Los rincones del API Win32
Archivos proyectados en memoria

Vamos a profundizar en el último aspecto que nos falta sobre la gestión de memoria en Win32: los archivos proyectados en memoria.

Introducción

Durante los últimos artículos hemos ido explicando con todo detalle cómo se gestionan los recursos de memoria en la arquitectura Win32: memoria virtual, pila y montones. En las explicaciones (más o menos claras), hemos abordado tanto los aspectos generales, aplicables a cualquier entorno y arquitectura, como las características propias de Win32.

En esta ocasión vamos a tratar los archivos proyectados en memoria, un aspecto, que aunque específico de la plataforma Win32, también se puede aplicar a otras arquitecturas (de hecho UNIX y Linux los utilizan).

¿Qué son esas cosas?

Los archivos proyectados en memoria (del inglés *"memory-mapped files"*), son un tipo de archivo especial que se basan en la capacidad de la memoria virtual para utilizar espacio físico en disco como si fueran páginas de memoria RAM.

Básicamente, se trata de almacenar datos en memoria, como variables, registros, buffers, o cualquier otra estructura de datos, pero obligar al sistema a que proyecte esas páginas de memoria en un archivo concreto, en vez de utilizar el archivo de paginación de sistema. Después, al acceder a esas páginas de memoria, en realidad estaremos accediendo a un espacio en el disco duro, correspondiente al archivo que hemos proyectado.

La teoría

Cuando hablamos de la memoria virtual (http://www.lawebdejm.com/?id=22101), dijimos que el espacio de direcciones virtuales se estructuraba en un árbol formada por *"Page Directory"* - *"Page Tables"* - *"Page Frames"*. Toda esta estructura lo representábamos en una tabla lineal para no complicar más el asunto. En esta tabla (nuestro *"Page Directory"*) se indicaba tanto el número de cada una de las páginas (en total 1.048.576 páginas de 4 KB cada una), como la dirección virtual que le corresponde a cada una de ellas, y la dirección física donde está proyectada (que se calculaba a partir de la dirección virtual) . Lo más habitual era que una página de memoria virtual, se proyectase sobre el archivo de intercambio del sistema (normalmente "Win386.swp" en sistemas Windows 95 y "pagefile.sys" en sistemas Windows NT/2000).

Como ya vimos, cuando un programa necesita leer datos de una página que no estaba en RAM, se produce un fallo de página (*page fault o page interrupt*), lo cual desencadena un proceso en el que

se accede a los datos dentro del archivo de intercambio, y se vuelcan a memoria RAM física. Después de esto, se puede continuar la ejecución de modo transparente al usuario.

Ahora supongamos que le decimos al sistema que ciertas páginas de memoria, sean almacenadas en un archivo específico (por ejemplo "c:\datos.dat") en vez de utilizar el archivo de intercambio de sistema. El proceso sería el mismo, con la única diferencia del lugar donde se almacenan las páginas.

Así podríamos manipular cualquier estructura de memoria (un array, una pila, una cola, registros, etc.) a través de punteros, pero en realidad estaríamos leyendo y grabando datos en un archivo de disco.

Con este sistema conseguiríamos olvidarnos del antiguo método de lectura de ficheros (apertura, lectura secuencial o aleatoria y cierre), más propio de la década de los 80 y la programación en COBOL que los nuevos lenguajes de programación del siglo XXI.

Pues, gracias a los archivos proyectados en memoria, esto es posible. Con ellos, podemos manipular un archivo pensando que está cargado en memoria al completo, y simplemente con un puntero accederemos a donde queramos: al principio, al final, desde atrás, hacia adelante... el sistema será el responsable de cargar las páginas que sean necesarias cuando accedemos a ellas.

Cuándo utilizarlos

Básicamente, los archivos proyectados en memoria se utilizan en 3 tareas:

1. Leer los archivos contenidos en los archivos .EXE y .DLL cuando se ejecuta un proceso.
2. Realizar operaciones de E/S a disco sin buffers de memoria intermedios.
3. Crear zonas de memoria compartida, para intercambiar datos entre procesos.

Vamos a explicar cada uno de estos usos:

Lectura de archivos ejecutables

Supongo que alguna vez os habréis preocupado viendo que vuestro ejecutable ocupa demasiado espacio en disco, pensando que cuanto mayor sea el ejecutable, más lenta será la carga, incluso quizá alguien ha utilizado uno de esos compresores de ejecutables, que hacen que el programa "parezca" más pequeño.

En realidad, esto no es así, *un ejecutable no carga más lento por ser más grande*, ya que los archivos EXE (y librerías DLL) se cargan a través de archivos proyectados en memoria.

Vamos a explicarlo detalladamente con un ejemplo. Supongamos que hemos creado un ejecutable que ocupa en disco 400 KB. Cuando hagamos doble clic sobre él (es decir, cuando lo ejecutemos), ocurrirá lo siguiente:

1. Se abre el fichero a través de la función del API CreateFile.
2. Se leen los datos de la cabecera del ejecutable, como referencias a funciones en DLLs externas, posición en el espacio de memoria donde debe comenzar la imagen, creación de variables estáticas (las que son globales, que no están definidas dentro de ninguna función o clase), etc. Todas estas tareas se hacen a través de la funciones de la librería "imagehlp" (dbghelp.dll), tales como ImageDirectoryEntryToData, ImageLoad, ImageNtHeader, MapAndLoad, etc.

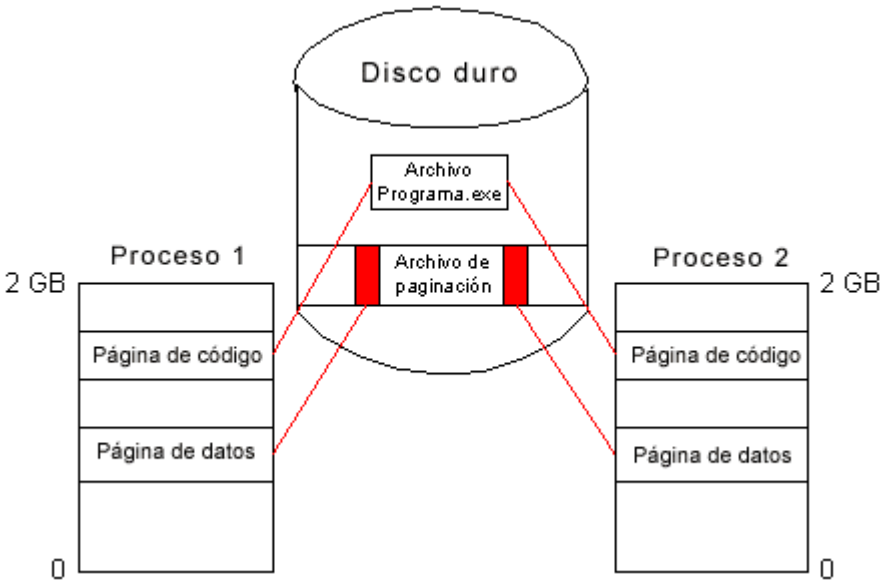
3. Se reserva un bloque de memoria virtual para almacenar todo el ejecutable, pero se compromete en el mismo espacio físico donde reside el archivo EXE. Es decir, el "Page Directory" indicará que el rango de páginas reservado cuenta con almacenamiento físico comprometido en el archivo "Programa.exe". De este modo no se lee todo el ejecutable para volcarlo a memoria, sino que se deja en disco, y cada vez que se lea una nueva instrucción para ejecutarla, se accederá a la página en disco (si no está todavía en RAM).

