

TThread V: Un buscador de Archivos (I)

Salvador Jover – s.jover@wanadoo.es

Cualquier momento es bueno para poner en práctica lo aprendido..., aunque casi siempre acabemos descubriendo, que del dicho al hecho anda un buen trecho... Vamos a plantear el desarrollo de un Buscador de Ficheros multihilo, con la colaboración de Jose Manuel Navarro.

¿Por donde comienzo esta historia, Jose Manuel?... Umm... Quizás por comentar en estas líneas del artículo, las primeras reflexiones que hacíamos en varios correos, al principio de esta serie:

Como sabéis, la serie sobre hilos de ejecución, se inicia en el número 11 de la Revista Síntesis. Era Octubre del 2002, y recién publicado el primer artículo de introducción, ya preveía que se haría necesario algún ejemplo más complejo. Así pues, ni corto ni perezoso, tras el comentario en la lista en donde nos vemos de cuando en cuando los colaboradores de Síntesis, Jose Manuel se ofreció a ayudarme a poner un poco de imaginación al asunto, que ya sabéis que no me sobra demasiada. Acepté sin dudar... ¿habéis leído sus artículos...?. Era una garantía de que la cosa saldría bien.

El primer correo que cruzamos tiene fecha de 31 de Octubre. Es la primera toma de contacto que tuvimos, y me entregó, adjunto al mismo, el algoritmo prácticamente resuelto. Se trataba de un buscador de ficheros, con un planteamiento “recursivo”, con la particularidad de que la búsqueda recursiva la efectuaba mediante la creación de un número indefinido o indeterminado de hilos. Se iba a generar un hilo para cada uno de los directorios encontrados, por lo que, aún sabiendo que la vida de éstos (los hilos) iba a ser muy corta, a medida que se profundizaba en el árbol, convivían y malvivían hacinados. ¿Cuántos...? No se... depende de la profundidad del árbol y del número de carpetas.

¿Por qué comento todo esto...?

Veréis: cuando iniciamos este ejemplo, no era con la idea de entregar un desarrollo impoluto, presentado como un 'producto' del que solo ves la imagen fría, y no los pasos, ni los problemas que habían ido surgiendo a medida que se reflexionaba sobre la creación de hilos y su diseño. Parecía mucho más instructivo preparar este artículo y los venideros, como una evolución de la idea original, la que nos pidió nuestro “cliente”, y que posiblemente, a lo largo del razonamiento, irá ganando y perdiendo, según el caso. Ni siquiera la explicación, línea a línea del código fuente, aún siendo importante, puede sustituir en ocasiones a la necesidad de “visualizar” la imagen evolutiva del desarrollo, porque al final, se perdía lo más importante, y en lo que intentábamos hacer hincapié en los primeros dos artículos: los desarrollos que hacen uso de múltiples hilos, presentan problemas que tienen una relación directa “temporal,” y que pueden no repetirse en sucesivas evaluaciones o pruebas. A mi profesor, Higinio, en la Universidad, le he escuchado decir en innumerables ocasiones, que la no aparición de un problema no supone siempre que éste no pueda existir. Simplemente no se ha manifestado. En otras condiciones, en otro Pc., teniendo otros procesos abiertos, etc... quizás no se de y permanezca latente.

Es decir, que de alguna forma, si esperáis un ejemplo de quita y pon, algo ya listo para el uso, vale la pena que no avancéis una línea más del artículo, porque os hará perder mucho tiempo. En la red y

rebuscando un poco, es muy fácil encontrar algo acabado e incluso gratuito. La idea que perseguimos es que sea útil de verdad a quien se inicia en Delphi también.

Para los que nos quedemos, y sigamos avanzando en la lectura de éste y los próximos, vamos a intentar iniciar esta narración como Dios manda... respondiendo a la primera pregunta que parece sensato plantearse: ¿Que me pide este cliente...?

Así vamos a hacerlo. Toda vez que ya hemos comentado en los cuatro artículos anteriores los aspectos más teóricos, nos metemos en harina de un supuesto práctico y casi real, al menos no en la cantidad de hijos...

El cliente quería...

El cliente siempre tiene la razón. Es así... la regla máxima de cualquier negocio.

A pesar de saber que en Windows existe un buscador, y que busca -fácilmente- lo que el usuario le pide, (si no se pasa pidiendo), se empeñaba mi cliente en que quería un buscador de ficheros, como una opción más del programa de gestión que le estaba preparando . No sería yo quien le contrariase... Si me hubiera pedido que el buscador, cantara y bailara cada vez que encontrara un fichero, también me hubiera parecido razonable. Si no a mí, a mis cinco churumbeles sí.

Eso sí. Le había intentado explicar que también podía buscar en dos directorios simultáneamente, aunque solo fuera por dejar la conciencia limpia de que no le iba a programar algo que ya tenía a mano, disponible. ¿Lo habéis hecho alguna vez? Me refiero a la búsqueda. La verdad es que no me lo había planteado, y siempre que buscaba algo, seleccionaba un directorio y finalizada ésta, resolvía otra.

Pulsad “Inicio”, “Buscar” y “Archivos y carpetas”. Seleccionad varias rutas de búsqueda: “**C:\; G:\;C:\Archivos de Programa**” y un fichero tal que “***.bmp**”. Si lo habéis hecho, en la ventana de resultados, se ordenaran los resultados siguiendo el orden de las peticiones. Posiblemente sea porque se utiliza un único hilo y resuelve la búsqueda de forma serial. Primero busca en “**C:**” y en todas sus subcarpetas. Luego busca en “**G:**”, también en todas sus subcarpetas. Y finalmente en “**C:\Archivos de Programa**” y también en todas sus subcarpetas.

Así que lo que nos pide el cliente es que desarrollemos un buscador de ficheros (eso en principio, porque mas tarde, con seguridad que desearía extender la utilidad del mismo).

Manos a la obra. Vamos a imaginar cómo es y cómo se comportaría dicho objeto, añadiéndole algunos atributos más de nuestra cosecha, con la idea de acercarnos mas el tema del trabajo y a los desarrollos con múltiples hilos de ejecución:

El objeto buscador de ficheros debería de ser capaz de encontrar una coincidencia, a petición del cliente, dentro de una o varias rutas, pudiendo descender a todas y cada una de las carpetas o rutas interiores a éstas, según el deseo expresado por nuestro usuario en la petición. A diferencia del implementado por Windows, el nuestro, crearía un hilo por cada una de esas rutas, como mínimo; de haber contemplado la posibilidad de trabajar con un sistema multiprocesador, se deberían resolver éstas con mayor rapidez. Ya comentamos al principio de la serie y en el artículo de introducción, que la multitarea, en sistemas Windows como por ejemplo Windows 98, era simulada y que, a pesar de hacernos creer que los distintos procesos concurrían, simultáneamente, esto era más una ilusión óptica

que real. Podría nuestro cliente hacer uso de los dos comodines clásicos (* y ?) finalmente, y los resultados se expondrían en algo similar a un cuadro de lista.

Además, dicho cliente, va a condicionar un poco nuestro interfaz gráfico, al pedirnos poder revisar la exploración de directorios efectuada sobre un árbol visual, donde se muestren todas las carpetas que han sido encontradas. ¡Algo desconfiado este hombre...!, ¡Será por asegurarse de que no nos dejamos alguna por recorrer!

Si seguimos razonando sobre los requisitos de este 'objeto' buscador, o más bien, de aquellos parámetros necesarios para poner en marcha todo el proceso de búsqueda, razonaríamos lo siguiente: el usuario debería disponer de algún método para hacerle llegar al buscador, las distintas rutas sobre las que considerar la búsqueda, un parámetro adicional para saber si debe descender a las subcarpetas, y finalmente el fichero de coincidencias.

Es decir: tenemos al menos tres parámetros que debe conocer el objeto Buscador. En el caso del binomio "ruta&fichero", vamos a considerarlo como uno solo. Será el algoritmo de búsqueda el que discrimine ambos, una vez en el interior del objeto THiloBusqueda.

Nos podemos mentalmente imaginar un poco la secuencia de pasos que seguirá nuestro usuario, aunque sea esta imagen mental poco académica y no sujeta a ningún formalismo.

```
INICIO
RUTAS -----> Buscador.RecibeRutaoRutas
TOKEN o FICHERO -----> Buscador.RecibeToken
¿HAY SUBCARPETAS?-----> Buscador.RecibeValorVerdad
                          Buscador.EvaluaDatosCorrectos----> ERROR ---> INICIO
                                                                ----> NO ERROR -->ALGORITMO
                                                                BUSQUEDA
                          ....
                          Buscador.DevuelveResultados
FIN
```

¿Qué es lo que sucede en los puntos suspensivos? En este punto del razonamiento todavía no lo sabemos, pero suceda lo que suceda, realmente tampoco nos preocupa, al menos de momento. El resultado final es que nuestro Ente buscador, partiendo de datos válidos y de un estado consistente, nos debería entregar un resultado correcto, y que consistirá, finalmente, en una lista con todos los archivos encontrados.

