Los rincones del API Win32
La memoria virtual

Este artículo explica en profundidad cómo usar la memoria virtual y el modo de gestionarla en la plataforma Win32.

Índice

Memoria, y más memoria

La memoria es el elemento más utilizado de un ordenador, y desde nuestros programas estamos continuamente accediendo a ella, ya sea para leer las instrucciones que debemos ir ejecutando, como para leer/escribir los datos con los que operamos. Aunque desde el punto de vista del hardware, la memoria siempre es igual (chips con megas y más megas, y cada byte numerado con una dirección única), el modo de gestionarla que tiene el sistema operativo y la manera de permitirnos acceder a ella ha ido evolucionando durante los últimos tiempos.

Si volvemos al pasado unos cuantos años y hacemos un programa para MS-DOS (que no es lo mismo que un programa de consola para Windows), el sistema operativo nos da acceso a toda memoria, se fía de nosotros (je je, pobre, no sabe el error que está cometiendo). Es decir, desde MS-DOS podemos acceder con un puntero al byte 0, al 5784 o al que queramos, y de este modo nos estaremos metiendo donde nadie nos ha llamado.

Desde nuestros programas esto es muy sencillo, simplemente apuntando una variable puntero a una dirección fija:

```
{
    int *p;

    p = (int *) 0xFFFF;
    [...]
}
```

Este código será válido si el sistema operativo realmente nos da acceso a esa zona de memoria.

Esto tiene su parte buena: control absoluto. Una utilidad típica a esto solía ser direccionar un puntero hacia la zona de memoria de la pantalla, y escribir directamente en ella, como si fuera una matriz de caracteres. Con eso se conseguía una rapidez similar a la del ensamblador a la hora de dibujar en la pantalla. Otra ventaja era que dos programas residentes en memoria podían comunicarse muy fácilmente entre si, ya que el uno podía ver la memoria del otro.

Por su contra, el sistema era tremendamente inestable, ya que lo más normal es que los punteros se nos fueran de madre y acabásemos sobrescribiendo los datos o el código de otros programas, o lo que podía ser peor: machacar el sistema operativo ¡¡o incluso la BIOS!!

Allá cuando nació Windows 3.x, las cosas cambiaron ligeramente, ya que la gente de Microsoft comenzó a verificar que no sobrescribiesemos algunas zonas de memoria donde residía el sistema operativo. Cuando intentábamos escribir en alguna zona protegida, Windows lanzaba un “Access Violation” para defenderse de los intrusos. De este modo se conseguía un sistema más estable, aunque en aquella época no era posible mucho más, ya que todavía podíamos tocarle las tripas a zonas de memoria de otros procesos, y eso hacía que si fallaba nuestro programa (por escribir en la memoria del vecino) hiciésemos que otros procesos se fuesen al traste, todo en cascada. Internamente teníamos toda la memoria para nosotros, pero no podríamos leer ni escribir en zonas protegidas por Windows. Este sistema de memoria es lo que llamaron el famoso “Modo protegido” de Windows.

A partir de la introducción de Windows 95 comenzó la era de los 32 bits, o lo que es lo mismo para Window-hablantes: la era de la arquitectura Win32. Esto puede poner los pelos de punta a más de uno, porque realmente hay cosas con las que hemos ido a peor (ahora nos encontramos las siglas M$ hasta en la sopa), pero también hay que decir que con Win32 el sistema completo se vuelve más estable, a la vez que más escalable. Al César lo que es del César.

Ahora vamos a ver porqué me atrevo a decir esto.

Modelo de memoria plano

Cada vez que hacemos doble clic sobre un ejecutable (o cualquier otra manera de iniciar un programa), Windows llamará a la función CreateProcess para iniciar un nuevo proceso. Una de las cosas que hace esta función (entre otras muchas) es crear un espacio privado de direcciones virtuales de 4 GB... ¿Mandeeeeeeeee? A muchos lo único que le sonará familiar son los 4 GB, porque el resto parece chino. No es tan complicado, esto significa que Windows considera que tenemos una memoria de 4 GB (casi ná) y encima toda para nosotros. Así que disponemos de un rango de direcciones desde 0x00000000 hasta 0xFFFFFFFF, desde 0 hasta 11111111111111111111111111111111 (que nadie los cuente, que son 32 bits) o en cristiano: desde el byte 0 al byte número 4.294.967.295.