4. Cada vez que se vaya a ejecutar una instrucción de nuestro programa, se irá a leer a la dirección de memoria virtual correspondiente. Si la página de esa dirección virtual no se encuentra en ese momento en RAM, se producirá un fallo de página y el sistema entrará en el proceso de lectura de la página y volcado a RAM. Una vez que ya se cuenta con la nueva instrucción en memoria física, se continúa la ejecución.

Como hemos visto, el tamaño del ejecutable no influye (hasta cierto punto) en el tiempo de carga, ya que realmente no se realiza toda la carga del código ejecutable al arrancar, sino que se va haciendo conforme se necesita. Si el contenido de una página nunca llega a ejecutarse, esa página nunca se cargará en RAM, sino que permanecerá en el propio archivo ejecutable.

Además, si creamos una nueva instancia de nuestro programa (ejecutándolo de nuevo), se reservará otro bloque de memoria proyectándolo en el mismo espacio físico (el archivo EXE de nuevo), por lo que se reserva memoria virtual para cada una de las instancias, pero sólo se utiliza la memoria física que ocupa el archivo ejecutable en el disco.

En la figura de la derecha podemos ver un esquema de cómo se proyecta un mismo ejecutable sobre el espacio de direcciones virtuales de dos procesos.



Realizar operaciones de E/S a disco

Esta característica nos permite manipular cualquier estructura en memoria, cuando en realidad lo que estamos haciendo es manipular datos en disco. El propio sistema es el encargado de llevar a RAM las páginas que leemos, y volcar a disco las páginas que escribimos.

Supongamos que tenemos una aplicación que almacena datos en disco (por ejemplo los favoritos, o un historial de acciones o cualquier otro elemento persistente). El algoritmo típico en estos casos es:

1. Al arrancar la aplicación, leer los datos del archivo en disco y volcarlo en estructuras en memoria, como un array de registros, un TList, o un TStringList, etc.

2. Manipular las estructuras de memoria, según las acciones del usuario (añadir, borrar, cambiar nombre, etc.).

3. Al cerrar la aplicación, grabar de nuevo los datos a los archivos en disco

Este sistema, aunque muy utilizado, tiene bastantes inconvenientes:

- Si los archivos de datos son suficientemente grandes, el tiempo de lectura puede ser largo, ralentizando el arranque de la aplicación. Lo mismo ocurre a la hora de grabar de nuevo los datos al cierre de la aplicación.

  Esto se puede solucionar, o mejor dicho: disimular, con una bonita ventana de splash con la que el usuario pueda entretenerse mientras nosotros cargamos los datos; ¡¡incluso una barra de progreso!! (esto sólo para las aplicaciones más avanzadas) nos informa del tiempo que "puede" que tengamos que esperar. A veces, estas ventanas suelen estar demasiado tiempo abiertas, o la barra de progreso va demasiado lenta, porque la lectura que se está realizando se toma su tiempo.

  La solución definitiva pasa por usar archivos proyectados en memoria, ya que sólo se accederá a cada página del disco cuando necesite ser leída, acelerando el arranque y cierre de la aplicación.

- Si arrancamos dos instancias de la misma aplicación, estaremos poniendo en peligro la integridad de los datos, ya que el último que grabe en los ficheros será el que "gane la partida".

  Imaginemos que tenemos dos instancias del Internet Explorer o cualquier otro navegador. Si desde una de ellas añadimos un favorito, lo lógico es que desde la otra también lo veamos. Además, con este sistema, si desde ambas instancias añadimos favoritos, los últimos datos en grabarse (los de la aplicación que se cierre más tarde), serán los que perduren, ya que la aplicación cuando graba sus estructuras de memoria, sobrescribe los archivos de disco.

  La solución a esto suele pasar por arriesgarse a "la buena fe del usuario", o bien limitar nuestro programa a una sola instancia (demasiado drástico ¿no?).

  Los archivos proyectados también nos ayudan, ya que nos permiten compartir una misma proyección de archivo entre varios procesos, para que todas las instancias vean y modifiquen los mismo datos. Incluso podemos hacer que ciertos datos se compartan, hasta que algún proceso modifique su contenido. Esto es conocido como es el método "Copy on Write" que utilizan muchos sistemas operativos.

- Se consumen muchos más recursos, ya que necesitamos espacio en RAM, en el archivo de intercambio y en los archivos de datos para almacenar las estructuras donde volcaremos los datos.

  Sin embargo, con archivos proyectados, no necesitaremos espacio en el archivo de intercambio (porque utilizamos el mismo archivo de datos), y sólo ocuparemos espacio en RAM para aquellas páginas que sean leídas durante la ejecución del programa, y no para todas.

- La lectura "tradicional" de un archivo puede suele ser más lenta que la lectura que hace el gestor de memoria virtual. Esto es debido a que cuando se lee con funciones de fichero (ReadLn, fread, etc.) se realiza una operación de E/S por cada llamada a dicha función.

  Por ejemplo, si tenemos un archivo con 1.000 registros de 20 bytes cada uno de ellos, tendremos que hacer un bucle de 1.000 llamadas a Read o fopen. Por cada una de estas iteraciones, se estará produciendo una lectura física en disco (esto sin tener en cuenta la caché de bajo nivel que pueda tener el disco duro), con el correspondiente posicionamiento de las cabezas lectoras. Al final, habremos realizado 1.000 posicionamientos de las cabezas del disco para leer un total de 19,5 KB.

  Sin embargo, la misma operación realizada a través de archivos proyectados efectuaría las lecturas en bloques de 4 KB: al intentar acceder al primer registro, en realidad se

leerían todos los siguientes hasta completar la página completa, es decir: 210 registros (4.096 / 19,5). De este modo las siguientes lecturas de página se harían al acceder a los registros 211, 421, 632 y 843. En total, se han realizado 5 lecturas de 4.096 bytes cada una, contra las 1.000 lecturas de 20 bytes cada una según el método tradicional.

Sin ninguna duda, al realizar menos lecturas físicas, se posicionarán menos veces las cabezas lectoras del disco, con lo que la operación de E/S será mucho más rápida.

