Los rincones del API Win32
Los montones

Lo prometido es deuda, así que en esta ocasión vamos a estudiar otra de las estructuras de memoria en la arquitectura Win32: los montones.

Índice

Introducción

Como hemos ido viendo en los artículos anteriores, los programas Win32 utilizan memoria virtual en todos los almacenamientos volátiles, y lo hacen a través de varias estructuras de memoria, entre ellas la pila y el(los) montón(es).

La pila, como ya vimos en el anterior artículo (www.lawebdejm.com/?id=21102), almacena las variables locales que se van utilizando durante la ejecución del programa y cada hilo posee su propia pila.

Sin embargo es posible almacenar variables en otro lugar: en el montón, montículo o "free store" (utilizaré la primera acepción ya que creo que es una palabra más coloquial).

Definición

El montón es una estructura de datos especial, ya que no establece ningún orden de entrada o salida de sus elementos (como sí lo hacen pilas o colas).

El montón se trata de una zona de memoria reservada donde se van almacenando variables, estructuras, buffers, objetos, etc.

El tamaño de esta zona de memoria suele ser mayor que el de la pila, y además, definido por el programador y ampliable en cualquier momento.

Los elementos que se almacenan en el montón pueden ser de distintos tipos y tamaños, aunque eso sí: el único requisito es que *sólo es posible acceder a ellos a través de un puntero*.

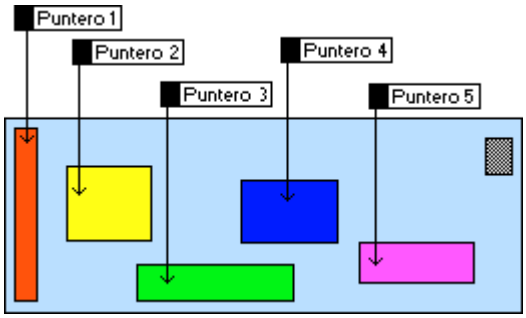
La piscina

Por poner un ejemplo, podemos decir que el montón es como una piscina donde nadan peces de distintos tipos y tamaños. Si queremos "acceder" a algún pez, debemos pescarlo con una caña, es decir, debemos conseguir un puntero al pez. Todo pez que esté apuntado por su caña (que esté pescado) será accesible, mientras que si un pez no está pescado, podemos decir que está fuera de control y no sabremos ni donde está, ni cómo son sus características, ni mucho menos, sacarlo de la piscina y cocinar un buen besugo al horno.

Este es un símil bastante aproximado de lo que es un montón en realidad: *una zona de memoria previamente reservada, donde se almacenan elementos heterogéneos, cada uno de un tamaño y naturaleza distinta*. Si queremos acceder a cada uno de esos elementos, debemos tenerlo localizado, y la única forma de hacer es a través de los punteros.

Ambas opciones son válidas, pero... sólo si el sistema operativo realmente nos da acceso a esa zona de memoria.

Siguiendo con el símil, si queremos almacenar una variable "integer" en el montón, este será nuestro pez (la variable integer), y nuestra caña deberá ser una variable puntero. La forma de “pescar” al integer es que la variable puntero almacene la dirección donde resida la variable integer, o dicho de otra forma: que el puntero apunte a la variable integer.



En la figura de la izquierda podemos ver un esquema de este símil. La zona azul representa el montón (la piscina) y cada uno de los elementos contenidos (peces) es accesible a través de un puntero (una caña de pescar). En el ejemplo también vemos que uno de los elementos (el de más a la derecha) no está apuntado, así que no es accesible. Las variables puntero (las que están fuera del montón) a su vez pueden estar almacenadas en la pila, en otro montón (incluso en el mismo), en la zona de datos estáticos, o en cualquier otro lugar de la memoria.

Voy a asumir que todo el que lea este artículo entiende bien el concepto de puntero y tiene cierta soltura en su manejo. Si no es así, sugiero que pongáis una consulta en cualquiera de los foros que hay en internet.