Este comentario puede parecer anecdótico pero en realidad no lo es tanto. Intentaré explicarlo: Es posible que nuestra primera tentación sea la de diseñar el interfaz de usuario. Llenamos nuestro *form* con objetos que van a responder, relacionándose, en cada uno de los pasos del algoritmo. Podemos insertar un componente TListBox o un TListView para mostrar los resultados. Elegimos un par de botones que responderán a Ejecutar o Cancelar. Dos casillas de edición (TEdit) para introducir la ruta y el nombre del fichero. Y finalmente un TCheckBox, para la selección ¿Hay subcarpetas?. Diez minutos a lo máximo, para montar este entramado de objetos... Si os acordáis del cuento de los tres cerditos, en donde había una casita de paja. Y también un lobo para derrumbarla de un solo soplido...

Programar así resulta "cómodo". El botón ejecutar dispone del código necesario para lanzar la ejecución del algoritmo. El botón cancelar también para la acción asociada. Así sucesivamente. Creo que

se entiende lo que intento explicar. A poco que crezca nuestro desarrollo, a poco que los objetos visuales no se ajusten totalmente a la lógica del algoritmo, a poco que nuestro cliente nos pida nuevos requerimientos, irán creciendo de forma exponencial nuestros problemas.

Suponed que pasado un mes, leyendo en un libro al azar, encontramos un nuevo algoritmo de búsqueda mucho más eficaz, ¿va a ser fácil esa posible sustitución? Pues depende...

Ahí está la clave que motivó este razonamiento. Lo que se esconde tras los puntos suspensivos debería ser similar a la acción de cambiar una bombilla cuando se funde: algo sencillo. En eso nos van a ayudar los objetos y el nivel de abstracción que va implícito en ellos. Es algo más de trabajo, pero suele compensar a largo plazo. Intentémoslo:

Vamos a considerar dos entidades. La primera la llamaremos a partir de ahora TBuscador, y representa a la petición del usuario. La segunda entidad la podemos llamar THiloBusqueda y puede representar al algoritmo utilizado. De esa forma, nuestro diseño nos debería conducir a que fuera factible sustituir o modificar el comportamiento de la entidad THiloBusqueda, con un comportamiento distinto, sin que TBuscador se viera visiblemente afectado por ello.

Ahora ya podemos empezar a pensar en nuestro algoritmo de búsqueda, aunque solo sea para hacernos una idea de cual va a ser el comportamiento del objeto THiloBusqueda, del que todavía sabemos más bien poco. Vamos a imaginar un planteamiento recursivo. Veamos un supuesto:

El usuario nos ha pedido que busquemos en "C:\" el fichero "*Dinosaurio.txt*", con búsqueda en subcarpetas. En ese caso, THiloBusqueda, debería situarse en la carpeta seleccionada por el usuario, y ver si existen ficheros que coincidan con el pedido. Hecho esto, anotaría todas las carpetas que ha encontrado dentro de la pedida y para cada una de ellas, crearía otro THiloBusqueda, con la única misión de hacer lo mismo que él ha hecho. Esto supone necesariamente que el árbol de directorios será recorrido totalmente, mientras existan carpetas por explorar... Eso sí... con un coste algo elevado, puesto que nos va a ser necesaria la creación de un objeto THiloBusqueda, para cada una de las carpetas.

Nuestro primer borrador de trabajo.

En la carpeta de fuentes que acompaña a este artículo "*1er Borrador Multifind*", encontraréis el primer boceto que se nos ocurrió y que vamos a comentar. Buscad el módulo "*Buscador.pas*" de dicha carpeta.

Nos centraremos en primer lugar en el procedimiento *Execute* de THiloBusqueda. Nos olvidamos por un momento de todo lo que resulte accesorio en él, eliminando aquel código que no resulte decisivo en la comprensión del algoritmo. Se trata de intentar comprender qué está haciendo:

```
procedure TBusqueda.Execute;  
var  
  FindData: WIN32_FIND_DATA;  
  SearchHandle: THandle;  
  Hay: Boolean;  
  ...  
begin  
  Hay:= false;
```

```

                                PRIMERA FASE DE BUSQUEDA: CARPETAS QUE CONTIENE
if FSubcarpetas then begin                                CONDICION INICIAL: ¿HAY SUBCARPETAS?
    SearchHandle := FindFirstFile(ruta, FindData); SE SITUA EN EL PRIMER FICHERO
    if SearchHandle <> INVALID_HANDLE_VALUE then
    begin
        repeat                                          REPITE MIENTRAS QUEDEN CARPETAS
            if (FindData.dwFileAttributes and FILE_ATTRIBUTE_DIRECTORY <> 0) and
                (FindData.cFileName[0] <> '.') then
            begin                                       PARA CADA UNA DE ESAS CARPETAS ENCONTRADAS
                Busqueda := TBusqueda.Create(FBuscador, FNode, RutaArg, true);
                Busqueda.Resume;                          LANZA OTRA BUSQUEDA
            end;                                         ENTREGANDOLE LOS PARAMETROS NECESARIOS
        until not FindNextFile(SearchHandle, FindData);
        // error en algún paso de la búsqueda
        // aquí van las rutinas para su tratamiento
    end;                                               FIN DE LA CONDICION ¿HAY SUBCARPETAS?
                                                    FIN PRIMERA FASE DE BUSQUEDA
-----
                                SEGUNDA FASE DE LA BUSQUEDA: BUSCAMOS COINCIDENCIA CON EL NOMBRE DEL FICHERO
// busco el archivo en la carpeta actual
SearchHandle := FindFirstFile(carpeta, FindData); SE SITUA EN EL PRIMER FICHERO
if SearchHandle <> INVALID_HANDLE_VALUE then
begin
    ...
    // Se itera en la carpeta actual
    repeat                                          REPITE MIENTRAS QUEDEN CARPETAS
        if not ((FindData.dwFileAttributes and FILE_ATTRIBUTE_DIRECTORY <> 0) or
            (FindData.cFileName[0] = '.')) then
        begin                                       HAY COINCIDENCIA
            ...
            ultimo := FBuscador.FLista.Items.Add; AÑADO A LISTA DE RESULTADOS
            ultimo.Caption := ExtractFilePath(carpeta) + FindData.cFileName;
            ...
            Hay := true;
        end;
    until not FindNextFile(SearchHandle, FindData);
    // error en algún paso de la búsqueda
    // aquí van las rutinas para su tratamiento
end;                                               FIN SEGUNDA FASE DE LA BUSQUEDA
-----
FResultado := Hay; // ¿ha encontrado ficheros en esa carpeta?
end;

```

Básicamente es el mismo esquema repetido en dos ocasiones. Una primera, para las carpetas, hasta localizar todas las contenidas en la considerada. Y otra segunda, para todos los ficheros, coincidentes con el Token, o fichero que desea encontrar el usuario.

Windows me echa una mano.

Tiene mala fama en algunos círculos, pero no es tan malo como parece. A nosotros nos va a ayudar en esta ocasión con el uso de algunas funciones que pone a nuestra disposición, como *FindFirstFile()* o *FindNextFile()*. Ambas son responsables de que podamos montar el bucle que gestiona la búsqueda de ficheros dentro de una carpeta. Nos vamos a detener un momento en ambas.

Primeramente tenemos que conocer el registro TWin32FindData

```
typedef struct _WIN32_FIND_DATA {
    DWORD      dwFileAttributes;           ATRIBUTOS DEL FICHERO
    FILETIME   ftCreationTime;            - CREACION DEL FICHERO
    FILETIME   ftLastAccessTime;         - ULTIMO ACCESO
    FILETIME   ftLastWriteTime;         - ULTIMA MODIFICACION
    DWORD      nFileSizeHigh;            TAMAÑO DEL FICHERO
    DWORD      nFileSizeLow;            TAMAÑO DEL FICHERO
    DWORD      dwReserved0;              ---
    DWORD      dwReserved1;              ---
    TCHAR      cFileName[ MAX_PATH ];    NOMBRE LARGO DEL FICHERO
    TCHAR      cAlternateFileName[ 14 ];  NOMBRE CORTO (FORMATO 8.3)
} WIN32_FIND_DATA;
```

Recoge los atributos que habitualmente podemos ver, al ejecutar con el botón derecho de nuestro ratón, sobre un archivo en la sección de propiedades. Algunos de los campos de este registro, nos podrían servir para completar la información que nos presenta la lista de resultados. Windows lo hace así. Nos da el nombre, la ruta, la fecha de última modificación, etc... De momento, nos quedamos tan solo con el nombre largo.

Para acceder a los valores de este registro, hacemos uso de las funciones nombradas anteriormente y que vamos a detallar a continuación. Ambas funciones son declaradas en el módulo *Windows.pas*:

| | | |
|----------------------------------|----------------------------------|------------------------------|
| HANDLE FindFirstFile (| HANDLE FindNextFile (| <u>TIPOS EQUIVALENTES A:</u> |
| LPCTSTR lpFileName, | LPCTSTR lpFileName, | PChar |
| LPWIN32_FIND_DATA lpFindFileData | LPWIN32_FIND_DATA lpFindFileData | var TWin32FindData |
|); |); | |

El primer parámetro de ambas funciones es un puntero a una cadena de caracteres acabada en nulo, conteniendo la ruta y el nombre del fichero. El segundo parámetro, un puntero al registro TWin32FindData, con la información adicional mencionada.