Crear zonas de memoria compartida

Quizá la utilidad estrella para los archivos proyectados en memoria sea la compartición de datos entre procesos.

Como ya hablamos en su día, la plataforma Win32 pone muy difícil que los datos puedan ser compartidos entre distintos procesos, dado el carácter privado de su espacio de memoria y el nivel de seguridad que se busca a la hora de desarrollar un sistema operativo robusto. Para solucionar esto, la gente de Microsoft recomienda el uso de archivos proyectados, utilizando dos métodos:

- **Proyecciones nombradas**: se basa en crear la proyección de archivo con un nombre, y abrir esa misma proyección desde otro proceso, referenciándola por su nombre. Este es uno de los sistemas para comunicar dos procesos independientes.
- **Herencia de descriptores**: los objetos proyección son objetos del núcleo del sistema operativo. Hasta ahora no hemos hablado este tipo de objetos, pero nos bastan con saber que se referencian a través de un descriptor (handle) y que estos descriptores pueden heredarse a los procesos-hijos (o subprocesos) del proceso que los creó. Este es uno de los sistemas para comunicar un proceso-padre con sus procesos-hijos.

Más adelante profundizaremos en estos métodos.

Cuando cualquiera de los dos procesos modifique algún dato, debe notificar al otro proceso (por ejemplo, con un mensaje de ventana) y éste dispondrá de los datos inmediatamente.

Internamente, la compartición de datos se consigue a través de una estructura llamada *"prototype page table entry"*. A esta *"cosa"*, que no sé ni cómo traducirla, le llamaremos PPTE.

Esta estructura, es parecida a nuestro amigo el "Page Table", sin embargo, todos los procesos tienen acceso a un único PPTE global, el cual gestiona qué páginas están compartidas entre procesos y cuales son las direcciones virtuales y físicas de cada una de estas páginas. Creo que con esto nos basta.

Además de compartir datos entre procesos, Windows permite desligar estos datos cuando alguno de los procesos hagan modificaciones. Es decir, los datos se comparten inicialmente, y continúan en este estado mientras no hayan sido modificados, cuando alguno de los procesos cambie el contenido de una página, se hará una copia de esta página y las modificaciones se harán sobre la nueva copia. El proceso que modificó el contenido ahora proyectará su memoria sobre la copia de la página, en vez de la página compartida. Este comportamiento se denomina *"Copy on Write"* o *"Copia con escritura"* y es uno de los métodos más utilizados en distintos sistemas operativos.

Windows proporciona otros métodos para compartir datos entre procesos, como el obsoleto DDE o el mensaje WM_COPYDATA, aunque internamente utiliza una sistema basado en archivos proyectados en memoria.

Las funciones disponibles

Ahora que sabemos cual es la base teórica, vamos a ver cómo se hace a través del API Win32.

Como su propio nombre indica, los archivos proyectados en memoria constan de tres componentes:

Archivos (objeto File), proyectados (objeto FileMapping) y memoria (objeto View)

Objeto File (archivo)

Se tratan de un objeto del núcleo que se representa por un descriptor. Este descriptor se obtiene a través de la función del API CreateFile, que aunque no lo parezca, se usa tanto para abrir un archivo existente como para crearlo.

Otra opción es utilizar un objeto de cualquier descendiente de la clase THandleStream que nos proporciona la VCL, aunque no vamos a entrar en este caso, ya que nuestro objetivo es centrarnos en el API Win32.

La función CreateFile en realidad se utiliza para abrir cualquier dispositivo de E/S: desde un fichero de disco o una partición física, hasta un puerto de comunicaciones, aunque nosotros nos vamos a centrar en los ficheros de disco, que es el caso que nos ocupa.

El prototipo de la función es el siguiente:

```
function CreateFile(
             lpArchivo:         PChar;
             dwModoAcceso:      LongWord;
             dwCompartir:       LongWord;
             lpSecuridad:       PSecurityAttributes;
             dwCreacion:        LongWord;
             dwAtributos:       LongWord;
             hFicheroPlantilla: THandle
         ): THandle;
```

Como podéis ver la función es bastante compleja. Vamos a intentar aclarar un poco el asunto:

- **lpArchivo**: un puntero a una cadena con el nombre y ruta del archivo a abrir.
- **dwModoAcceso**: indica el modo de apertura, pudiendo ser una combinación de los siguientes valores:
    - GENERIC_READ: para modo lectura.
    - GENERIC_WRITE: para modo escritura.
    - 0: para modo consulta. Este modo se utiliza consultar/modificar los atributos del archivo sin leer ni alterar su contenido.
- **dwCompartir**: indica si se podrá compartir del archivo mientras esté abierto por nuestro proceso. Si no queremos que pueda ser abierto, debemos indicar un 0. Si queremos compartirlo de algún modo podemos usar cualquier combinación de los siguientes valores:
    - FILE_SHARE_READ: se permiten aperturas en modo lectura.
    - FILE_SHARE_WRITE: se permiten aperturas en modo escritura.

- **lpSeguridad**: un puntero a la estructura de seguridad, para indicar si el descriptor creado puede ser heredado en los subprocesos. Los sistemas Windos 95/98/Me no soportan esta característica de seguridad de descriptores. Indicando un **nil** se utilizará las opciones por defecto (no heredable).
- **dwCreacion**:especifica el modo en que se comportará la función cuando el archivo no exista. Los posibles valores son:
    - CREATE_NEW: crea un nuevo archivo y falla si el archivo ya existe.
    - CREATE_ALWAYS: crea un nuevo archivo sobrescribiendo si existe.
    - OPEN_EXISTING: abre el archivo sin crearlo y falla si no existe.
    - OPEN_ALWAYS: abre el archivo sin existe o lo crea si no existe.
    - TRUNCATE_EXISTING: abre el archivo vaciándolo, y falla si no existe.
- **dwAtributos**: indica los atributos con que se marcará al archivo una vez realizada la apertura. Se puede pasar cualquier combinación de los siguientes valores:
    - FILE_ATTRIBUTE_ARCHIVE: indica que se debe marcar como "archivado".
    - FILE_ATTRIBUTE_COMPRESSED: indica que los datos deben comprimirse (sólo válido en sistemas NTFS).
    - FILE_ATTRIBUTE_HIDDEN: indica que debe marcarse como "oculto".
    - FILE_ATTRIBUTE_NORMAL: se deben eliminar todos los atributos que tenga el archivo. Este valor no puede combinarse con ningún otro.
    - FILE_ATTRIBUTE_READONLY: indica que está marcado como "sólo lectura".
    - FILE_ATTRIBUTE_SYSTEM: se debe marcar como archivo de sistema, lo que indica que no puede ser escrito ni borrado.
    - FILE_ATTRIBUTE_TEMPORARY: se debe marcar como archivo temporal. El sistema de archivos trata a los archivos temporales de forma distinta, intentando mantener el archivo continuamente en memoria RAM.

