Esto nos lleva a comentar el primer punto clave del desarrollo del algoritmo. La invocación de *BuscarEnCarpeta()* para inicializar la búsqueda en un nuevo directorio. En este punto se inicia una nueva búsqueda, y como ya os debéis imaginar, será esta misma rutina la que llamada posteriormente y desde otro tramo de código genere la recursividad.

```
function TJMHiloBusqueda.BuscarEnCarpeta(carpeta: string): integer;
var
  ret: integer;
begin
  result := 0;

  //
  // primera vuelta para buscar los archivos en esta carpeta
  //
  ret := BuscarArchivos(PChar(carpeta));
  if ret = -1 then
  begin
    result := ret;
    exit;
  end
  else
    Inc(result, ret);

  //
  // segunda vuelta para buscar las subcarpetas de esta carpeta
  //
  if FSubcarpetas and (not Terminated) then
  begin
    ret := BuscarSubcarpetas(carpeta);
    if ret = -1 then
    begin
      result := ret;
      exit;
    end
    else
      Inc(result, ret);
  end;
end;
```

Podemos subdividir la implementación de este procedimiento en dos fases diferenciadas, de la misma forma que ya hemos hecho anteriormente: La fase de búsqueda de coincidencias en la carpeta actual y una

segunda fase, que Jose Manuel denomina “segunda vuelta...” en esos comentarios de código y cuyo punto central es la invocación de *BuscarSubcarpetas*. Lógicamente, solo se entrará en esta fase si *fSubcarpetas* tiene valor verdadero, si queremos explorar las subcarpetas y si el hilo no ha sido finalizado prematuramente (*not Terminate*).

Veamos que pasa en el interior del método *BuscarSubcarpetas*. Prescindimos de aquellos trozos de código que resultan mas accesorios. Remarco en otro color la llamada a *BuscarEnCarpeta*:

```
function TJMHiloBusqueda.BuscarSubcarpetas(const carpeta: string): integer;
var
  FindData:      WIN32_FIND_DATA;
  SearchHandle: THandle;
  ret:           integer;
  mascara:      string;
  dir:          string;
begin
  result := 0;

  mascara := '\' + ExtractFileName(carpeta);
  dir     := ExtractFilePath(carpeta);
  dir     := IncludeTrailingBackSlash(dir) + '*.*';

  SearchHandle := FindFirstFile(PChar(dir), FindData);
  if SearchHandle <> INVALID_HANDLE_VALUE then
  begin
    // Se itera en la carpeta actual
    repeat

      // si es carpeta, hay que llamar recursivamente
      if (FindData.dwFileAttributes and FILE_ATTRIBUTE_DIRECTORY <> 0) and
        (FindData.cFileName[0] <> '.') then
      begin
        dir := ExtractFilePath(carpeta);
        dir := IncludeTrailingBackSlash(dir) + FindData.cFileName + mascara;

        ret := BuscarEnCarpeta(dir);
        if ret = -1 then
          result := -1
        else
          Inc(result, ret);
      end;

    until (not FindNextFile(SearchHandle, FindData)) or Terminated or (result = -1);

    ...
    ...
  end;
end;
```

Nos quedamos con los dos puntos claves de la implementación. El primero es la obtención de la nueva ruta y que se representa en el parámetro *dir*, remarcado en color naranja. El segundo punto clave es la llamada a *BuscarEnCarpeta* una vez que se ha modificado anteriormente la variable *dir* con los valores correctos. El bucle **repeat ... until**, que encierra este código, garantiza que la exploración se va hacer para cada uno de los directorios que componen la carpeta actual. Si este ciclo, tal y como lo hemos contado, lo trasladamos al interior de cada uno de los directorios encontrados, garantizamos que la búsqueda se va a mantener mientras quede alguna carpeta por explorar, recorriendo en profundidad todo el árbol de directorios.

Mas comentarios o rarezas de Jose Manuel...