Tenemos que tener en cuenta, que esto que estamos comentando, es aplicable tanto a ficheros como a subcarpetas. Con la primera función nos situamos sobre el primer fichero o subcarpeta del directorio actual. Con el segundo, avanzamos en ese buque hasta recorrerlos todos.

Si buscáis en las líneas de código anteriores, corresponderían al fragmento:

```
repeat
...
until not FindNextFile(SearchHandle, FindData);
```

Es posible que tampoco conozcamos el uso del procedimiento *RaiseLastWin32Error*. En el código anterior ha sido excluido y correspondería al bloque posterior a la búsqueda, donde se evalúa el error.

Este procedimiento es implementado en el módulo *SysUtils.pas* y su misión es limpiar el posible error producido en la llamada a las funciones de Windows. Internamente lo único que hace es llamar a *GetLastError.*, y lanzar la excepción.

```
procedure RaiseLastWin32Error;
var
  LastError: DWORD;
  Error: EWin32Error;
begin
  LastError := GetLastError;
  if LastError <> ERROR_SUCCESS then
    Error := EWin32Error.CreateResFmt(@SWin32Error, [LastError,
      SysErrorMessage(LastError)])
  else
    Error := EWin32Error.CreateRes(@SUnkWin32Error);
  Error.ErrorCode := LastError;
  raise Error;
end;
```

Y por último, un detalle: es necesario invocar a *FindClose()*, declarado en *Windows.pas* para cerrar el manejador (*Handle*) de búsqueda de ficheros, que era el valor de retorno de las dos funciones primeramente comentadas.

Podemos comentar alguna conclusión que nos parezca interesante. Ejecutad el ejemplo correspondiente al primer borrador y lo vemos:

Conclusiones sobre el primer borrador.

El primer desarrollo se compone principalmente de tres módulos de código: el de proyecto (*multifind.dpr*), el principal, denominado *main.pas*, y que contiene los elementos gráficos de la interfaz, y el módulo *buscador.pas*, componiendo a los objetos *TBuscador* y *THiloBusqueda*. Este último módulo, recoge toda la lógica del buscador. Veis?. Aquí se puede apreciar el primer comentario que hacíamos. Jose Manuel ha encapsulado el objeto *Buscador*, aislándolo en lo posible de la interfaz gráfica. Y digo “en lo posible”, porque de alguna forma, la necesidad de que aparezca el árbol de resultados y la lista de resultados nos condiciona.

¿Se ejecuta correctamente?

Sí. Aparentemente puede ser una de nuestras primeras conclusiones. Es rápido. Comparémoslo con el buscador de Windows en una consulta con la mismas características. Vale... Primero lo ejecuto en un equipo moderno, reciente, un PIV a 2000 Ghz, 512 Mbs de RAM, con Windows 2000. Luego hago lo propio con otro muy modesto: un Pentium 166 AT 64 Mbs de RAM, con Windows 98. Sin embargo no todas las pruebas resultan exitosas y en algún momento el buscador parece quedar congelado, recuperándose poco después.

Hacíamos ese comentario, Jose Manuel y yo, tras cruzar varios correos en donde analizábamos todo este tipo de aspectos. Funcionar funcionaba, pero... Sigamos:

A todo esto, lo siguiente que decidí fue observar la ejecución del programa desde Delphi. Lo ejecuté varias veces, visualizando los hilos creados en la ventana del depurador de Hilos. Mi idea hasta ese momento era que, siendo la vida de cada hilo tan corta, el tiempo que dura en explorar una carpeta y obtener todos los ficheros coincidentes, convivirían una cantidad razonable de hilos durante toda la ejecución del algoritmo de búsqueda. Otra idea equivocada. Aquello se iba de las manos... la lista de hilos crecía en ocasiones espectacularmente, de forma desmesurada y puedo suponer que los cambios de contexto iban a reléntizar mas si cabe, agravando la situación. Posiblemente, era lo que sucedía en aquellas ocasiones en que el programa parecía congelarse.

¿Algún detalle más que se pudiera apreciar?

Un detalle que tiene en este momento de la narración una importancia relativa, pero que posteriormente nos afectará. Hasta el punto de que tan solo me han quedado dos días para escribir este artículo, revisando el problema que ahora os comento: De cuando en cuando, al cerrar la ventana principal, se generaba una excepción y no lo hacía siempre, solo en aquellas ocasiones en que el Buscador estaba ejecutándose. Si el algoritmo finalizaba, no. La ventana se cerraba y no producía excepción alguna.

En la **figura 1** se puede apreciar una imagen capturada de la ejecución de este primer borrador creado.

El problema es que tenemos que valorar la existencia de usuarios nocivos para nuestros programas. Son ese tipo de usuarios que nunca esperan a que finalice la búsqueda, que andan haciendo “zapping” de ventana en ventana, con sus “clicks” erráticos y ciertamente “perversos”. Son capaces de detectar un error en nuestra aplicación y pensar que esconde alguna puerta secreta hacia no se que lugar. ¡como si el programador no tuviera suficientes problemas como para ir dejando puertas ocultas! Al menos los que yo conozco. Es decir, que no podemos suponer que siempre que sea ejecutado el buscador el usuario esperará pacientemente a que este termine, antes de cerrar la ventana. Lógico, ¿no?. Tomamos nota y ya veremos de corregirlo en la segundo intento de resolver el desarrollo.

No hemos comentado nada sobre el TBuscador en sí, y no es porque no haya nada que comentar sino por que, lo vamos a hacer un poco más adelante. Un comentario sí que parece oportuno, y es la convivencia de ambos clases, TBuscador y THiloBusqueda, conviviendo en el mismo módulo. En el siguiente desarrollo separaremos ambas y convivirán en módulos distintos. La clase TBuscador permanecerá en el módulo “*Buscador.pas*”, mientras que THiloBusqueda, será alojada en el módulo “*HiloBusqueda.pas*”.

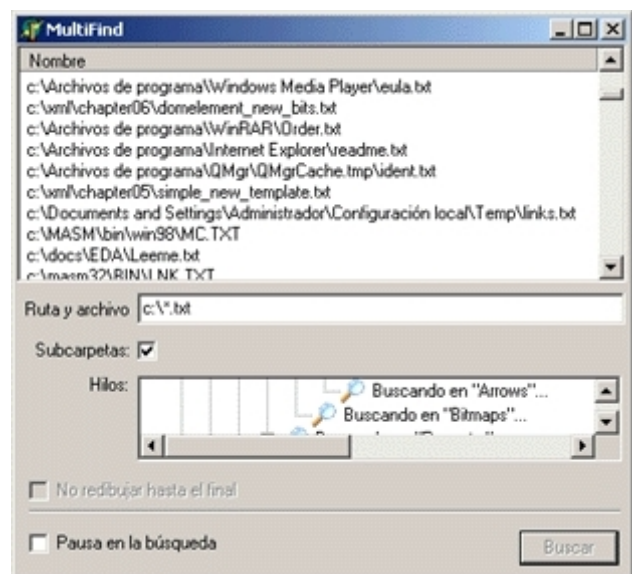


Figura 1. Primer borrador del programa “Multifind.exe” en ejecución.

¿Cuál va a ser la consecuencia inmediata? Desde luego, la primera consecuencia es la ocultación de información, dado que la visibilidad entre ambas clases se vera restringida a los métodos públicos y propiedades publicadas del componente. La convivencia en un mismo módulo hace “amigas” a dos clases y esto supone que se pierden las características propias de la encapsulación. Haced una prueba. Os podéis situar por ejemplo en la línea 242 del módulo “Buscador.pas”:

```

type
  TBuscador = class(TWinControl)
  private
    ...
    ...
    procedure baIncPos( var Msg: TMessage); message BA_INCPOS;
    procedure baIncMax( var Msg: TMessage); message BA_INCMAX;
    procedure reNumItem( var Msg: TMessage); message RE_NUMITEM;
    procedure reSelect( var Msg: TMessage); message RE_SELECT;
    procedure reAdd( var Msg: TMessage); message RE_ADD;
    procedure reDel( var Msg: TMessage); message RE_DEL;
    procedure arAdd( var Msg: TMessage); message AR_ADD;
  protected
    procedure WMSize( var Message: TWMSize); message WM_SIZE;
    procedure DoExecute; virtual;
    procedure DoCancel; virtual;
    procedure AddThread(Hilo: TThread); virtual;
    procedure DeleteThread(Hilo: TThread); virtual;
    property BarraProgreso: TProgressBar read fBarra;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Add(const NewPath: string);
    procedure Execute;
    procedure Pause(parar: boolean);
    procedure Cancel;
    property ArbolResultados: TTreeView read fArbol;
    property MSecs : LongWord read fMSecs default 0;
    property Estado : TEstado read fEstado default esInactivo;
  published
    property OnEnd: TNotifyEvent read FOnEnd write FOnEnd;
    property OnAbort: TAbortEvent read FOnAbort write FOnAbort;
    property Align;
    property Color: TColor read fColor write SetColor;
    property Image: TCustomImageList read fImage write SetImage;
    property ArbolVisible: Boolean read fArbolVisible write SetArbolVisible;
    property Token: String read fToken write SetToken;
    property HaySubcarpetas: Boolean read FSubcarpetas write FSubcarpetas default True;
  end;

```

Listado 1. Definición del componente TBuscador desarrollada en Buscador.pas

(Incluida solo parte de la interfaz)

```
242         Busqueda := TBusqueda.Create(FBuscador, FNode, RutaArg, true);  
243         Busqueda.Resume;
```

Ahora podéis insertar entre ambas:

```
        Busqueda.
```

Y [Code Insight], os mostrará todos los campos, métodos y propiedades accesibles para la referencia. Y estarán todos: campos privados, métodos protegidos, procedimientos y funciones públicas, etc... Mala cosa... ¿no?. Porque a fin de cuentas, al definir un interfaz estamos estableciendo un contrato de nuestra clase con el resto del “mundo” y de alguna forma, hacer “amigas” dos clases puede romper ese contrato. Así que en ese caso, nuestra actitud debería ser, pienso yo, actuar como si realmente estuvieran en módulos distintos, evitando la tentación de hacer, de cuando en cuando, alguna que otra trampa.