En teoría podríamos recorrernos ese rango a través de un puntero y todo funcionaría. Y digo en teoría porque las cosas no son tan sencillas, pero luego veremos por qué. A este modo de gestionar la memoria se le denomina “Modelo de memoria plano”.

Que tengamos un espacio de direcciones de 4 GB no significa que tengamos 4 GB de memoria RAM instalada (a ver quien es el afortunado), sino que la suma entre la memoria RAM física, y la memoria virtual puede llegar a 4 GB. ¿Y qué es la memoria virtual?

Memoria Virtual

Es un tipo de memoria que utiliza el sistema, pero en vez de almacenarse en los chips de la memoria RAM, se almacena en un archivo que está en nuestro disco duro.

Para descubrir los orígenes de la memoria virtual tenemos que remontarnos a los años 40. En esa época tan sólo existía un ordenador programable: el ENIAC, que debía ocupar como una habitación entera, lleno de válvulas y relés por todos los lados, y programable con tarjetas perforadas. Los programadores de semejante aparato se dieron cuenta de que los tiempos de búsqueda de sus programas eran enormes, así que propusieron asignar un bloque de memoria antes de iniciar la ejecución del programa, y después mantener este bloque creado durante toda su vida. Esto aceleraba los accesos de memoria, ya que ya estaba creada para el programa, sin embargo era un sistema muy rígido, ya que una vez asignada la memoria no se podía ampliar ni reducir su tamaño, y en esa época la cosa no estaba como para desperdiciar ni un solo octeto. Ya en los años 50, investigadores de la universidad de Manchester inventaron un método más astuto: dividirían la memoria en zonas (llamadas páginas lógicas) y almacenarían todas estas páginas en un tambor magnético (los abuelos de los discos duros que conocemos actualmente) . Cuando se fuera a ejecutar un programa, se llevaban a memoria centrar un grupo de páginas, dependiendo de los requisitos de memoria del programa a ejecutar. Podía ser una sola página (si el programa era pequeño), o todo el tambor completo (si el programa tenía mucho código o datos). Incluso durante la ejecución del programa, si era necesaria más memoria, se almacenaban en el tambor aquellas páginas que ya no fueran necesarias, y se volcaban ven memoria nuevas páginas desde el tambor.

Este método no llego a implementarse hasta 1962, cuando se construyó el ordenador Atlas, que fue el primer ordenador del mundo en utilizar un sistema de lo que se denominaría “memoria virtual”.

Bien, volvamos a nuestro tiempo. Decíamos y que la memoria virtual es aquella que no se almacena en los chips de memoria, sin en un archivo de nuestro disco duro, que normalmente tiene

extensión SWP (de Swap = intercambio). Este archivo, llamado archivo de paginación o de intercambio, sirve como una ampliación auxiliar de memoria cuando las cosas se ponen feas, es decir, cuando el sistema escasea de memoria (y en los tiempos que corren eso es demasiado habitual).
Cuando tenemos muchas aplicaciones abiertas y el tamaño de todas ellas supera el total de memoria RAM física instalada, Windows mueve parte del contenido de la memoria al disco, para dejar espacio libre para nuevas aplicaciones, concretamente, mueve aquellas partes que hace más tiempo que no han sido accedidas. Cuando esos datos se necesitan de nuevo, se cargan otra vez en memoria. Por esta razón, cuando nuestro equipo está escaso de memoria, notamos que no para de leer del disco duro, porque está continuamente moviendo zonas de memoria del disco a RAM y viceversa (y por eso va lento).

Como veis, esto de la memoria virtual no es ninguna idea revolucionaria, y ya en los años 50 se pensaba en este sistema, aunque con las limitaciones de hardware que había en esa época.

Memoria paginada

Este trasiego de datos entre la memoria RAM y el archivo de intercambio no se realiza byte a byte, ya que sería muy lento, sino página a página. Una página es la cantidad de datos que lee cada vez que accede al archivo de intercambio. En procesadores x86 su tamaño es de 4 KBytes. En muchas operaciones, el tamaño más pequeño de memoria con el que podemos trabajar es el tamaño de página, así que se considera como la unidad mínima de memoria (realmente no es la unidad mínima, aunque para el sistema operativo sí lo es).