  Además de estos atributos, se pueden combinar una serie de banderas para modificar el comportamiento de la función. No vamos a entrar en estos detalles, ya que para nuestros ejemplos no vamos a utilizarlas.
- **hFicheroPlantilla**: indica el descriptor de un fichero abierto. Este fichero se utilizará como plantilla y se copiarán todos los atributos al nuevo archivo.

Esta función nos retorna el descriptor del objeto del núcleo de tipo "File". Si se indicó la bandera OPEN_ALWAYS o CREATE_ALLWAYS y el archivo existe, se retorna ERROR_ALREADY_EXISTS, y si se ha producido algún error, se retornará INVALID_HANDLE_VALUE.

Una vez que esté abierto el fichero, y ya no lo necesitemos más, debemos cerrarlo a través de la función propia para cerrar objetos del núcleo: CloseHandle.

Objeto FileMapping (Proyección de archivo)

Una vez que tenemos el objeto archivo disponible, podemos crear la proyección, o bien abrir una proyeción ya creada.

Los objetos proyección son otro tipo de objetos del núcleo, que están disponibles para el proceso que los creó y los sus procesos-hijo (si son heredables). En cualquier momento podremos crear un nuevo objeto proyección (a través de CreateFileMapping), abrir una proyección existente (con OpenFileMapping) o bien duplicar el descriptor (con DuplicateHandle).

La creación de un nuevo objeto proyección de archivo se hace a través de la función del API Win32 CreateFileMapping:

```
function CreateFileMapping(
            hArchivo:        THandle;              { archivo abierto                }
            lpSeguridad:     PSecurityAttributes; { estructura de seguridad        }
            flProtección:    LongWord;             { protección de la proyección    }
            dwTamañoMaxHigh: LongWord;             { tamaño máximo (32 bits + altos) }
            dwTamañiMaxLow:  Longword;             { tamaño máximo (32 bits + bajos) }
            lpNombre:        PChar                 { nombre de la proyección        }
       ): THandle;
```

Como vemos, esta función tampoco es sencilla, pero tranquilos, que allá vamos con la explicación:

- **hArchivo**: indica el descriptor del archivo que hemos abierto previamente. El modo de apertura de este archivo debe ser compatible con el parámetro "flProtección" indicado más abajo.
  Si este parámetro es $FFFFFFFF, se creará la proyección sobre el propio archivo de paginación del sistema, en vez de utilizar un archivo dedicado. Windows recomienda utilizar el archivo de paginación cuando sea posible, ya que las operaciones se ejecutarán más rápido. Esto es debido a que cuando cerramos la proyección, Windows intentará grabar a disco todas las páginas modificadas (llamadas dirty-pages, es decir: páginas sucias), y esto es una operación lenta. El sistema se ahorra este proceso si utilizamos el archivo de proyección en vez de un archivo físico en disco.

- **lpSeguridad**: un puntero a la estructura de seguridad, al igual que en la función CreateFile.

- **flProtección**: indica el modo de acceso que se permitirá a las distintas vistas de esta proyección. Los modos disponibles son:

  - PAGE_READONLY: permite acceso de sólo lectura a los datos de la proyección. Cualquier intento de modificar una página proyectada, desembocará en una violación de acceso. Los archivos utilizados para este tipo deben abrirse al menos con la bandera GENERIC_READ.

  - PAGE_READWRITE: permite acceso de lectura/escritura a los datos de la proyección. Los archivos proyectados con esta bandera deben abrirse con los modificadores GENERIC_READ y GENERIC_WRITE.

  - PAGE_WRITECOPY: permite acceso de lectura/escritura a los datos de la proyección, pero utilizando el método "Copia con escritura". Los archivos proyectados con esta bandera deben abrirse con los modificadores GENERIC_READ y GENERIC_WRITE.

  Además de las banderas de protección, se puede combinar cualquier número de las siguientes banderas:

  - SEC_COMMIT: compromete espacio físico en memoria, o en el archivo de paginación, para todas las páginas de la proyección. Este es el valor por defecto.
  - SEC_RESERVE: reserva memoria para todas las páginas de la proyección sin comprometer almacenamiento físico. Estas páginas puede comprometerse con alguna llamada posterior a VirtualAlloc. Esta bandera sólo es válida cuando se utiliza el descriptor de archivo $FFFFFFFF, es decir, el propio archivo de paginación del sistema.
  - SEC_IMAGE: la proyección que se está realizando es la imagen de un archivo ejecutable o librería DLL. Esta bandera no puede aparecer combinada con ninguna otra.

- **dwTamañoMaxHigh**: contiene los 32 bits más altos del tamaño máximo permitido para la proyección. Si se indica un cero en este parámetro, se utilizará el valor pasado en dwTamañoMaxLow.

- **dwTamañoMaxLow**: contiene los 32 bits más bajos del tamaño máximo permitido para la protección. Si el valor de dwTamañoMaxHigh y dwTamañoMaxLow es cero, se utilizará el tamaño del archivo pasado en el primer parámetro, o lo que es lo mismo, el valor retornado por GetFileSize(hArchivo), aunque esto no es válido si estamos proyectando sobre el archivo de intercambio (hArchivo = $FFFFFFFF).
  Si el tamaño total (el valor combinado de dwTamañoMaxHig y dwTamañoMaxLow) es mayor que el tamaño del archivo, el archivo en disco crecerá hasta el tamaño indicado.
  Si el tamaño total es menor que el tamaño del archivo, sólo se tendrá acceso a los datos dentro del rango indicado por la combinación de dwTamañoMaxHigh y dwTamañoMaxLow.
  Es importante calcular o estimar bien el tamaño que queremos darle al archivo, ya que una vez creada la proyección, no podremos modificar su tamaño. Un buen sistema para esto es crear la proyección de un tamaño lo suficientemente grande para que entren todos los datos provistos. Al finalizar el trabajo, cerrar la proyección, calcular el tamaño real que debería tener, y volver a crear otra proyección del tamaño justo, símplemente para ajustar el tamaño del fichero.

- **lpNombre**: indica un puntero a la cadena que contiene el nombre asignado a la proyección. Este nombre se utilizará para que otros procesos puedan acceder al objeto. Se permite cualquier caracter expecto la barra invertida "\". Para crear un objeto de proyección sin nombre, se debe indicar **nil** en este parámetro.

La función nos retorna un descriptor de objeto proyección si todo va bien. Si el nombre indicado ya está siendo utilizado, se retornará ERROR_ALREADY_EXISTS, y nos retornará el descriptor del objeto con ese nombre (como si hubieramos utilizado OpenFileMapping).
Si la función falla, se retorna **nil**.
Como cualquier otro objeto del núcleo, una vez terminenos de utilizarlo, debemos cerrarlo a través de la función CloseHandle.