Para evitar tentaciones, las separaremos en nuestra próxima versión.

¿Iniciamos una nueva versión, por favor...?

En el **listado 1**, podemos observar la nueva definición que hemos hecho de la clase TBuscador, añadiendo algunas propiedades y eliminando otras. Nos hemos planteado: Si tenemos que hacer uso posteriormente de esta ventana de búsqueda, parecería conveniente que nos hiciéramos la vida un poco más fácil, y nos planteásemos convertir la clase TBuscador en un componente, capaz de ser usado desde Delphi, desde el entorno. Así que ni cortos ni perezosos alteramos la ascendencia de nuestro TBuscador y en lugar de descender de un objeto, le asignaremos una familia mucho más notoria e importante. No descenderá de un triste y pobre TObject, sino que le vamos a dar un poco más de abolengo. Su ascendente directo será la clase TWinControl. Ésta clase puede ser un buen punto de partida para crear componentes que puedan necesitar un Handle o manejador de ventana, y que además, hagan uso de propiedades y eventos comunes a este tipo de controles. Como puede ser este caso, dado que nos habíamos apoyado principalmente en tres componentes: TListView, TTreeView y TProgressBar. Todos descienden directa o indirectamente de TWinControl.

Nuestro componente buscador será un descendiente de la clase TWinControl. En sus campos interiores, almacenaremos una referencia a tres objetos creados, y que corresponden respectivamente a las tres clases citadas. Así pues, cuando deseemos insertar el buscador en otra ventana, tan solo tendremos que arrastrarlo desde la paleta de Delphi y manipularlo desde los métodos y propiedades que hace público su interfaz.

Esta nueva versión la encontraréis en la carpeta “*multifind*” que acompaña a las fuentes del artículo. Allí aparecerán los mismos módulos citados anteriormente, mas el añadido “*hilobusqueda.pas*” y el módulo de instalación del componente “*search.dpk*”, que debería ser instalado previamente. En realidad no es estrictamente necesario, pero si se abre el proyecto, os debería emerger una ventana con el comentario de que el componente no existe. Una solución inmediata es instalar “*search.dpk*”. La otra no tanto, y consiste en facilitar la ruta donde residen los módulos “*buscador.pas*” e “*hilobusqueda.pas*”, en el entorno *[Project/Options/Directories&Conditionals]* y eliminar el componente TBuscador del form, para que sea creado en tiempo de ejecución, de la forma habitual que ya conocemos.

En la **figura2** podéis ver el desarrollo ya terminado, mientras se ejecuta.

Primeros comentarios sobre TBuscador.

No podemos comentar todos los métodos y propiedades del componente. De hacerlo, el lector posiblemente diera por finalizada la lectura del artículo. Realmente no hace falta, porque algunas propiedades que se han añadido son bastante intuitivas. Activamos la propiedad *Align*, escondida en en la clase TWinControl al ser declarada como protegida. Con muy poco esfuerzo, ahora la tenemos disponible.

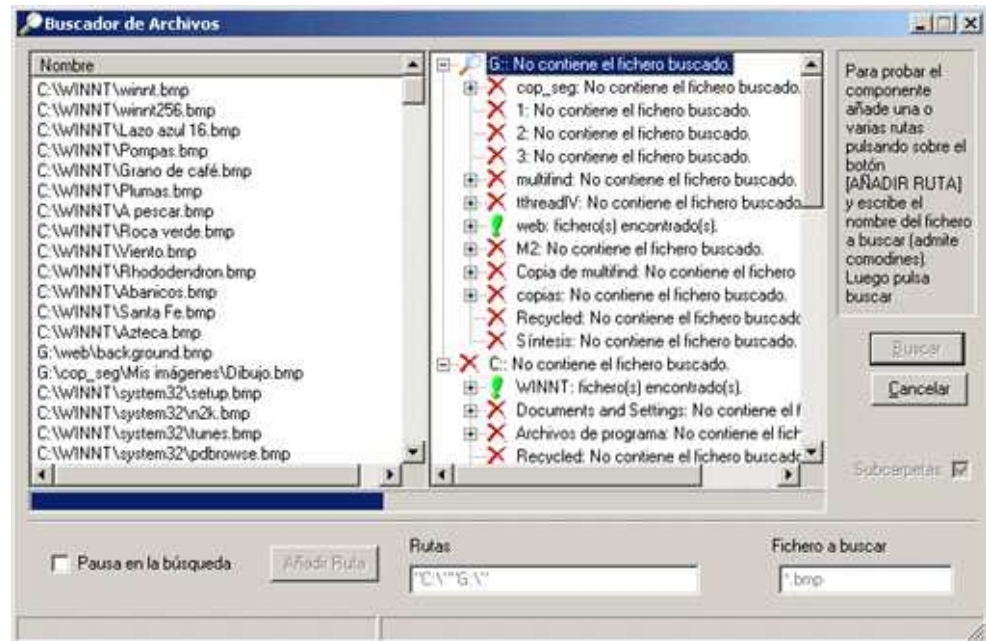


Figura 2. Vista del programa “Multifind.exe” en ejecución. Nuestro Buscador de Ficheros.

Parece también razonable que el nuevo componente pueda cambiar el color de fondo, para lo cual, tras definir la nueva propiedad Color, en el método de escritura nos aseguramos que el nuevo valor se transmita a los tres componentes visuales que representan el buscador.

Lo mejor es verlo:

```
procedure TBuscador.SetColor(const Value: TColor);
begin
  fColor:= Value;
  fArbol.Color:= Value;
  fResultados.Color:= Value;
  fResultados.UpdateColumns; // <--- Accedemos a un método protegido
end;
```

Lo único destacable es la llamada a UpdateColumns, método protegido en TCustomListView, por lo que justifica la postura que hemos adoptado. En lugar de crear un componente TListView, lo hacemos de un descendiente, con acceso a ese método.

Se puede ocultar el árbol, si fuera el caso de que deseáramos esto, y cuando cambia el tamaño de la ventana, se mantiene la relación entre los tres componentes internos. En la siguiente versión del componente, y que será abordada en un artículo posterior, veremos de dotarle con la capacidad de ser

modificada esta proporción, a voluntad del usuario, mediante el uso del ratón. Como lo haríamos con el componente TSplitter.

Nos limitamos a permitir tan solo a que el tamaño de la lista de resultados sea el total de la ventana o la mitad, dependiendo de la visibilidad o no del árbol. Es lo que hemos hecho tras redefinir el tratamiento del mensaje WM_SIZE.

```
procedure TBuscador.WMSize(var Message: TWMSize);
begin
  inherited;
  if fArbolVisible then begin
    fResultados.Height:= Height - fBarra.Height;
    fResultados.Width:= Width div 2;
    fBarra.Top:= Height - fBarra.Height;
    fBarra.Width:= Width;
    fArbol.Left:= FResultados.Width;
    fArbol.Height:= FResultados.Height;
    fArbol.Width:= Width div 2;
  end
  else begin
    fResultados.Height:= Height - fBarra.Height;
    fResultados.Width:= Width;
    fBarra.Top:= Height - fBarra.Height;
    fBarra.Width:= Width;
    fArbol.Left:= FResultados.Width;
    fArbol.Height:= FResultados.Height;
  end;
end;
```

Nuestra primera rutina para que nuestros ascendientes traten el mensaje: la llamada a *Inherited*. Posteriormente, y según sea visible el árbol, ajustamos las nuevas dimensiones. En este caso concreto, como decíamos, tan solo le pedimos al componente que divida su anchura (*Width*) y la comparta entre el árbol y la lista de resultados.

La barra de progreso quizás podría ser algo auxiliar y podría haber sido eliminada. Sin embargo, parece razonable que el usuario pueda necesitar saber en que punto de ejecución del algoritmo se halla, informando también en el caso de que la búsqueda se haya detenido inesperadamente.