Espacio de direcciones virtuales

Como todos sabemos virtual significa falso, irreal, ficticio... de ahí el nombre de memoria virtual (porque no es auténtica memoria RAM). Pero... ¿porqué se llama espacio privado de direcciones virtuales? Pues porque es un rango de direcciones falso. Simplemente se trata de una numeración que Windows asigna a las páginas de memoria. Cuando Windows nos dice que una variable está almacenada en la posición 0x00000123, no es que esté en esa posición en RAM, sino que le ha asignado la casilla 123 a nuestra variable. Así cada proceso (o programa) en ejecución tendrá su propio espacio de direcciones (de ahí lo de “privado”) y un proceso no puede acceder a los datos de otro, porque no tiene ninguna forma de referenciarlos.

Pongamos un ejemplo más cotidiano: Si nosotros entramos al portal número 22 de la calle “X”, no significa que estemos entrando en el portal número 22 de la calle “Y”. Del mismo modo, si accedemos a la posición de memoria 22 del proceso “X”, no estamos accediendo a la posición 22 del proceso “Y”, ya que ambas posiciones de memoria son independientes. Del mismo modo, si se cae el edificio del portal 22 de la calle “X”, el de la calle “Y” seguirá en su sitio. Así cuando alguna operación escribe en una dirección equivocada, estará sobrescribiendo un dato de su mismo proceso, pero no de otros, por lo que el único perjudicado es el proceso que escribió mal.

Pero las direcciones virtuales tienen que almacenarse en la memoria, por lo que también estarán situadas en una posición de memoria determinada, así que cada dirección virtual tiene una equivalencia (o proyección) con dirección real, que es donde está almacenado el dato físicamente.

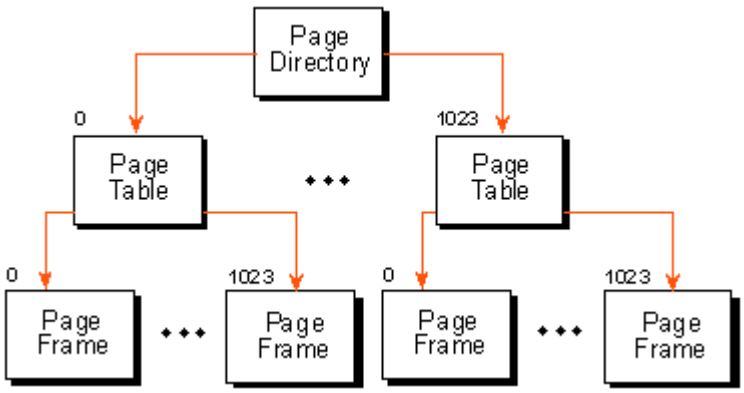
Internamente, Windows es capaz de calcular una dirección física a partir de una dirección virtual, aplicando una serie de algoritmos (que no vamos a detallar). Una vez que ha hecho este cálculo, almacena la equivalencia en una tabla interna, para no tener que recalcular de nuevo la misma dirección.

La estructura de la memoria en Win32 es arbórea, esto es: parte de un nodo raí y después se va subdividiendo en grupos más pequeños, hasta que termina en el más pequeño posible: la página de memoria.

El nodo raíz es una tabla llamada “page map” o Mapa de páginas que se subdivide en:

- **Page Directory:**
  Cada proceso cuenta con un solo directorio de páginas, que no es más que una tabla de 1.024 entradas, donde se almacenan los punteros a las 1.024 “Pages Tables” con las que cuenta un proceso.
- **Page Table:**
  Cada tabla de páginas es otra tabla (también de 1.024 entradas) donde se almacenan los punteros de los 1.024 “Pages Frames” que posee cada Tabla de páginas.
- **Page Frame:**
  Es un bloque de 4 Kb (una página) donde se almacena los datos de cada proceso.