Como podéis observar, intento no comentar cada punto de código. Me parece más importante, pararme en aquellas rutinas que pueden llamar la atención del lector. Es el caso, de algunas clases que introduce Jose Manuel en el modulo *jmHiloBusqueda*. Estoy hablando de las clases TListaHilos y TIteradorHilos. En el **listado 1** podéis ver la definición de ambas clases.

```
TIteradorHilos = class;

//
// Clase auxiliar que define una lista de hilos
//
TListaHilos = class(TList)
private
    function GetHilo(i: integer): TJMHiloBusqueda;
public
    function CreateIterator: TIteradorHilos;
    procedure ReleaseIterator(var it: TIteradorHilos);
    property Items[i: integer]: TJMHiloBusqueda read GetHilo; default;
end;

//
// Un iterador para recorrer la lista de hilos
//
TIteradorHilos = class(TObject)
private
    FListaHilos: TListaHilos;
    FIndex: integer;
    function GetCurrent: TJMHiloBusqueda;
    function GetFirst: TJMHiloBusqueda;
    function GetLast: TJMHiloBusqueda;
    function GetNext: TJMHiloBusqueda;
    function GetPrevious: TJMHiloBusqueda;
public
    constructor Create(lista: TListaHilos);
    property Current: TJMHiloBusqueda read GetCurrent;
    property First: TJMHiloBusqueda read GetFirst;
    property Last: TJMHiloBusqueda read GetLast;
    property Next: TJMHiloBusqueda read GetNext;
    property Previous: TJMHiloBusqueda read GetPrevious;
end;
```

Listado 1 : J. Manuel diseña dos clases auxiliares para el manejo de la lista de hilos

Para entender como se relacionan ambas clases, hay que volver al constructor de TJMBuscador, donde se crea y se guarda una instancia de la clase TListaHilos, *fHilos*, descendiente de TList, y cuya misión es almacenar un puntero a cada uno de los hilos creados. En condiciones normales, nuestro buscador podría manejar directamente la lista de hilos. Tiene los elementos necesarios para hacerlo: la clase TList devuelve en *count* el numero de objetos añadidos a la lista, y podemos mediante un bucle obtener una referencia a los objetos a los que apunta sus items. Jose Manuel ha querido dotar a la clase TListaHilos, de los métodos necesarios para manipular y recorrer la lista de hilos, obteniendo referencias a cada uno de los hilos creados. Son, lo que habitualmente se llaman iteradores.

Un ejemplo de su uso lo podéis encontrar en el interior del método *Cancel*, del buscador. Vamos a analizar esas rutinas. En ellas, se procede a recorrer la lista de hilos para cancelar los que están ejecutándose.

```

...
it := FHilos.CreateIterator();
FEstado := ebCancelando;
try
  // cancelar los hilos y esperar a que cada uno de ellos se haya cancelado
while it.Next <> nil do
  begin
    hilo := it.Current;
    hilo.Terminate();
    hilo.Resume();
  end;

finally
  FEstado := ebInactivo;
  FHilos.ReleaseIterator(it);
end;
...

```

Es la lista de hilos la que, cada vez que necesita recorrerse a si misma, crea una instancia de la clase TIteradorHilos, destruyéndola una vez cumple sus objetivos. Jose Manuel encierra esta destrucción en un bloque de protección **Try... Finally** para asegurarse de ello.

Entender la lógica del iterador de hilos no es complicado si se está atento a algunos detalles. En su creación, el iterador inicializa el índice *fIndex* a -1, con un funcionamiento similar al de cualquier lista de objetos. Ese índice, almacena la posición actual.

En el ejemplo concreto, el bucle **while ... do**, con la condición *it.Next <> nil*, recorrerá toda la lista de hilos, desde el primero de ellos hasta el último, permitiendo, mediante la llamada al método *Current* del iterador, obtener un puntero a cada uno de los objetos TJMHiloBusqueda que son guardados en su interior. Finalmente, cuando ya no es necesarios, se procede a destruir la instancia creada. Sencillo, ¿no?

Si lo pensáis bien, el uso de esta clase que itera sobre la lista se puede complicar tanto como se desee. El ejemplo que nos presenta Jose Manuel es sencillo, y permite verlo con claridad. Esto es de agradecer, ciertamente.

Lo mejor es que experimentéis con el ejemplo, ejecutando paso a paso, y haciendo un punto de parada en aquellos procedimientos que queráis ver con mayor detenimiento.