Otra opción que se nos ofrece es acceder a un objeto proyección que haya sido previamente creado, ya sea en nuestro mismo proceso o en otros.
Esto se realiza a través de la función OpenFileMapping:

```
function OpenFileMapping(
             dwTipoAcceso:       LongWord; { el tipo de acceso               }
             bHeredarDescriptor: LongBool; { si se hereda a subprocesos      }
             lpNombre:           PChar     { nombre de la proyección a abrir }
         ): THandle;
```

Ahora la cosa es algo más sencilla:

- **dwTipoAcceso**: puede ser cualquier combinación de las siguientes banderas:

  - FILE_MAP_WRITE: Se abre la proyección para lectura/escritura. La proyección original tuvo que ser creada con la bandera PAGE_READWRITE.
  - FILE_MAP_READ: Se abre la proyección para sólo lectura.
  - FILE_MAP_ALL_ACCESS: Lo mismo que FILE_MAP_WRITE.
  - FILE_MAP_COPY: Se abre la proyección para lectura/escritura y "Copia con escritura". La primera vez que se modifican los datos de una página, se hace una copia de esta y se realizan las modificaciones sobre la copia. De este modo, el archivo original nunca será modificado. La proyección original ha tenido que ser creada con la bandera PAGE_WRITECOPY.

- **bHeredarDescriptor:** indica si el descriptor retornado se heredará a los subprocesos. Si un descriptor es heredable, cualquier proceso-hijo podrá tener acceso a él, permitiendo así otro sistema de comunicación entre procesos. De esto hablaremos más adelante, cuando tratemos el tema de comunicación entre procesos.
- **lpNombre**: un puntero a cadena con el nombre de la proyección que queremos abrir.

Retorna el descriptor del objeto proyección, o **nil** si no encuentra ningún descriptor con ese nombre o si las banderas indicadas en dwTipoAcceso no son compatibles.

Al igual que con CreateFileMapping, debemos cerrar el descriptor del objeto del núcleo, ya que al abrir un objeto ya existente en el núcleo, se incrementa un contador de referencias interno. Cerrando el descriptor (con CloseHandle), decrementaremos el contador de referencias, o destruimos definitivamente el objeto (si el contador de referencias llega a cero).

Objeto View (vista)

El último paso es crear una vista concreta sobre la proyección de archivos. La vista representa una porción del objeto proyección al que tendremos acceso a través de un puntero. Podemos crear una vista completa del archivo (un puntero al inicio), desde la mitad, una vista de 100 KB, etc.

Windows permite crear múltiples vistas sobre una misma proyección, incluso desde distintos procesos. La dirección de memoria retornada por la vista puede apuntar al mismo bloque de memoria virtual (en sistemas Windows 95/98/Me siempre es así), o a distintas zonas de memoria virtual (aunque realmente se proyectan sobre la misma memoria física).

La función que debemos utilizar para crear vistas es la siguiente:

```
function MapViewOfFile(
             hObjetoProyección:    THandle;  { objeto proyección             }
             dwTipoAcceso:         LongWord; { tipo de acceso a la vista     }
             dwDesplazamientoHigh: LongWord; { desplazamiento (32 bits altos) }
             dwDesplazamientoLow:  LongWord; { desplazamiento (32 bits bajos) }
             dwTamaño:             LongWord  { nº de bytes de la vista       }
         ): Pointer;
```

Y ahora la explicación:

- **hObjetoProyección**: indica el descriptor del objeto proyección que previamente hemos obtenido a través de la función OpenFileMapping o CreateFileMapping.
- **dwTipoAcceso**: indica el modo de acceso que se va a hacer sobre los datos de la vista. Los valores admitidos son los mismos que para la función OpenFileMapping.
- **dwDesplazamientoHigh**: contiene los 32 bits más altos del desplazamiento inicial. Si se indica un cero en este parámetro, se utilizará el valor pasado en dwDesplazamientoLow.

- **dwDesplazamientoLow**: contiene los 32 bits más bajos del desplazamiento inicial. El desplazamiento indica la posición a partir del inicio del archivo en que comenzará la vista, o dicho de otro modo: el incremento del puntero desde el principio del archivo. Para obtener el desplazamiento total, de combinará este valor con el indicado en dwDesplazamientoHigh. El único problema con el uso de estos parámetros, es que el valor combinado debe ser múltiplo de la granularidad de reserva. Este valor se obtiene a través de la función GetSystemInfo, en el campo dwAllocationGranularity de la estructura SYSTEM_INFO, aunque en plataformas x86 tiene un valor de 64.

- **dwTamaño**: indica el tamaño en bytes de tendrá la vista. Más allá de este tamaño, no se podrá acceder a los datos. Si se indica 0 en este parámetro, se creará una vista sobre el archivo completo.

La función retorna un puntero al inicio de la zona de memoria que representa la vista, o **nil** si se produce algún tipo de error.

Si proyectamos varias vistas sobre la misma proyección, no se nos garantiza que la dirección retornada por MapViewOfFile sea la misma, ya que el sistema buscará el lugar más adecuado para crear la vista.

Además de esta función, existe una versión extendida (MapViewOfFileEx), en la que se añade un nuevo parámetro: lpDirecciónBase. Este parámetro (un puntero genérico), nos permite indicar la dirección de inicio para la vista. Esta dirección de inicio debe ser múltiplo del valor dwAllocationGranularity, del que ya hablamos anteriormente. La función retornará la dirección de inicio de la vista, que será el valor que hemos pasado en ldDirecciónBase, o el múltiplo de 64 KB inmediatamente anterior. Si en la dirección indicada no es posible crear la vista (por falta de espacio o cualquier otro motivo), se retornará **nil**.

Esta función es útil para crear una vista en una posición fija, y que cualquier proceso acceda directamente a esta dirección (abriendo la proyección previamente), para leer los datos del archivo proyectado.

Una vez que hemos terminado con los datos de la vista, es necesario cerrarla para liberar los recursos asociados. Esto se hace a través de la función UnmapViewOfFile:

```
function UnmapViewOfFile(
             lpDirecciónBase: Pointer { dirección de la vista }
         ): LongBool;
```

En este caso la función es muy sencilla, simplemente debemos pasar la dirección de una vista previamente creada (el valor retornado por MapViewOfFile).

Windows garantiza que todas las páginas que sean modificadas y no hayan sido todavía guardadas en el archivo, se volcarán al archivo proyectado, aunque esto sólo es así cuando se ha creado una proyección sobre un archivo físico, y no sobre el archivo de paginación del sistema.

De todas formas, en cualquier momento podemos forzar a que Windows almacene el contenido de las páginas modificadas en el archivo físico, a través de la la siguiente función:

```
function FlushViewOfFile(
             const lpDirecciónBase: Pointer;
             dwNumeroBytes:         LongWord
         ): LongBool;
```

Esta función también es sencilla, simplemente se nos pide la dirección a partir de la que queremos grabar, y el número de bytes totales.
Nos retornará TRUE si las páginas han sido grabadas con éxito, o FALSE si ha ocurrido algún error.