En el siguiente esquema se puede ver este "Page Map":



Nosotros, para hacernos una idea y simplificar un poco este jaleo, lo vamos a representar como una tabla lineal, en la que cada fila será una página de memoria (o un “Page Frame” que viene a ser lo mismo). Algo así como se muestra en la siguiente tabla:

Página	Rango de direcciones virtuales	Rango de direcciones físicas	Almacenado en...
56	0x00000123 - 0x00001123	0x0FF231AB - 0x0FF241AB	RAM
102	0x00000FFF - 0x00001FFF	0x0AB124AA - 0x0AB134AA	C:\WINDOWS\Win386.swp
5798	0xA10002FB - 0xA10012FB	0xC000FF12 - 0xC0010F12	Proyección C:\PEPE.DAT
9487	0xCFBB02FB - 0xCFBB12FB	0xAFFABC11 - 0xAFFACC11	RAM
9510	0xAFAB0F37 - 0xAFAB1F37	NULL	NULL
[...]	[...]	[...]	[...]

La primera columna indica la página de memoria que queremos describir.

La segunda columna indica el rango de direcciones virtuales que componen la página. Como una página equivale a 4 KB de memoria, el rango de direcciones irá desde la dirección X hasta la dirección X+0x1000 (en hexadecimal), o lo que es lo mismo: de X hasta X + 4.096 (4 KB).

La tercera columna indica el rango de direcciones de memoria física donde se está almacenada la página. Dicho en un lenguaje más técnico: el rango de direcciones donde se proyecta la página. Una página, como ya hemos visto puede estar en memoria RAM (si tenemos suficiente) o en memoria virtual (en el disco, si el sistema está sobrecargado). Este rango puede ser tanto de direcciones de RAM como de posiciones en el archivo de intercambio o proyectado. Lógicamente, también ocupa 4 KB, por la misma razón que antes.

La última columna indica en qué medio está almacenado el dato, si en RAM (memoria física) o en disco (memoria virtual) o en una proyección de archivo (http://www.lawebdejm.com/?id=21140).

Pero en la última fila, vemos algo raro... ¿qué significa que un dato esté almacenado en NULL? Pues sencillamente eso, que no está almacenado.

Pero eso lo vamos a explicar a continuación.

Operaciones con el espacio de direcciones

En Windows, para tener acceso al espacio de direcciones virtuales es necesario hacer dos operaciones:

1.- Reserva del rango de direcciones

Esta operaciones la realizamos desde nuestros programas, a través de la función VirtualAlloc con la bandera MEM_RESERVE. Esta operación es muy rápida, independientemente de la cantidad de memoria reservada. Lo único que se hace es buscar un bloque de memoria contigua del tamaño especificado, y marcarlo como "Reservado". Supongamos que tenemos una estructura definida del siguiente modo:

```
typedef struct {
    char    Dato1[2048];              // 2048 * 1 = 2048 bytes
    int     Dato2;                    // 4 bytes
    int     Dato3[100];               // 100 * 4 = 400 bytes
    void    *Dato4[100];              // 100 * 4 = 400 bytes
    char    *Dato5[1024];             // 1024 * 4 = 4096 bytes
} MiEstructura, *PMiEstructura;       // Tamaño total: 6948 bytes
```

El siguiente código reserva espacio para almacenar 5 estructuras de tipo MiEstructura (5 * 6948 = 34740 bytes = 33'925 Kb) y almacena en la primera estructura un dato.

```
{
    MiEstructura   *p;

    p = (MiEstructura *) ::VirtualAlloc( NULL, 5 * sizeof(MiEstructura),
                                         MEM_RESERVE, PAGE_NOACCESS );

    if ( p )
        strcpy(p->Dato1, "este es el primer dato");
    else
    {
        TCHAR buff[256];

        wsprintf( buff,
                  "No se han podido reservar %d bytes de memoria virtual.",
                5 * sizeof(MiEstructura) );
        ::MessageBox( NULL, buff, "Error de reserva", MB_ICONSTOP);
    }
}
```

En la primera instrucción, lo único que le estamos pidiendo a Windows es que nos asigne una dirección dentro del rango de direcciones virtuales y marque como ocupado el rango que le indicamos (34.740 bytes). Como ya sabemos, la unidad mínima de memoria con la que trabaja Windows es la página, así que redondeará el tamaño que nosotros le hemos pedido al múltiplo más cercano del tamaño de página:

34.740 bytes = 33'925 KB -> redondeado a 36 KB = 9 páginas de 4 KB cada una

Al asignarnos una dirección, el sistema operativo añade las entradas correspondientes al Mapa de páginas. Su aspecto será como el de la siguiente tabla:

Página	Rango de direcciones virtuales	Rango de direcciones físicas	Almacenado en...
56	0x00000123	0x0FF231AB	RAM
[...]	[...]	[...]	[...]
479	0x00000BAC5	NULL	NULL
480	0x00000CAC5	NULL	NULL
481	0x00000DAC5	NULL	NULL
482	0x00000EAC5	NULL	NULL
483	0x00000FAC5	NULL	NULL
484	0x000010AC5	NULL	NULL
485	0x000011AC5	NULL	NULL
486	0x000012AC5	NULL	NULL
487	0x000013AC5	NULL	NULL

Nota: En vez de escribir el rango de dirección virtual y física, solamente escribo la dirección inicial, ya que la final será 4 KB más adelante

En color rojo aparece nuestra reserva. Aunque vemos que no está almacenado en ningún sitio. Esto es debido a que en Windows, una reserva de dirección no es equivalente a una reserva de espacio físico. Es por esto, que si se llega a ejecutar la segunda instrucción (asignarle un dato a la primera estructura) se produce una violación de acceso (también llamado GPF: General Protectection Failure), ya que esa dirección virtual no tiene dirección física asociada. Por esta razón el último parámetro, que indica la protección de la página, se ha pasado como PAGE_NOACCESS, para que si se intenta acceder a este rango de direcciones, se produzca una violación de acceso.

Hay que tener en cuenta que al reservar espacio, Windows buscará una región de páginas contiguas que estén libres, y si el bloque requerido es muy grande y la memoria está fragmentada, la función VirtualAlloc fallará, devolviendo NULL.

Para conseguir memoria física hay que realizar una segunda operación: comprometer almacenamiento físico.

2.- Compromiso del almacenamiento físico

Una vez que la dirección está reservada, hay que decirle al sistema que nos busque algún lugar donde proyectar esas páginas. Internamente no es posible saber si una página está almacenado en RAM o en el archivo de paginación. Eso es una operación interna de Windows, y las páginas están continuamente pasando de RAM a disco y viceversa, conforme los requerimientos de memoria van cambiando. Para nosotros nos vale con saber que la página estará cuando la necesitemos y Windows ya se buscará la vida para guardarla donde sea.

La operación que hay que realizar para comprometer almacenamiento físico es una nueva llamada a VirtualAlloc, esta vez con la bandera MEM_COMMIT. Para dar más flexibilidad al asunto, la gente de Microsoft nos permite comprometer espacio para algunas páginas de las que previamente hemos reservado. La llamada a VirtuallAlloc para comprometer espacio físico será una operación más lenta cuanto más espacio estemos comprometiendo, así que debe realizarse con cuidado y sólo

en el momento en que sea necesario. En el siguiente código realizamos la misma reserva que en el ejemplo anterior, pero antes de asignar el dato, comprometemos espacio para la primera estructura:

```
{
    MiEstructura *p;
    TCHAR         buff[256];

    p = (MiEstructura *) ::VirtualAlloc( NULL, 5 * sizeof(MiEstructura),
                                         MEM_RESERVE, PAGE_NOACCESS);

    if ( !p )
    {
        if (::VirtualAlloc(p, sizeof(MiEstructura), MEM_COMMIT, PAGE_READWRITE))
            strcpy( p->Dato1, "ahora sí: este es el primer dato en C" );
        else
        {
            wsprintf( buff,
                      "No se ha podido comprometer %d bytes de memoria virtual.",
                      5 * sizeof(MiEstructura));
            ::MessageBox( NULL, buff, "Error de compromiso", MB_ICONSTOP );
        }
    }
    else
    {
        wsprintf( buff,
                  "No se han podido reservar %d bytes de memoria virtual.",
                  5 * sizeof(MiEstructura) );
        ::MessageBox( NULL, buff, "Error de reserva", MB_ICONSTOP );
    }
}
```

Como solamente hemos comprometido espacio para una estructura (6.948 bytes), Windows redondeará al múltiplo de página mayor más cercano y comprometerá espacio para esas páginas.

6.948 bytes = 6'785 KB -> redondeado a 8KB = 2 páginas de 4KB cada una.