La versión definitiva de Salvador...

¿Qué es lo que hace diferente la versión del tercer borrador a ésta? Esa es la pregunta que queda en el aire y que me corresponde responder en este apartado. Veámoslo.

Decíamos, anteriormente, que una buena parte de los problemas quedaban resueltos con el tercer borrador. Ya podíamos incluir varias instancias del componente en un form sin que ello presentar mayor problema, puesto que algunas de las modificaciones que se habían efectuado, permitían que ya fuera así. La lista de nodos dejaba de ser un objeto global y pasaba a pertenecer al buscador, de forma que cada buscador gestionaba su propia lista de nodos. Una buena parte de las aportaciones que hacía el tercer borrador iban a permanecer en la versión “definitiva”. Sin embargo, algunos de los cambios y concretamente uno de ellos, es el que me parece más relevante: la desaparición de las zonas críticas y la aparición de cinco procedimientos que protegen el acceso a los objetos del hilo principal mediante *Synchronize()*. Existen estos cinco procedimientos:

```

procedure EncuentraItem;
procedure ExtraeNodo;
procedure SumaNodo;

```

```

procedure SumaNodoArbol;
procedure CoincidenciaNodoArbol;

```

Como podéis observar, los dos últimos, nos indican que ya se ha incluido el árbol de resultados en nuestro componente. Básicamente tiene la misma funcionalidad que le pedíamos como clientes imaginarios en nuestro artículo anterior. Hemos mantenido que su ascendiente siga siendo la clase TWinControl, en la creencia de que, ahora que ya funciona correctamente, podamos crear otro descendiente del buscador que tenga en su interior una instancia a otro componente que nos sea necesario. Quizá pueda ser un botón, cualquier otro objeto que se nos ocurra... En el ejemplo que preparé, se han creado dos instancias del Buscador, para que pueda comprobar el lector que efectivamente funciona sin problema alguno. No olvide el lector si lo utiliza, como en esta ocasión, en tiempo de ejecución y no en tiempo de diseño, asignar el Parent a nuestro componente. Esto resulta transparente en tiempo de diseño, justo en el mismo momento en que un componente es depositado sobre otro componente pero en tiempo de ejecución hay que hacerlo explícitamente.

```

TsjBusqueda = class(TThread)
private
  fNuevaBusqueda: PNode; // puntero a estructura de nodo
  fMiNodo: PNode;
  //Referencias a los árboles
  fNodoPadre: TTreeNode; // nodo padre de nuestro nodo actual
  fNodo: TTreeNode; // apunta al nodo del arbol de resultados actual

  fBuscador: TsjBuscador; // componente buscador que inicia la ejecución
  fRuta: string; // ruta a buscar
  fCarpetaHilo: string; // ruta (variable auxiliar)
  fSubcarpetas: boolean; // ¿hay subcarpetas?
  fResultado: boolean; // ¿hay coincidencia en la búsqueda?
  //¿Se ha encontrado archivos en esa carpeta

  fItem: String;
  procedure OnEnd(Sender: TObject); // al finalizar su ejecución
protected
  procedure Buscar; virtual;
  procedure ObtenerCarpetas(const ARuta: String); virtual;
  procedure FijaNuevaBusqueda(const ARuta: string); virtual;
  function PreparaNuevaBusqueda: Boolean;

  procedure EncuentraItem; // Métodos sincronizados
  procedure ExtraeNodo; // -----
  procedure SumaNodo; // -----
  procedure SumaNodoArbol; // -----
  procedure CoincidenciaNodoArbol; // -----

public
  constructor Create(const ABuscador: TsjBuscador; const ANodoPadre: TTreeNode; const
ARuta: string; const ASubcarpetas: boolean);
  procedure Execute; override; // Método execute de la clase TThread
end;

```

Listado 2 : Interfaz de la clase TsjBusqueda. Algunos métodos son añadidos

Con respecto a la clase TsjBusqueda no hay más modificaciones que sean realmente importantes. En el **listado 2** podéis ver la definición de dicha clase, con los métodos nuevos añadidos.