Veamos el constructor con más detenimiento:

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

Inicia nuestro constructor la típica llamada al constructor en el ascendiente, para que se encadenen uno tras otro, cada uno de los constructores de los objetos y componentes que forman parte del actual.

```
  inherited Create(AOwner);    // invocación del constructor heredado
```

Nos saltamos el uso que hemos hecho de la variable *varBuscador*, que veremos con detalle al comentar el uso del gancho (*Hook*), y proseguimos con la creación de cada uno de los componentes principales, asignándoles el Parent, necesario para que se pueda visualizar sobre el formulario.

```
varBuscador:= Self;

fResultados:= TListaVista.Create(Self); // LISTA RESULTADOS
fResultados.Parent:= Self;
...
fBarra:= TProgressBar.Create(Self); // BARRA PROGRESO
fBarra.Parent:= Self;
...
fArbol:= TTreeView.Create(Self); // ARBOL
fArbol.Parent:= Self;
...
```

Para llegar a los dos objetos mas importantes del componentes y que se hacen necesarios para poner en funcionamiento y mantener la ejecución del algoritmo. Es el caso de la dos listas de punteros: *fListaNodos* y *fListaThreads*.

```
//la que contiene los nodos de busquedas pendientes
fListaNodos:= TList.Create;
//la que contiene las cadenas de busquedas pendientes
fListaRutas:= TStringList.Create;
//la que contiene las busquedas en ejecucion
fListaThreads := TList.Create;
```

La lista de cadenas *fListaRutas* resulta más superflua. El componente debería haber hecho uso de una propiedad de tipo TStrings para mantener y gestionar las peticiones del usuario: las rutas donde debe buscar el fichero. El haber hecho uso de esta lista de cadenas, de forma dinámica y haber accedido a ella mediante el procedimiento *Add()*, será motivo de revisión en la siguiente versión del componente, donde estudiaremos esto con mas detenimiento. En dicha versión, publicará una propiedad llamada *Cadena*, de tipo Strings. Todo depende del diseño que consideremos.

El resto de propiedades nos ayudan a mantener la consistencia del estado, así que tampoco nos detenemos demasiado.

```
fEstado:= esInactivo; // estado inactivo

fColor:= clWindow; // color clWindow
fArbolVisible:= True; // arbol visible
fSubcarpetas:= True; // queremos que se busque en subcarpetas
end;
```

Pero volvamos a la listas de punteros: ¿cual es la misión de ambas?

Cuando el usuario establece una petición de búsqueda de varias rutas, tal que *"C:\\"C:\Archivos de Programa\"D:\"*, el buscador recibe mediante *Add()* cada una de ellas individualmente, y las guarda en la lista de cadenas. Eso lo podemos ver en el método *Execute* de la clase *TBuscador*, donde, tras comprobar que el componente cumple la Precondición exigida por el algoritmo de Búsqueda, inicia la extracción de cada una de estas rutas para crear y lanzar las búsquedas correspondientes:

```
procedure TBuscador.Execute;
```

```
begin
    DoExecute;
end;

procedure TBuscador.DoExecute;
var
    varBusqueda: TBusqueda;
    s: String;
begin
    fMsecs := 0; // ponemos a cero el contador de tiempo
    fMsecs := GetTickCount;
    fEstado:= esEjecucion; // iniciamos el estado de ejecución
    fArbol.Items.Clear; // vaciamos el arbol visual
    fResultados.Items.Clear; // vaciamos la lista de resultados
    // PRECONDICION DEL ALGORITMO
    // Abortamos lanzamiento en caso de errores
    ...
```

A partir de esta línea, y no considerando las rutinas que gestionan la detección del error, iniciaremos un bucle para crear un objeto THiloBusqueda por cada una de las peticiones, vaciando uno a uno y desde el primer ítem, cada una de las rutas, a medida que van siendo lanzados.

```
// proceso de creación de los objetos Hilos (THiloBusqueda)
// para cada una de las rutas elegidas por el usuario
While FListaRutas.Count > 0 do begin
    s:= fListaRutas[0]; // almacenamos en [s] la primera ruta
    fListaRutas.Delete(0); // ya podemos borrarla y se resituaran de nuevo
    fBarra.Max:= FBarra.Max + 1; //añadimos al valor máximo
    fBarra.Position:= fBarra.Position + 1; // y la posición de la barra
    if s[length(s)] <> '\' then s:= s + '\'; // nos aseguramos que es correcto el final en
'\
    // creación del hilo
    varBusqueda := TBusqueda.Create(Self, nil, s + FToken, FSubcarpetas);
    // y lo añadimos a la lista de hilos
    fListaThreads.Add(varBusqueda);
    varBusqueda.Resume; // y ya podemos lanzar su ejecución
end; // el proceso se repite mientras queden rutas

// finalmente activamos el gancho solo si no está instalado
if WinHook = 0 then
    WinHook:= SetWindowsHookEx(WH_CALLWNDPROC, @WHCALLWNDPROC, 0, GetCurrentThreadId);
end;
```

Tenemos así, ya en ejecución un solo hilo para cada una de las carpetas en las que ha de buscar coincidencias, y el contador -en este momento- de *fListaThreads*, coincidirá con el número de hilos creados. La variable *fListaNodos* todavía no ha participado. Su contador vale cero (0).

Si razonamos un poco en como va a funcionar nuestro algoritmo, dado que no queremos que el nuevo hilo cree otro número indeterminado de hilos, que a su vez hagan lo mismo, en lugar de indicarle que proceda a crear un nuevo THiloBusqueda por cada una de las carpetas encontradas, le diremos:

- Cuando encuentres una nueva carpeta que sea factible de ser motivo de búsqueda, crea un nodo (una estructura TNode), y almacena en ellas todos los parámetros necesarios para crear una nueva hilo, los mismo que necesitó TBuscador para crear al primer o primeros hilos THiloBusqueda. Una vez almacenado los valores en el nodo, insértalo en la lista a tal efecto (*fListaNodos*). Si haces esto con cada una de las carpetas encontradas, tan solo te restará comunicar al buscador si existe el fichero en ese nivel (2ª fase de búsqueda que mencionábamos anteriormente), y antes de destruirte, extrae el primer nodo de la lista de nodos y crea con el un nuevo THiloBusqueda que prosiga tu trabajo... En ese punto, el objeto THiloBusqueda puede dormir en paz, por no decir suicidarse apaciblemente y con la conciencia tranquila.

¿Veis?. Esto supone, que por cada una de las rutas en las que ha de buscar el algoritmo, cada vez que nace un objeto THiloBusqueda es porque otro esta destruyéndose, y nunca convivirían más del número de peticiones del usuario, al menos ejecutándose.

Nacer, crecer y morir...

Todo bicho viviente que nace, irremediamente ha de morir... Desgraciadamente también les pasa a nuestros componentes, a pesar del cariño que hallamos puesto en un momento de su ejecución. Con todo el pesar que esto supone, nuestra misión como padres del mismo, es que en esa marcha apresurada hacia el otro mundo, deje pagadas todas sus deudas: Si le pidió prestada memoria al sistema operativo para poner en marcha un gran proyecto, debe devolverla, como nos pasa a nosotros con cualquiera de las entidades bancarias al solicitar un préstamo.

Vamos a ver en el destructor, el ajuste de cuentas referido:

```
destructor TBuscador.Destroy;  
var  
    i: Integer;  
    tmp_Nodo: PNode;  
begin
```

Volvemos a posponer el tema de la instalación del objeto gancho (Hook), para no perder el hilo narrativo. Y nos fijamos en la línea posterior, cuando asignamos el estado del componente a inactivo:

```
if WinHook <> 0 then UnHookWindowsHookEx(WinHook);
```

```
fEstado:= esInactivo;
```

¿Y algunos compañeros se preguntarán, qué sentido puede tener hacer tal asignación? A fin de cuentas, podemos liberar la lista de rutas, como se ve en las líneas siguientes, podemos liberar la lista de nodos, pendientes de ejecución, y la memoria reservada a cada uno de ellos, y podemos liberar la lista de Hilos en ejecución... pero, ¿qué sucede con los que han cambiado de contexto justo en el momento en el

hilo principal se ha activado para cerrarse? Razonémoslo unas líneas más abajo, suponiendo que se ha liberado todos los objetos que habían sido creados dinámicamente:

```
if Assigned(fListaRutas) then begin
  fListaRutas.Clear;
  fListaRutas.Free;
end;

fImage:= nil;    // desasignamos la imagen asociada

//de los nodos y de la lista de nodos
for i := FListaNodos.Count-1 downto 0 do begin
  tmp_Nodo:= PNode(FListaNodos.Items[i]);
  Dispose(tmp_Nodo);
  fListaNodos.Items[i] := Nil;
end;
fListaNodos.Clear;
fListaNodos.Free;
fListaNodos:= Nil;

//de los búsquedas en ejecución (hilos), eliminando la lista
if fListaThreads <> nil then begin
  for i := fListaThreads.Count - 1 downto 0 do begin
    TThread(fListaThreads[i]).Terminate;
    fListaThreads[i] := Nil;
  end;
  fListaThreads.Clear;
end;
fListaThreads.Free;
fListaThreads:= Nil;

// destruimos los objetos y componentes creados dinámicamente
fBarra.Free;
fArbol.Free;
fResultados.Free;
inherited Destroy; // invocacion del destructor en el ascendente
end;
```

Vale. Ya estamos situados. En este punto del programa, puede haber un `THiloBusqueda` dormido, bien esperando entrar en el método `Execute`, y en ese caso la condición `Terminated`, que se activo cuando se invocó el destructor en el ascendente `inherited Destroy`, le podría disuadir de acceder a dicho método, cosa realmente mala para nosotros, ya que no le deberíamos permitir el acceso a los objetos relacionados con hilo principal, los cuales podría ser que ya no estuvieran disponibles. Pero también puede darse el caso de que este esperando iniciar el procedimiento asociado a la destrucción del hilo, en cuyo caso la condición `Terminated` ya no nos vale, puesto que todos los hilos, pasan por ese punto en cualquier momento de la ejecución del programa, y en nuestro caso, sí accedemos desde allí a los objetos relacionados con el form principal (los nodos del árbol necesitan en último lugar una ventana donde pintarse).