Después de haber realizado el compromiso (la segunda llamada a VirtualAlloc) el Mapa de páginas tendrá el aspecto de la siguiente tabla:

Página	Rango de direcciones virtuales	Rango de direcciones físicas	Almacenado en...
56	0x00000122	0x0FF231AB	RAM
[...]	[...]	[...]	[...]
479	0x00000BAC5	0xFF4578A	RAM
480	0x00000CAC5	0xFF4678A	RAM
481	0x00000DAC5	NULL	NULL
482	0x00000EAC5	NULL	NULL
483	0x00000FAC5	NULL	NULL
484	0x000010AC5	NULL	NULL
485	0x000011AC5	NULL	NULL
486	0x000012AC5	NULL	NULL
487	0x000013AC5	NULL	NULL

En color rojo pueden verse los cambios respecto al estado anterior, ya que dos páginas ya tienen almacenamiento físico.

Por último, y como todos sabemos, toda reserva necesita de una liberación, y en nuestro caso esto se hace a través de la función VirtualFree. Esta función nos des-comprometerá y/o liberará una región de páginas de memoria virtual.

Siguiendo con el ejemplo anterior, vamos a completar el algoritmo liberando la memoria que hemos reservado y comprometido:

```
{
    MiEstructura *p;
    TCHAR         buff[256];

    p = (MiEstructura *) ::VirtualAlloc( NULL, 5 * sizeof(MiEstructura),
                                         MEM_RESERVE, PAGE_NOACCESS );

    if ( p )
    {
        if (::VirtualAlloc(p, sizeof(MiEstructura), MEM_COMMIT, PAGE_READWRITE))
        {
            strcpy( p->Dato1, "ahora sí: este es el primer dato en C" );

            if ( !::VirtualFree( p, sizeof(MiEstructura), MEM_DECOMMIT ) )
            {
                wsprintf( buff, "No se ha des-compromoter la memoria virtual.",
                          5 * sizeof(MiEstructura) );
                ::MessageBox(NULL, buff, "Error de de-compromiso", MB_ICONSTOP);
            }
        }
        else
        {
            wsprintf( buff,
                    "No se han podido comprometer %d bytes de memoria virtual.",
                    5 * sizeof(MiEstructura) );
            ::MessageBox( NULL, buff, "Error de compromiso", MB_ICONSTOP );
        }

        if ( !::VirtualFree( p, 5 * sizeof(MiEstructura), MEM_RELEASE ) )
        {
            wsprintf( buff, "No se ha liberar la memoria virtual.",
                      5 * sizeof(MiEstructura) );
            ::MessageBox( NULL, buff, "Error de liberación", MB_ICONSTOP );
        }
    else
    {
        wsprintf( buff,
                  "No se han podido reservar %d bytes de memoria virtual.",
                  5 * sizeof(MiEstructura) );
        ::MessageBox( NULL, buff, "Error de reserva", MB_ICONSTOP );
    }
}
```

Hay que tener en cuenta que si intentamos des-comprometer una página que no ha sido previamente comprometida, la función fallará. Para saber la situación de una página concreta, podemos hacer uso de la función VirtualQuery.

Distribución del espacio de direcciones virtuales

El espacio de direcciones se divide en distintas secciones, aunque dependiendo del sistema operativo utilizado, estos rangos tendrán una utilidad u otra. Básicamente hay dos familias de sistemas operativos Win32: La familia Windows 95 (que la componen la versión 95, 98 y Millenium) y la familia NT, compuesta por la versión NT, 2000 y XP

Región 1

Familia 95: de 0x00000000 a 0x003FFFFF

Familia NT: de 0x00000000 a 0x00000FFF

Región privada para permitir la compatibilidad con aplicaciones de MS-DOS. En MS-DOS no era posible escribir dentro de los primeros 4Kb de memoria, así que cuando diseñaron Windows tuvieron que continuar con la misma limitación. Un intento de lectura/escritura sobre las primeros 4 Kb (desde 0x0000 a 0xFFFF) provocará una violación de acceso. Esa región también es privada para detectar la utilización de punteros nulos, ya que intentarían escribir en la dirección 0.

Región 2

Familia 95: de 0x00400000 a 0x7FFFFFFF

Familia NT: de 0x00001000 a 0x7FFEFFFF

En esta región es donde se almacenan los datos privados de cada proceso. Como podemos ver, la cantidad de memoria que puede llegar a utilizar una aplicación no son 4 GB como dijimos en principio, sino que son aproximadamente 2 GB.
En la familia NT se almacenan tanto el código y datos de los ejecutables como las DLL del sistema y de usuario. Cualquier variable que utilicemos desde nuestros programas estará en este rango.