Bueno, estas son las funciones disponibles para el manejo de archivos proyectados en memoria. Ahora vamos a profundizar en uno de los usos más importante de los archivos proyectados: la comunicación entre procesos.

Comunicación entre procesos

Uno de los usos más importantes que le podemos dar a los archivos proyectados es para comunicación de datos entre dos o más procesos.

La arquitectura de Windows pone muy difícil esta tarea, ya que la gente de Microsoft, en un intento de hacer más robusto el sistema operativo, se preocupó mucho de hacer inaccesible el espacio de memoria de un proceso.

La solución recomendada para este problema pasa por el uso de los archivos proyectados en memoria, utilizando las características de los denominados "objetos del núcleo", en los que a continuación vamos a profundizar.

Objetos del núcleo

¿Y por qué vamos a hablar de objetos del núcleo en un artículo sobre archivos proyectados? Pues porque los archivos proyectados son un tipo de objetos de núcleo, y es muy importante comprender el funcionamiento de los objetos del núcleo para entender a los archivos proyectados.

Estos objetos son un tipo especial dentro de todos los que podemos crear dentro del sistema Win32. Básicamente, Win32 tiene tres tipos de objetos:

1. Objetos de usuario (como ventanas, menús, ganchos, etc.): son objetos identificados por un descriptor (que no suele ser más que la dirección de memoria donde comienza el objeto). Cualquier proceso que conozca el descriptor puede acceder al objeto en cuestión.

2. Objetos gráficos o GDI (como el objeto Bitmap, Font, Pen, Brush, etc.): estos objetos son privados al proceso que los crea, aunque éste tiene disponibilidad total del objeto.

3. Objetos del núcleo (como la proyecciones de archivo, hilos, etc.): son objetos gestionados por el núcleo del sistema operativo. Estos objetos pueden ser accedidos desde varios procesos, siempre y cuando se conozca el descriptor que los identifica.

Todos ellos se identifican por un descriptor (el famoso handle), aunque el significado de este descriptor es muy distinto dependiendo del tipo de objeto (puede ser un puntero, un índice de un tabla, etc.).

Los objetos del núcleo son un grupo de objetos con ciertas características especiales. La principal de ellas es que se almacenan en una tabla interna al proceso. Cada vez que se crea un objeto del núcleo, se busca una entrada libre en esta tabla y se retorna la posición que ocupa. Esa posición (el índice) será el descriptor del objeto. Esta es la razón por la que los descriptores del objetos del núcleo suelen ser números bajos, mientras que los descriptores de objetos de usuario o GDI son número altos (porque son direcciones de memoria).

Un proceso tendrá acceso a los objetos del núcleo que aparezcan en su tabla de objetos, aunque él no los haya creado.

Dentro de estos objetos del núcleo tenemos: archivos, variables de entorno, procesos, hilos, mutex, eventos, semáforos, tuberías y proyecciones de archivo.

Cada objeto del núcleo tiene una función específica para su creación, normalmente CreateXXX (donde XXX es el nombre del objeto): CreateFile, CreateFileMapping, CreateSemaphore, CreateMutex, etc.

Básicamente, lo que se hace durante la creación de un objeto del núcleo es crear una nueva entrada en la tabla de objetos, y retornar el índice asociado. Además, en la misma tabla se marcan las propiedades de cada objeto, como sus atributos de seguridad, si es heredable etc.

En cada una de estas funciones, tenemos un parámetro que es un puntero a una estructura de tipo SECURITY_ATTRIBUTES. Este es, precisamente, el atributo que controla si el descriptor a crear va a ser heredado o no por los subprocesos. Esta estructura se define del siguiente modo:

```
type
    _SECURITY_ATTRIBUTES = record
        nLength:              LongWord;
        lpSecurityDescriptor: Pointer;
        bInheritHandle:       LongBool;
    end;
    TSecurityAttributes = _SECURITY_ATTRIBUTES;
```

- **nTamaño**: indica el tamaño de la estructura. Se debe asignar siempre el valor sizeof(Estructura). A modo de curiosidad: este tipo de campos es muy habitual encontrárselos en las estructuras definidas en Win32 y su principal misión es detectar distintas versiones de la estructura durante la ejecución. Dentro del código de las funciones que utilizan esas estructuras, se comprueba el tamaño, y dependiendo de su valor, se puede deducir que versión de estructura se está utilizando, y actuar en consecuencia.

- **lpDescriptorSeguridad**: es un puntero a una estructura de tipo SECURITY_DESCRIPTOR, que representa la seguridad que se le dará al objeto (qué usuarios podrán acceder, con qué permisos, etc.). Indicando un **nil**, se asignará el descriptor de seguridad por defecto. Este atributo sólo tiene sentido en sistemas NT/2000/XP, ya que Windows 95/98/Me no soporta seguridad. No vamos a entrar en este tema, ya que no sé decir mucho más sobre esto. (-:

- **bHereadarDescriptor**: este es el campo más importante. Indica si el descriptor es heredable. La herencia de descriptores se basa en que, cuando un descriptor se marca como heredable, éste se copiará a la tabla de objetos en cada proceso hijo que sea creado. Es decir, cuando llamamos a CreateProcess desde un proceso, Win32 recorre la tabla del proceso padre (el que llama a CreateProcess), y copia cada una de las entradas que han sido marcadas como heredable a la tabla del proceso hijo. Además, Win32 garantiza que cada entrada se copia en la misma posición que en la tabla-padre, es decir: conserva el descriptor.

Además de la creación de objetos, en algunos casos es posible la apertura de objetos previamente creados, a través de funciones OpenXXX (como nuestra amiga OpenFileMapping). Estas funciones, lo que hacen es copiar la entrada correspondiente de la tabla de objetos del proceso creador (algo así como forzar la herencia).

Cada objeto del núcleo cuenta con un contador de referencias, que no es más que un valor que indica la cantidad de procesos que tienen acceso al objeto. Cuando se crea un objeto del núcleo, se añade la entrada a la tabla de objetos y se marca el contador con 1. Cada vez que el objeto se hereda o abre desde otro proceso, el contador de referencias se incrementa en uno. Cuando ya no sea necesario utilizar un objeto del núcleo, debe cerrarse a través de una función genérica: CloseHandle. Esta función, lo que hace es eliminar la entrada en la tabla de objetos (aunque no el objeto en sí), y restar uno al contador de referencias. Cuando el contador de referencias de un objeto llega a 0, todos los recursos asociados al objeto se liberan y éste se considera destruido.

Esta es la razón por la que hay que llamar a CloseHandle siempre, hayamos creado o abierto el objeto, ya que debemos decrementar el contador de referencias, y será el sistema el encargado de liberar el objeto.