Así que nos valemos del estado del buscador para comunicar al hilo que va a finalizar, y que va a ejecutar el método *OnEnd*, que ha finalizado su tarea y que debe de salir del procedimiento y yacer en paz.

Y seguía fallando el maldito...

El razonamiento era redondo, perfecto... Una buena amiga mía, hace ya tiempo, me decía algo parecido. Decía que era perfecta... pero le pasaba algo parecido a lo que a este desagradecido algoritmo: no tenía memoria. Se olvidaba nuestro buscador, o por lo menos, los hilos que creaba, de que hacia pasado ya mucho tiempo desde que el form había sido limpiado, y seguían obstinados por acceder a los nodos del árbol.

Para comprobarlo, debéis limpiar de los módulos todo lo que haga referencia a los Hooks (ganchos) o por lo menos la instalación del gancho. Luego ejecutáis desde Delphi, paso por paso, el proyecto "*multifin.dpr*" con algunos puntos de parada, a partir del momento en que se cierra la ventana. Si evaluamos el valor de *Terminated*, en los momentos anteriores a la excepción, es *false*. Ese parece ser el problema. Me arriesgo al suponer, aunque solo sea por desperdiciar un poco de imaginación, que algunos hilos quedan atrapados en el interior del método *Execute*, en el cambio de contexto al hilo principal y desconocen al despertar que algunas de las referencias a las que van a acceder ya no son válidas. Supongo que es hablar por hablar...

La depuración de este tipo de problemas no resulta sencilla. Os comentaba al inicio del artículo que había perdido muchas horas en intentar evitar este problema. Abstraído y aburrido, en una pantalla que desfilaba línea a línea, variable a variable, hasta intentar localizar el problema.

¿Que hice entonces? Lo mejor... cerrar la ventana del entorno y abrir el navegador en busca de aire fresco, con la idea de encontrar en otro código fuente alguna idea.

En el artículo anterior, hacía referencia a unos ejemplos que encontré sobre Hilos en la sede de Borland. En los ejemplos que visualicé, destruían la ejecución de los hilos en el evento *OnClose* del formulario, antes de que se cerrase la ventana. Así que valoré esa posibilidad, con la confianza de que el método *Cancel* del buscador sí funcionaba correctamente. Y ahí entran a jugar la instalación y desinstalación del gancho. Con él, capturo el evento *OnClose*, antes de que sea tratado por la ventana principal y le ordeno al buscador que cancele la ejecución mediante *Cancel*. De esa forma, la destrucción de los hilos se debía producir correctamente y sin la generación de excepciones. Y parecer ser que funciona.

Jugar a Peter Pan...

A pesar de que el tema de las funciones gancho, nombre mas correcto que el empleado por nosotros hasta ahora, es un buen candidato para iniciar una próxima serie, avanzaremos que este tipo de funciones nos permiten monitorizar acontecimientos esperados, interceptar mensajes para darles un tratamiento especial a pesar de no tener la invitación de la sufrida ventana... Existen funciones gancho para interceptar los mensajes del teclado, las notificaciones de la Shell, los del ratón, etc... dependiendo del tipo de garfio (Hook) instalado.

En nuestro caso, hemos hecho uso de un gancho de tipo `WH_CALLWNDPROC`, que nos permitirá interceptar los mensajes pertenecientes a nuestro propio proceso, antes de que sean enviados al procedimiento de ventana de destino. Recordar que queremos capturar el mensaje `WM_CLOSE`.

Podemos ver el código:

```
function WHCALLWNDPROC(nCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
var
  Estructura: ^TCWPStruct;
begin
  if nCode > -1 then begin //Si nos corresponde procesar la información
    if wParam = 0 then begin // Si el mensaje proviene de este proceso
      Estructura:= Pointer(lParam); //¿se cierra la ventana?
      if Estructura.message = WM_CLOSE then varBuscador.DoCancel; //cancelamos
    end;
    Result:= 0; // No procesamos el mensaje (valor siempre 0)
  end //si no nos corresponde procesar el mensaje, que otro gancho lo intente
  else Result:= CallNextHookEx(WinHook, nCode, wParam, lParam);
end;
```

Los comentarios adicionales son suficientemente explicativos como para necesitar un comentario adicional. La instalación es sencilla. Hacemos uso de la función `SetWindowsHookEx()`

```
WinHook:= SetWindowsHookEx(WH_CALLWNDPROC, @WHCALLWNDPROC, 0, GetCurrentThreadId);
```

Donde `WH_CALLWNDPROC` es el tipo de gancho utilizado, `@WHCALLWNDPROC` es un puntero a la función gancho, 0 es el parámetro *hmod*, (el handle al módulo dll que podría contener la función gancho y que por lo tanto hace referencia al proceso actual, y por último, `GetCurrentThreadId`, que identifica al hilo actual.

La desinstalación es si cabe mucho más sencilla. La función `UnHookWindowsHookEx()` recibirá como parámetro el *handle* obtenido por la primera.

```
if WinHook <> 0 then UnHookWindowsHookEx(WinHook);
```

En este caso, y creo que razonablemente, he elegido para la instalación de la función gancho, el procedimiento de ejecución del algoritmo de búsqueda, *Execute*. Cosa distinta, y que valoraremos posteriormente, es si existe algún procedimiento mejor que el uso de una función gancho. De momento, nos valdrá para comprobar que efectivamente los problemas vienen ocasionados por la destrucción de la ventana y para que al menos, nuestro componente empiece a ser funcional.

Cancelar y mas cosas pendientes.

Del método *Cancel* y *DoCancel*, hay poco que comentar. Contiene prácticamente las mismas rutinas que *Destroy*, con la salvedad, dado que el buscador no va a ser destruido, no se liberan las listas de punteros y nos limitamos a dejar el componente en condiciones de iniciar una nueva búsqueda. Se inicializan los valores de la barra de progreso, el estado del buscador se desactiva, y se elimina cualquier tarea pendiente, como las listas de nodos, destruyendo la memoria asociada.

```
procedure TBuscador.DoCancel;
var
  i: Integer;
  tmp_Nodo: PNode;
begin
  fEstado:= esInactivo; //inicializamos el estado
  // inicializamos la barra de progreso
  fBarra.Max:= 0;
  fBarra.Position:= 0;
  // inicializamos la lista de nodos. Deben ser eliminados todos los nodos
  for i := FListaNodos.Count-1 downto 0 do begin
    tmp_Nodo:= PNode(FListaNodos.Items[i]);
    Dispose(tmp_Nodo);
    FListaNodos.Items[i] := Nil;
  end;
  fListaNodos.Clear;
  // inicializamos la lista de Hilos. Deben terminar
  for i := fListaThreads.Count - 1 downto 0 do begin
    TThread(fListaThreads[i]).Terminate;
    fListaThreads[i] := Nil;
  end;
  fListaThreads.Clear;
  // inicializamos la lista de rutas
  if Assigned(FListaRutas) then begin
    FListaRutas.Clear;
  end;
end;
```

Pero si parece llamativo que lo intentemos hacer como vimos en anteriores artículos. Declarando un método protegido y virtual, que pueda ser redeclarado en un descendiente, y ser usado internamente por el buscador. Y el método público *Cancel*, envuelve al método protegido, añadiendo condiciones especiales que no son requeridas en el uso interno del continente, aunque sí para las llamadas externas al mismo.

```
procedure TBuscador.Cancel;
begin
  if fEstado = esInactivo then exit;
  DoCancel;
  if Assigned(FOnAbort) then FOnAbort(Self, inCancelacion);
end;
```

En este caso concreto, el procedimiento para cancelar lleva aparejado el evento *FOnAbort*, facilitando al usuario la implementación del mismo y la respuesta del interfaz gráfico al acontecimiento comentado.