Con respecto a la clase TsjBuscador, existe una clase auxiliar añadida: TPendientes. Al recibir la versión definitiva de Jose Manuel, y tras ese primer vistazo a la implementación de la clase TIteradorHilos,

pensé en la necesidad de añadir a mi prototipo, una clase que gestionara la lista de nodos, pensando que de esa forma, se facilitaba la lectura del código fuente y se reducía la posibilidad de equivocarse en la liberación de la memoria que los nodos habían reservado. El proceso, tanto de creación como de destrucción de los nodos, se centralizaba en este objeto TPendientes, y nos limitábamos a invocar sus métodos, que nos facilitaba la gestión de la lista de nodos. En el **listado 3** se puede apreciar los métodos que define esta clase:

```
TPendientes = class
private
  fListaNodos: TList;
  fBuscador: TsjBuscador;
  function GetListaPendientes(Indice: Integer): TNodo;
public
  procedure Clear;
  procedure AddNodo(Nodo: PNodo);
  function ExtractFirst: PNodo;
  constructor Create(const ABuscador: TsjBuscador);
  destructor Destroy; override;
  property ListaPendientes[Index: Integer]: TNodo read GetListaPendientes;
end;
```

Listado 3 : Interfaz de la clase TPendientes. Es una clase auxiliar de TsjBuscador.

De esa forma, al invocar el procedimiento *Clear*, por ejemplo, recorriamos la lista de nodos y tras liberar la memoria asociada a cada uno de los nodos, se eliminaban los ítems de la lista, limpiando la misma. Ya no era necesario, repetir este bucle en varios procedimientos del Buscador de ficheros sino tan solo la invocación del método *Clear*, eliminando la posibilidad de modificaciones involuntarias en tan solo una parte de las apariciones de este código: -Ahhh... me olvidé cambiarlo... cambié aquí pero no allí..., puede ser una frase que se repite en ocasiones.

```
procedure TPendientes.Clear;
var
  i: Integer;
  ANodo: PNodo;
begin
  for i:= 0 to fListaNodos.Count - 1 do begin
    ANodo:= fListaNodos[i];
    Dispose(ANodo);
    fListaNodos[i]:= Nil;
  end;
  fListaNodos.Clear;
end;
```

Y como decíamos anteriormente, en esta versión definitiva, desaparece el procedimiento *SubClassWndProc()* -el nombre lo he copiado de uno de los ejemplos de la ayuda en línea de Delphi-. Desaparecerá también la zona crítica que definíamos para el acceso a la lista de nodos, ya que el acceso a la misma desde cada hilo, se hace envolviendo el método en *Synchronize()*, quedando protegido por la zona crítica que se crea y ejecutándose todos los métodos desde el contexto del hilo primario.

Qué cada cual saque sus propias conclusiones...

Es bueno, cuando se acaba de analizar un tema, como lo hemos hecho durante estos 6 artículos, en donde hemos intentado abordar los aspectos principales tanto desde una perspectiva teórica como práctica,

arriesgarse a sacar uno mismo sus propias conclusiones, aun a riesgo de equivocarse... es parte del proceso de aprendizaje.

Entre esas conclusiones yo me he hecho mi propia lista, anotando en esa lista imaginaria aquellos puntos que debo vigilar con mayor atención. Yo he anotado tres puntos que me parecen importantes:

- “*reduce tanto como puedas los accesos al hilo primario...*”.
- “cuando utilices la clase TThread, *haz uso de Synchronize* para acceder a los objetos del hilo primario... y olvídate -si te es posible- de definir zonas críticas para algo que ya te da este método... Te evitarás problemas.”
- “crea los hilos tan *solo cuando sea realmente necesario*; y si lo haces, no lo hagas de forma indiscriminada, poniendo en peligro la estabilidad del sistema.... Los recursos del sistema también son limitados.”