Este es el comportamiento que tienen los objetos de núcleo, y de él nos podemos aprovechar para la comunicación entre procesos.

Básicamente nos podemos comunicar con otros procesos con la siguientes métodos:

Objetos nombrados

Este sistema se utiliza para comunicar dos procesos independientes, sin ningún "parentesco" entre ellos. Se basa en la apertura del objeto del núcleo a través de los denominados *"named-filemapping"*, es decir: proyecciones de archivo nombradas.

Ya hemos hablado algo de esto, y básicamente, se trata de asignar un nombre a una proyección de archivo y abrir la proyección utilizando el nombre que hemos asignado.

Cuando se abre la proyección, en realidad lo que está ocurriendo es que se incrementa el contador de referencias del objeto, se copia la entrada de la tabla de objetos del núcleo y devuelve el mismo descriptor (porque la copia se hace en el mismo índice de la tabla).

Para abrir la proyección tenemos dos opciones: utilizar la función para tal efecto (OpenFileMapping), o intentar crear una proyección con el mismo nombre. Si ya existe una proyección con ese nombre, la función CreateFileMapping incrementará el contador de referencias y nos retornará ERROR_ALREADY_EXISTS. Este código de error puede despistarnos, ya que en muchas ocasiones no se estará produciendo ningún error, sino que será la situación esperada. Sin embargo, en otras ocasiones sí que será un error, ya que no podemos garantizar que se nos reserve un nombre de proyección. Si, por ejemplo, queremos crear una proyección llamada "Mi proyección", nadie nos garantiza que otro proceso (incluso programado por otra persona o empresa), haya utilizado este nombre. Para solucionar esto, es recomendable utilizar nombres largos y descriptivos (de hasta 256 caracteres), por ejemplo indicando nuestro nombre, la fecha de programación o incluso un GUI, como puede ser:

JM.7C7B001B-2831-4078-A5DD-2B89C0FD91F7.Nombre de la proyección

El *"modus operandi"* a seguir para compartir datos es el siguiente:

1. Desde el primer proceso: crear una proyección de archivo con un nombre dado.
2. Desde el segundo proceso: abrir la proyección con el mismo nombre, o bien intentar crearla y comprobar si retorna ERROR_ALREADY_EXISTS.
3. Utilizar el objeto indistintamente desde uno o otro proceso. Cada vez que se produzcan cambios, puede ser recomendable notificar a los demás procesos (más adelante hablaremos de esto).
4. Al finalizar los procesos que hayan abierto o creado el objeto, se debe cerrar el descriptor con la función CloseHandle.

Este es el sistema más sencillo y práctico, así que he seguido este método en el ejemplo de Delphi que acompaña al artículo (encapsulado en la clase TProyeccion).

Herencia de descriptores

Este caso se basa en la característica que nos ofrece Windows de que un proceso pueda crear procesos hijos (o subprocesos). En la documentación de Win32 se define proceso-hijo como "un proceso que se crea por otro proceso, llamado proceso-padre". La definición no es muy buena (yo diría que es bastante mala), pero para ir tirando nos puede servir. Para aclarar, podemos decir que cada vez que desde nuestros programas llamamos la función CreateProcess, estamos creando un proceso-hijo.

Ya sabemos que al aplicar herencia de descriptores, en realidad lo que se hace es copiar la entrada correspondiente de la tabla de objetos del núcleo e incrementar en uno el contador de referencias. Lo único que debemos hacer, es informar al nuevo proceso del descriptor del objeto, para que pueda acceder a él.

El modo de comunicar estos procesos "consanguíneos" es a través de los descriptores de objetos del núcleo.

El esquema es el siguiente:

1. Crear una proyección de archivo (como ya sabemos hacer), indicando TRUE en el campo bHeredarDescriptor de la estructura SECURITY_ATTRIBUTES.
2. Crear un proceso hijo llamando a CreateProcess desde el proceso padre.
3. Notificar al proceso recién creado (el hijo). Para esto se puede utilizar cualquier sistema de notificación, como mensajes de ventana, variables de entorno, etc.
4. Cuando hayamos terminado de utilizar el objeto, debemos llamar a CloseHandle, tanto en los procesos hijos como en el padre.

Duplicación de descriptores

¿Alguien pensaba que esto era todo? Pues no, en la plataforma Win32 siempre hay alguna sorpresa. El tercer y último método para compartir información se basa en la duplicación de descriptores.

Básicamente es lo mismo que la herencia de descriptores, pero la copia de descriptores se hace en el momento que queramos, y no durante la creación del proceso.

La duplicación se hace a través de la siguiente función:

```
function DuplicateHandle(
             hProcesoOrigen:      THandle;
             hDescriptorOrigen:   THandle;
             hProcesoDestino:     THandle;
             lpDescriptorDestino: PHandle;
             dwAcceso:            LongWord;
             bHeredarDescriptor:  LongBool;
             dwOpciones:          LongWord
          ): LongBool;
```

Que nadie se asuste que la función es sencilla:

- **hProcesoOrigen**: descriptor del proceso que contiene el descriptor del objeto. Este valor se puede obtener llamando a la función GetCurrentProcess (para obtener el descriptor del proceso llamante), o la función OpenProcess (para abrir cualquier otro proceso).
- **hDescriptorOrigen**: el descriptor del objeto que queremos copiar.
- **hProcesoDestino**: descriptor del proceso donde se copiará el objeto. Este puede ser el mismo o distinto valor que hProcesoOrigen.

- **lpDescriptorDestino**: un puntero donde se almacenará el valor del nuevo descriptor. Si se pasa un **nil**, el descriptor se duplicará, pero no se copiará su valor a este parámetro.
- **dwAcceso**: indica el tipo de acceso que se permitirá al objeto recién copiado. Los valores permitidos dependen del tipo de objeto que estemos copiando, en nuestro caso, con los objetos proyección de archivo, se permiten los mismos valores que el parámetro dwTipoAcceso de la función OpenFileMapping.
- **bHeredarDescriptor**: indica si el objeto recién copiado será heredable.
- **dwOpciones**: indica opciones especiales, pudiendo utilizarse cualquier combinación de los siguientes valores:
    - DUPLICATE_CLOSE_SOURCE: Cierra el descriptor (con CloseHandle) del objeto origen.
    - DUPLICATE_SAME_ACCESS: Se ignora el parámetro dwAcceso y se utiliza el mismo valor que en el objeto origen.

La función retorna TRUE si todo ha ido bien, o FALSE en caso contrario.

Para compartir datos entre procesos utilizando esta técnica podemos optar por dos enfoques: duplicando el descriptor desde el proceso origen, o desde el proceso hijo.

En el primer caso, los pasos a seguir son los siguientes:

1. Obtener el descriptor del proceso destino. Normalmente esto lo obtendremos a través de la función CreateProcess, a la hora de crear el proceso-hijo.
2. Duplicar el descriptor de la proyección desde el proceso origen.
3. Notificar al proceso destino de su nuevo descriptor.
4. Ambos procesos (el origen y el destino), deben cerrar los objetos a través de la función CloseHandle.