Con *Execute* y *DoExecute* también hemos hecho esa deferencia, en la espera de que en posteriores artículos diferenciamos el uso de ambos métodos.

```

const

BA_INCPOS = WM_USER + 1000; // Incrementa valor POSICION de Barra
BA_INCMAX = WM_USER + 1001; // Incrementa valor MAXIMO de Barra

RE_NUMITEM = WM_USER + 1002; // Añade un nuevo TListItem a los resultados
RE_SELECT = WM_USER + 1003; // Selecciona TListItem de los resultados
RE_ADD = WM_USER + 1004; // Añade un nuevo hilo a la lista de Hilos
RE_DEL = WM_USER + 1005; // Elimina un hilo de la lista de hilos

AR_ADD = WM_USER + 1006; // Añade un objeto TTreeNode al Árbol

BU_ABORT = WM_USER + 1007; // Se ha producido un error o cancelación
BU_END = WM_USER + 1009; // Búsqueda ha acabado sin incidencias

```

Listado 2. Mensajes de comunicación entre THiloBusqueda y TBuscador.

Pero vamos a terminar con los dos métodos virtuales que nos restan y que serán motivo de comentario en nuestro próximo artículo, en donde acabaremos de ver y explicar el objeto THiloBusqueda y la comunicación que se produce entre este y el componente TBuscador. Adelantamos el **listado 2**, en donde podéis observar los mensajes que hemos definido y el **listado 3**, con la definición de tipos de la interfaz del objeto descendiente de TThread.

```

type

TBusqueda = class(TThread)
private
  fNuevaBusqueda: PNode; // puntero a estructura de nodo
  fNodoPadre: TTreeNode; // nodo padre de nuestro nodo actual
  fBuscador: TBuscador; // componente buscador que inicia la ejecución
  fRuta: string; // ruta a buscar
  fCarpetaHilo: string; // ruta (variable auxiliar)
  fSubcarpetas: boolean; // ¿hay subcarpetas?
  fNodo: TTreeNode; // apunta al nodo del arbol de resultados actual
  fResultado: boolean; // ¿hay coincidencia en la búsqueda?
  //¿Se ha encontrado archivos en esa carpeta
  procedure OnEnd(Sender: TObject); // al finalizar su ejecución
public
  constructor Create(const ABuscador: TBuscador; const ANodoPadre: TTreeNode;
    const ARuta: string; const ASubcarpetas: boolean);
    reintroduce; overload;
  procedure Execute; override; // Método execute de la clase TThread
end;

var
  FCriticalLista: TRTLCriticalSection;

implementation

```

Listado 3. Definición del objeto THiloBusqueda, descendiente de la clase TThread.

El procedimiento *AddThread()* recibe como parámetro el objeto hilo que lo invoca. Es llamado por la instancia del hilo desde la referencia que guarda al componente TBuscador, pero dado que éste es protegido, y no puede ser llamado directamente desde THiloBusqueda, encontraremos en los mensajes una oportunidad de invocarlo. Esto será motivo de nuestro artículo posterior pero ya desde este, parece necesario anticiparlo.

Los mensajes *RE_ADD* y *RE_DEL*, serán el punto de encuentro entre ambos. El primero para que el buscador añada la referencia pasada por el parámetro a la lista de hilos y el segundo para que la elimine de la misma. Es como si el objeto THiloBusqueda le estuviera diciendo al componente TBuscador: Oye, majo, soy yo, añádeme a la lista...

```
procedure TBuscador.AddThread( Hilo: TThread );
begin
    //añadimos el nuevo hilo
    if fListaThreads.IndexOf(Hilo) = -1 then fListaThreads.Add(Hilo);
end;
```

Por el contrario, al finalizar la vida del hilo y sintiendo su muerte cercana, no tiene otro deseo que despedirse: Oye... que me voy... acuérdate de eliminarme de la lista.

```
procedure TBuscador.DeleteThread(Hilo: TThread);
var
    SinHilos: boolean;
    ind:      integer;
begin
    ind := fListaThreads.IndexOf(Hilo); // indice de la búsqueda
    if ind <> -1 then fListaThreads.Delete(ind); // me elimino de la lista
    ...
end;
```

Y es precisamente en este procedimiento, cuando se puede evaluar la condición de finalización del algoritmo de búsqueda: que no queden hilos en la lista y que la lista de nodos pendientes este vacía. Si esto es así,

```
SinHilos := (fListaThreads.Count = 0) and (fListaNodos.Count = 0);
// si ya no quedan hilos, lanzo el evento en cuestión
if SinHilos then begin
```

Intentaremos dejar el componente buscador en condiciones de iniciar una nueva búsqueda, comunicando al interfaz gráfico el evento que le permite saber que ha acabado el algoritmo con éxito.

```
fEstado:= esInactivo; //inicializamos los estados
fBarra.Max:= 0;
fBarra.Position:= 0;
fMSEcs := GetTickCount - fMSEcs;
fArbol.Items.BeginUpdate;
fArbol.FullExpand; // desplegamos el arbol
fArbol.Items.EndUpdate;
if Assigned(FOnEnd) then FOnEnd(Self); // lanzamos el evento OnEnd
```

```

//que le da al usuario de inicializar la interfaz del formulario ante
// la finalización del algoritmo
// CUADRO EMERGENTE DE TOMA DE TIEMPOS -ACCESORIO-
MessageBox(GetActiveWindow, PChar(FormatFloat('#,0', fResultados.Items.Count) + '
coincidencias en un total de '+
                                FormatFloat('#,0', fArbol.Items.Count) + ' carpetas.' +
#13 + 'Tiempo empleado: ' +
                                FormatFloat('#,0', fMsecs) + ' milisegundos.'),
                                'Atención', MB_ICONINFORMATION);

end;
end;

```

Por último el procedimiento *Pause*, que nos permitirá Suspendar o Relanzar la ejecución de los hilos. Estos conceptos eran explicados ampliamente en los últimos artículos y tampoco merecerán que nos detengamos demasiado. A continuación os incluyo el método *Pause*:

```

procedure TBuscador.Pause(parar: boolean);
var
  i: integer;
begin
  if parar then
    for i := fListaThreads.Count - 1 downto 0 do
      TThread(fListaThreads[i]).Suspend
    else
      for i := fListaThreads.Count - 1 downto 0 do
        TThread(fListaThreads[i]).Resume;
      end;
end;

```

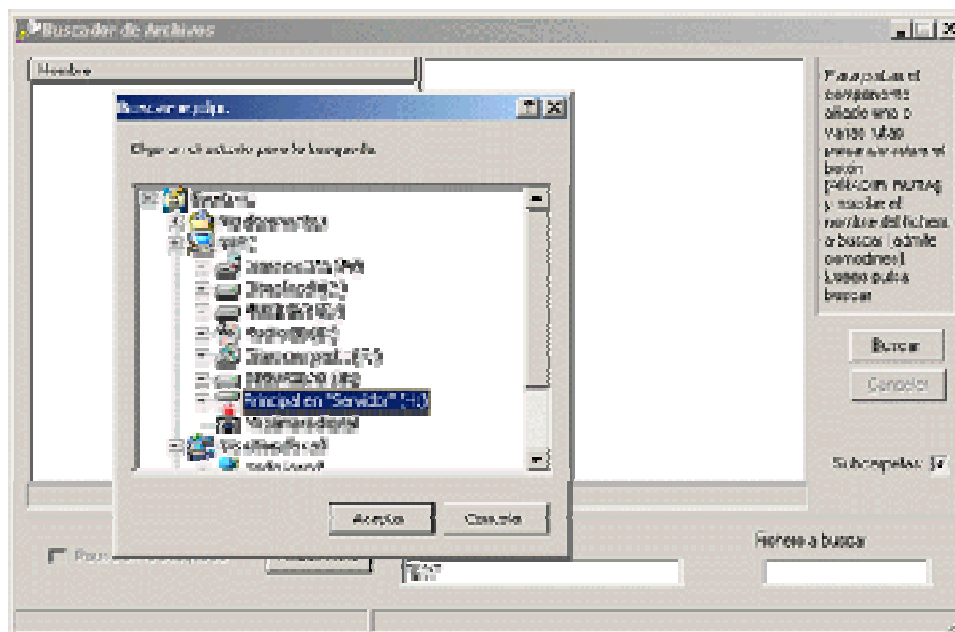


Figura 3. Cuadro de dialogo de selección de carpeta: SHBrowseForFolder.

Y sobre el módulo principal, ¿nada que contar?

¡Sabía yo que olvidaba alguna cosa! Me lo había dejado apuntado en algún papel sobre la mesa del escritorio y cuando ya andas pensando en cerrar el artículo con los últimos comentarios, entonces va y aparece el recordatorio... ¡En buenas horas!

Observad la **figura 3**, que corresponde a la pulsación del botón, mediante el que añadimos una nueva ruta al buscador. Como programadores nos debemos también al usuario, y nuestro interés debería ser que éste se sintiera feliz con nuestra aplicación: que fuera fácil de usar, intuitiva, que no le diera demasiados quebraderos de cabeza y la necesidad de adoptar decisiones que pudieran no ser de su incumbencia. Y en ese intento de facilitarle las cosas, incluiremos un cuadro de diálogo para que pueda seleccionar una carpeta, sin que deba recordar la ruta completa o correcta.