El primer punto y el segundo están muy relacionados entre si. Es habitual que, inevitablemente, los hilos acaben elaborando algún resultado en base a valores que se obtienen en el hilo principal de la aplicación, y que acaben depositando el resultado de ese proceso en algún objeto del mismo. Si pensamos que todos los métodos que finalmente sean protegidos por Synchronize, van a ser ejecutados de forma secuencial, podemos razonar que la velocidad en la ejecución del algoritmo va a estar condicionada por la forma y la frecuencia en que se produzcan estos accesos al hilo primario: “algún hilo siempre deberá esperar a que otro salga de una de las zonas definidas como críticas”. Por lo tanto, parece lógico que se pueda pensar en reducir en lo posible la frecuencia de acceso al hilo primario y el tamaño del código que va a intervenir en cada uno de esos accesos.

El último también es un razonamiento que me parece de sentido común y no merece demasiado comentario adicional. Además, como hemos visto, la depuración no es tan sencilla y a menudo nos va a llevar algunas horas hasta descubrir cual de las líneas de código generan el problema.

Cualquier comparación es odiosa...

Cuando el resultado de una comparación va en favor de uno mismo siempre es bienvenida... pero en este caso, parece que en igualdad de condiciones, la última versión del buscador de Jose Manuel es mucho más rápida. Hay que aceptarlo con buen humor... Para verlo he hecho un pequeño ejemplo comparativo que podéis encontrar junto con las fuentes, y en la carpeta cuyo nombre es *comparativa*. Uno de los resultados cualquiera da una idea de esas diferencias: En una búsqueda doble en el disco c: sobre un número total de 7349 carpetas por búsqueda, la versión de Jose Manuel obtiene una marca de 6,72 segundos para una búsqueda en 7349 x 2 carpetas. Mi buscador lo hace en 8,36 segundos sobre las mismas. Ambas obtienen 1251 x 2 coincidencias. Esta diferencia de 1,64 segundos, supone mas de un 15 % de ganancia en rapidez.

Habría que repetir muchas veces distintas pruebas y ejecuciones y obtener medias de los resultados pero a simple vista se puede entender el porque puede ser así. Ambas versiones crean un solo hilo por cada una de las búsquedas pero en mi caso, un parte de ese tiempo se emplea en reservar memoria en la lista de nodos , donde se almacenan las futuras rutas en las que buscar. Así mismo hay que perder un tiempo para liberar esa memoria que se reserva de forma dinámica.

Tampoco la forma en que se gestiona la introducción de la ruta y su tratamiento es igual. La versión de Jose Manuel, permitiría inclusive que existieran varios Token, mientras que en el planteamiento mío eso no resulta posible. Tendría que incorporar esos pequeños cambios para dotar de la misma funcionalidad. Es una posibilidad.

Sin embargo, no todo se ha perdido... con respecto a Windows, y pidiéndole que nos busque en las mismas o parecidas condiciones, se obtiene una diferencia sustanciosa en ambas versiones. Windows necesita alrededor de 16,4 segundos para la misma búsqueda referida en líneas anteriores. Y nuestro componente, podría obtener una ganancia todavía mayor si ésta se hiciera con unidades de red conectadas, dado que estas unidades también participarían en la búsqueda. Otra razón de peso para considerarlas...

En este apartado ya me queda poco que decir... creo que he comentado todo lo que tenía anotado como "pendiente" y no me dejo nada en el tintero. Como comentaba, líneas mas arriba, posiblemente se puedan subir ambos componentes al servidor del grupo albor, con la idea de que podáis participar y añadir, modificar o eliminar aquellos detalles que se puedan mejorar. Sería muy bueno para nosotros también, para el Grupo Albor, para los compañeros de Síntesis, saber que estáis ahí y que os importa esto que hacemos. Parece que el camino se hace menos largo cuando se recorre en compañía...

Nos despedimos.

Es un buen momento para despedirse: El sol va muriendo lentamente..., en ese horizonte rojizo, que adivina los primeros racimos de estrellas. Parece que nos invitan a salir, en busca de una pizca de aire fresco, a dejar nuestros sueños en algunos de esos racimos... Así que voy cerrando las ventanas que tenía abiertas hace unas horas, el entorno de Delphi 5, el editor, la conexión de internet... y esta pequeña serie sobre la clase TThread que hemos podido compartir.

Lo dicho, otro día seguimos conversando...

Gracias Jose Manuel...