En el segundo caso (duplicar desde el destino), los pasos a seguir son:

1. Obtener el descriptor del proceso origen. Para ello necesitamos el identificador de proceso (Process ID) y obtener el descriptor a través de la función OpenProcess. Una llamada típica suele ser algo así como:
   hProcess = ::OpenProcess(PROCESS_DUP_HANDLE, FALSE, ProcessID);
2. Duplicar el descriptor de la proyección desde el proceso destino.
3. Ambos procesos deben cerrar los descriptores.

Notificación de procesos

En todos los sistemas de comunicación entre procesos hemos hablado de "notificar al proceso destino".

Vamos a tratar esta tema, aunque en realidad no tenga mucho que ver con los archivos proyectados, pero sí con su uso más importante. Además, la aplicación de ejemplo en Delphi, utiliza la notificación junto con los archivos proyectados, para comunicar varios procesos entre sí.

La notificación de procesos consiste en informar a un proceso de cierto dato, normalmente un descriptor.

Existen varios métodos para notificar a un proceso de que ha ocurrido "algo", aunque ahora sólo voy a exponer el más típico: los mensajes de ventana.

La explicación completa de qué son los mensajes de ventana o el bucle de mensajes, cómo funcionan, y cómo los implementa internamente Win32 sería muy extensa, así que vamos a suponer que todo el mundo sabe lo que es un mensaje, aunque no comprendamos muy bien qué ocurre internamente.

El sistema de mensajes se basa en registrar un mensaje especial, y notificarlo a todas las ventanas del sistema.

A este mensaje sólo responderán aquellas aplicaciones que hayan registrado previamente este mensaje, es decir, nuestras aplicaciones.

El paso de registrar el mensajes es sencillo, nos bastan con la siguiente función del API:

```
function RegisterWindowMessage(
             lpNombre: PChar { nombre del mensaje }
         ): LongWord;
```

Simplemente indicando una cadena con el nombre del mensaje, el sistema nos retorna un identificador de mensaje único, dentro del rango $C000 hasta $FFFF. Cuando se intenta registrar varias veces el mismo nombre de mensaje, el sistema retornará el mismo identificador.

En nuestro ejemplo para Delphi, puede verse el uso que he dado a esta función dentro del método Registrar de la clase TNotificacion.

Una vez registrado, debemos lanzarlo a todas las aplicaciones del sistema, a través de la función PostMensage o SendMessage. No vamos a entrar en la diferencia entre ambas funciones, y lo dejaremos para otra ocasión, en que tratemos en profundidad los mensajes de ventana.

La sintaxis de ambas funciones es la misma, así que sólo explicaré una de ellas:

```
function SendMessage(
             hWindow: HWND;     { descriptor de ventana destino      }
             Msg:     LongWord; { identificador del mensaje a enviar }
             wParam:  LongInt;  { primer parámetro                   }
             lParam:  LongInt   { segundo parámetro                  }
         ): LongInt;
```

Los parámetros son los siguientes:

- **hWindow**: descriptor de la ventana destino. Los mensajes de ventana (como su propio nombre indica) se envían a una ventana específica. Este valor podemos obtenerlo a través de cualquier función que nos retorne un descriptor de ventana (como CreateWindow, GetWindow, FindWindow, etc.).
  Existe un valor especial contenido en la constante HWND_BROADCAST. Este valor le

indica a Windows que distribuya el mensaje por todas las ventanas del sistema, no sólo a una ventana concreta.

- **Msg**: indica el identificador del mensaje a pasar. En nuestro caso concreto, este valor lo obtendremos a través de la función RegisterWindowMessage, aunque para otros casos, podemos utilizar un mensaje predefinido por el sistema (normalmente una constante que comienza por WM_) o un mensaje de usuario (cualquier valor superior a WM_USER, vamos: 1024).
- **wParam**: en este parámetro se puede pasar cualquier valor de 32 bits como parámetro. La interpretación que se dará a este valor depende del mensaje que pasemos. En nuestro caso, vamos a utilizar este valor para pasar el descriptor de la ventana que envía el mensaje.
- **lParam**: este parámetro funciona como wParam, pudiendo pasarse otro valor de 32 bits.

En el ejemplo, se usa la función SendMessage junto con el parámetro especial HWND_BROADCAST dentro del método Enviar de la clase TNotificacion.

El último paso que debemos dar, es interceptar el mensaje que recibe la aplicación. Para ello he utilizado una técnica llamada "Subclasificación del bucle de mensajes", aunque no voy a entrar a detallarla, ya que para ello, lo primero que tendría que hacer es explicar qué es un bucle de mensajes. Símplemente diré que me he apoyado en la VCL para simplificar el código, concretamente en la propiedad WindowProc de la clase TControl.

Puede verse cómo hacer la "subclasificación" dentro de la clase TMainForm, a través del método SubClassWndProc.

Sincronización entre procesos

En ciertos casos, además de la notificación es necesaria una sincronización al acceso de la memoria entre procesos. Esto es necesario para asegurarnos la coherencia de los datos en memoria, para que un proceso lea datos cuando realmente están disponibles o completos, o un proceso grabe cuando ningún otro lo está haciendo.

Para asegurarnos la sincronización entre procesos, se utilizan las mismas técnicas y objetos que para la sincronización entre hilos (semáforos, eventos, etc.), que está explicando nuestro compañero Salvador Jover en su serie sobre Threads. Así que para profundizar en este tema, os remito a su serie de artículos, en la que todos aprenderemos mucho.

Conclusión

Esto es todo lo que han dado de sí los archivos proyectados en memoria. Creo que con lo que aquí he expuesto, todo el mundo está en condiciones de empezar a usar este sistema, ya sea para compartir datos entre aplicaciones, o utilizarlo como método de entrada/salida a disco.

Bueno, pues esto se acaba, tanto el artículo como la primera parte de la serie "Los rincones del API Win32".

Espero que en estos cuatro artículos hayamos comprendido un poco mejor cómo se maneja la memoria en la plataforma Win32. Hemos abarcado casi todos los aspectos, desde los más sencillos y documentados, hasta los más oscuros.

Nos veremos en la próxima serie de artículos de "Los rincones del API Win32" que tratará sobre... ¿alguna sugerencia?

Los ejemplos

Todo lo que hemos ido explicando, se utiliza de modo práctico en el siguiente ejemplo:

Delphi 5

Un pequeño editor de texto (un Memo) que actualiza su contenido a todas las instancias de la aplicación. Para ello utiliza archivos proyectados en memoria y un sistema de notificación basado en mensajes HWND_BROADCAST.
Ejemplo en zip (http://www.lawebdejm.com/?id=22141)

Autor: JM - http://www.lawebdejm.com