El procedimiento es el siguiente. Vamos a comentar lo más significativo:

```
procedure TMainForm.btn_AnadirRutaClick(Sender: TObject);
var
  ItemIDList      : PItemIDList;
  Registro_Info: TBrowseInfo;
  Ruta           : array[0..MAX_PATH] of char;
begin
```

Para empezar, debemos preparar un registro `TBrowseInfo`, que nos permitirá entregar un puntero como parámetro de la función `SHBrowseForFolder()`. La información usada por este registro será utilizada para preparar el cuadro de diálogo para la selección de una carpeta. Este registro contiene información como la raíz desde la que debe partir la visualización del árbol de directorios, un puntero a la matriz de caracteres de la carpeta elegida por el usuario, si es que la elige, y algunas cosas más: el título de la ventana, qué tipo de carpetas debe visualizar y un puntero a una función, que en este caso podría haber sido omitido.

```
FillChar(Registro_Info, SizeOf(Registro_Info), 0);
with Registro_Info do begin
  hwndOwner      := MainForm.Handle; // Ventana propietaria del cuadro de diálogo
  pidlRoot       := nil;             // raíz del arbol
  pszDisplayName := Ruta;            // selección del usuario
  lpszTitle      := 'Elige un directorio para la búsqueda'; // Titulo
  // ORDENADORES (Pc) y DIRECTORIOS DEL SISTEMA
  ulFlags        := BIF_BROWSEFORCOMPUTER or BIF_RETURNONLYFSDIRS;
  lpfn           := @MiFunRetorno; // podría hacer sido NIL
  lParam         := 0;
end;
```

Toda vez que tenemos listo nuestro registro e invocamos la función, obtenemos en `ItemIDList`, un puntero a una lista de elementos o bien, en el caso de que no tenga éxito la selección o bien sea cancelada por el usuario, el valor indefinido `nil`, que nos indica el error.

```
ItemIDList := SHBrowseForFolder(Registro_Info); // Ejecución del cuadro de diálogo
```

Nos falta extraer la ruta completa de la carpeta elegida para lo que hacemos uso de la función `SHGetPathFromIDList()`, que nos devuelve en la referencia del segundo parámetro dicho valor, listo para ser utilizado por nosotros.

```

SHGetPathFromIDList(ItemIDList, @Ruta); // Tras ser cerrado, extraemos la ruta
// y la mostramos en el cuadro de edición encerrada entre comillas
if ItemIDList <> nil then edi_Ruta.Text := edi_Ruta.Text + ''' + Ruta + ''';
end;

```

En algunos ejemplos que he observado en la documentación consultada, la función de retorno es utilizada para enviar mensajes a la ventana donde emerge el cuadro de diálogo. Nosotros no vamos a hacer uso de esto, aunque he dejado la función para que se pudiera ver y comentar.

```

function MiFunRetorno(hWnd: HWND; uMsg: UINT; lParam: LPARAM; lpData: LPARAM): Integer;
begin
  Result:= 0;
end;

```

Los últimos comentarios y nos vamos...

La **figura 4** es similar a la número 3. En este caso, hemos montado el cuadro de diálogo (solo visualmente) haciendo uso del componente TShellTreeView, que incluye Delphi 6 en la pestaña *Samples*. Presenta algunas mejoras con respecto al que nosotros hemos utilizado, como por ejemplo que nos permitiría muy fácilmente activar la capacidad para seleccionar múltiples archivos. Esto podría hacerse creando un descendiente de TCustomTreeView y publicando la propiedad *MultiSelect*. Así de sencillo.

No obstante, la necesidad de que este ejemplo fuera compatible con Delphi 5 nos ha llevado a elegir el primero, dado que el componente citado solo se incluye en la pestaña de *Samples* de Delphi 6 y no de la versión 5. Delphi 5 cuenta con un “intento” poco vistoso y que deja mucho que desear. Parece preferible el cuadro de dialogo de Windows desde la función que hemos utilizado.

Quedan pocos comentarios que hacer. Hace tan solo un día escribía un correo a Jose Manuel en donde le comentaba la posibilidad de retomar el componente, para seguir añadiéndole correcciones a los problemas observados, a los detalles que se pueden mejorar, y en definitiva, invitándole a compartirlo de nuevo conmigo. En principio, nos queda un artículo más, para acabar de explicar y de ver el código del objeto hilo, aunque por la extensión de estas líneas, me pienso que poco comentario va a necesitar salvo algún detalle concreto y que pueda pasar desapercibido.

Mas detalles de aspectos que no funcionan todavía: La función gancho hace uso de una variable global para invocar el método *Cancel*. ¿Podrían convivir dos buscadores en la misma ventana, esperando el cierre de la misma? Se ve claro que no. Se cancelaría la ejecución del segundo buscador creado, pero la del primero seguiría generando una excepción. Es algo que tenemos que revisar sin duda.

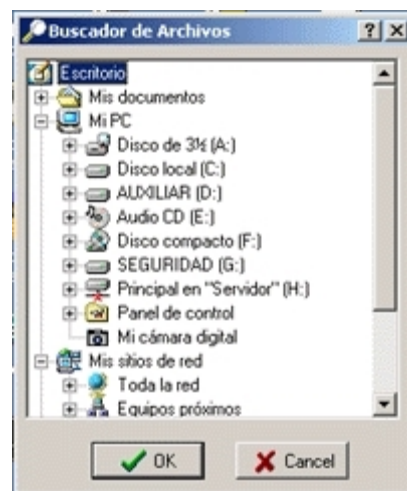


Figura 4. Posible cuadro de diálogo para la selección de una carpeta. Mediante TShellTreeView en Delphi 6.

Por otro lado, el buscador sigue presentando algunos problemas que intentaremos solucionar: si te has dado cuenta, la función gancho funciona correctamente (al menos eso parece), pero en todo esto existe un detalle que se nos escapa: necesita tener como *Parent* a la ventana principal para funcionar sin generar la excepción al cerrar. Eso significa que estamos condicionando el uso de nuestro componente e impidiendo que este pueda ser usado dentro de otro contenedor que no sea un TForm, como un panel por ejemplo. ¿Cual es la solución correcta? ¿Impedir que pueda ser soltado sobre un componente que no sea la ventana principal? ¿Averiguar la clase del componente sobre el que va a ser soltado en tiempo de diseño? ¿Intentar que funcione nuestra función gancho, independientemente del componente que acepte como padre? Sinceramente, ahora no lo se y es demasiado tarde para acabar de discurrirlo.

Acabo de recibir el correo de Jose Manuel. Seguiremos trabajando sobre este componente. Recalco de sus líneas algunos de sus comentarios que me parecen muy oportunos: el primero en la línea de resaltar la ganancia en el rendimiento de trabajar con varias líneas de ejecución, en una búsqueda en unidades de Red. Nuestro componente, o nuestro planteamiento es claramente superior, que planificar una búsqueda secuencial en unidades conectadas. Algo bueno teníamos que tener, ¿no? Tened en cuenta que nos ayuda también la máquina a la que nos conectamos, brindándonos gentilmente su procesador para buscar en su unidad mientras nosotros seguimos la búsqueda en la nuestra.

También comenta algo, que dejaba para el próximo capítulo, y es que, nuestro diseño, no se haya libre de hacer mejoras importantes, y que nos ayuden a mejorar la eficiencia y la calidad del código final del componente. Jose Manuel destaca, cómo podemos llegar un poco más lejos en el diseño del algoritmo buscador e intentar convertirlo en esa bombilla que se cambia en un abrir y cerrar de ojos. Vale... vale... No hemos llegado a eso todavía. Tenemos que introducir el uso de clases abstractas, que tan solo declaran “intenciones” pero que nunca mueven un dedo para hacer algo. Son sus descendientes los que lo hacen.

Jose pone un ejemplo, bastante intuitivo para entenderlo:

```
TBusquedaAbstracto = class(TThread)
    ...
end;

TBusquedaBinario = class(TBusquedaAbstracta)
protected
    procedure Execute; override; // aquí se implementa
end;

TBusquedaSecuencial = class(TBusquedaAbstracta)
protected
    procedure Execute; override; // aquí se implementa
end;

TBuscador = class(...)
private
    // aquí cabe cualquier tipo de búsqueda
    FBusqueda: TBusquedaAbstracto;
end;
```

TThread V: Un buscador de Archivos (I)

Salvador Jover
s.jover@wanadoo.es

Es decir, que empezamos a meternos en polimorfismo, un tema necesario y vital comprender en la construcción de objetos. Lo veremos.

Otra cosa: la eficiencia del algoritmo también es factible de mejora, si revisamos la secuencia de los bucles. Posiblemente nos sea posible eliminar uno de ellos, y que en el primero de los recorridos queden resueltas las dos misiones encomendadas: la de añadir nodos a la lista y la de resolver las coincidencias de los ficheros con el Token.

Me resta despedirme de vosotros con el deseo de que haya sido de vuestro agrado esta lectura y este rato que compartimos...

En el próximo artículo seguimos conversando. Recibid un saludo.

Salvador Jover s.jover@wanadoo.es

Agradecimientos: Desde estas líneas doy las gracias a Jose Manuel Navarro por la colaboración. Nos vemos en el próximo.