Región 3

Familia 95: de 0x80000000 a 0xFFFFFFFF

Familia NT: de 0x8000000 a 0xFFFFFFFF

Región privada donde se carga el kernel del sistema operativo y los controladores de dispositivos (archivos .VXD).
En Windows 95 además, esta región alberga las DLLs y los archivos proyectados en memoria. Otra diferencia entre Win95 y NT es que para el primero, la región es accesible por el proceso, y el segundo provocará una violación de acceso si se intenta leer/escribir en este rango de direcciones.

Otras funciones

Existen otras funciones que nos permiten realizar otras tareas con la memoria virtual:

- **VirtualLock**
  Bloquea un grupo de páginas (previamente comprometidas) en memoria física, evitando que sean pasadas al archivo de intercambio cuando el sistema necesita memoria RAM. Esta operación sólo es recomendable para drivers que necesitan tener un control absoluto sobre la memoria y asegurarse que un dato siempre va a estar en RAM, para situaciones más normales, debemos dejar que sea Windows el que decida si una página debe estar en RAM o en el archivo de paginación.

- **VirtualUnlock**
  Desbloquea un grupo de páginas que han sido previamente bloqueadas con VirtualLock.

- **VirtualProtect**
  Cambia el estado de protección de un grupo de páginas del proceso activo, por ejemplo para asegurarnos que ningún "puntero loco" cambie el contenido de esas páginas de memoria, o para detectar en qué punto de un programa se está modificando cierta región de memoria.

- **VirtualProtectEx**
  Cambia el estado de protección de un grupo de páginas de un proceso dado.

- **VirtualQuery**
  Consulta el estado y el tipo de un grupo de páginas del proceso activo.

- **VirtualQueryEx**
  Consulta el estado y el tipo de un grupo de páginas de un proceso dado.

Aplicación práctica

La utilidad de todo esto es continua. Siempre que reservamos memoria desde nuestros programas (ya sea con GetMem, malloc, declarando una variable, creando un objeto o de cualquier otra forma), se notifica al Kernel del sistema operativo de esa operación. Internamente, el Kernel traduce cualquier reserva de memoria en una llamada a VirtualAlloc, ya que toda la memoria manejada desde Win32 es virtual. Otro uso continuo de la memoria virtual es en la implementación que hace Windows de la pila (stack), utilizando páginas de guarda para detectar un posible desbordamiento. De esto hablaremos en el próximo artículo (www.lawebdejm.com/?id=22102)

También es posible que nosotros necesitemos llamar directamente a la función VirtualAlloc, normalmente para reservar grandes bloques de memoria que necesitamos manipular a través de un puntero. Un ejemplo muy conocido de esto es cualquier hoja de cálculo. Cada celda tiene asociada una estructura donde se almacenan sus datos (el contenido, el formato, la fórmula, etc.). Como una hoja de cálculo es una matriz de N x M, lo lógico es representarlo como un array bidimensional de N x M, siendo cada elemento del tipo de la estructura. Pero lo más habitual es que sólo se opere con las primeras celdas, no con todas (Excel admite hasta una matriz de 65.536 filas y otras tantas columnas), así que la reserva se hace inicialmente, pero el compromiso de las celdas se hace según se van necesitando. Otra utilidad parecida podría ser un calendario compuesto de 365 páginas (una por día). Cada página del calendario se comprometerá sólo cuando sea necesario su uso (cuando tengamos alguna cita para ese día o sea necesaria su información).

Conclusión

En este artículo hemos visto como la plataforma Win32 maneja la memoria virtual, un aspecto muy importante que influye en cualquiera de nuestros programas, ya que cualquier reserva de memoria que hagamos será traducida por el Kernel a una reserva de memoria virtual.

Hemos visto como una petición de memoria virtual se compone de dos pasos: reserva y compromiso y cómo es posible reservar cierto espacio pero comprometer sólo parte de este.

También hemos visto cómo se distribuye el espacio de direcciones virtuales de un proceso, diferenciando las distintas regiones y su finalidad.

Y por último hemos dado algunas funciones auxiliares para manipular la memoria virtual y algunas utilidades prácticas de la memoria virtual.

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