Utilización del montón

El montón no es algo exclusivo de la arquitectura Win32 (como tampoco era la pila), sino que es un elemento que se ha utilizado en prácticamente todas las plataformas y lenguajes de programación.

Quizá a algunos de vosotros os suene la frase *"asignación dinámica de memoria"*. Esta frase es equivalente a decir: *"crear memoria en el montón"*, ya que toda asignación de memoria que se haga en ejecución, se hará en esta estructura de datos.

Su función principal es almacenar aquello que no podía ser almacenado en la pila: variables cuyo tamaño no es conocido en tiempo de compilación o variables demasiado grandes para que quepan en la pila.

Como ya vimos en el artículo anterior, para almacenar una variable, su tamaño debía ser conocido en tiempo de compilación. Esto es necesario para que el compilador vaya "colocando" cada variable en su sitio (una encima de otra en la pila), ya que si no sabe lo que ocupa una variable, no podrá saber la posición de la siguiente. Por eso sólo es posible definir variables y vectores de tamaños conocidos en tiempo de compilación.

Sin embargo, si queremos definir una zona de memoria o vector de tamaño variable, hay que actuar de otra forma.

La siguiente función crea una zona de memoria de tamaño variable en el montón (un pez de peso variable)

```
void CrearPez( int peso )
{
    void    *caña;

    caña = malloc( peso );
    ZeroMemory( caña, peso );
}
```

Como vemos, la función es muy sencilla y consta de dos pasos:

- Reservar un bloque de tamaño variable en memoria. Este bloque de memoria, **reside en el montón** ya que su tamaño no se conocía en tiempo de compilación. Para reservar el bloque (el pez) se utilizan una de las funciones que nos proporciona C para la asignación dinámica de memoria: malloc, la cual retorna un puntero (la caña de pescar) al bloque que ha sido reservado en el montón.
- Rellenar este bloque de memoria con ceros a través de la función del API ZeroMemory.

Sin embargo se cometen un gran error: el puntero retornado por malloc() se almacena en una variable que reside en la pila, la cual quedará descartada al finalizar la función, por lo que el "pez" que hemos creado queda libre e inaccesible. El "pez" en cuestión tendría el mismo aspecto que el que aparece en el lazo derecho de la **Figura 1**: completamente perdido y sin posibilidad de ser eliminado (se nos ha escapado el muy condenado).

Este error (más común de lo que puede parecer) se denomina "goteo de memoria" o "memory leak" y desemboca en que, si esta situación se repite múltiples veces, podemos quedarnos sin memoria en el montón, ya que estaría lleno de objetos a los que no podemos acceder para eliminarlos. Estos objetos quedaría eliminados al terminar el proceso, porque la memoria de un proceso se libera completamente al terminar este.

Ahora vamos a ver el mismo código, pero con los errores corregidos:

```
void CrearPez( int peso )
{
    void    *caña;

    caña = malloc( peso );
    ZeroMemory( caña, peso );

    free( caña );
}
```

Vemos que antes de terminar la función, liberamos el bloque de memoria que teníamos creado en el montón. Después de esto, ya no importa si la variable “caña” desaparece, porque el valor que contiene no es válido ni necesario.

Este es un caso típico de asignación de memoria en el montón. Como norma general, y aprovechando la vena poética, se puede decir que: *"toda variable que se crea y destruye en ejecución, reside en el montón"*.

Ahora vamos a ver un ejemplo un poco más complejo:

```
void CrearBancoPeces( int peso, int NumeroPeces )
{
    void   *caña;
    int     i;

    for ( i = 0; i < NumeroPeces; ++i )
    {
        caña = malloc( peso );
        ZeroMemory( caña, peso );
    }

    free( caña );
}
```

En esta función, en vez de crearse un pez solitario, se crea un conjunto de peces. Si analizamos con cuidado, veremos que hay un "pequeño gran" bug: al dar la primera vuelta en el bucle, se crea el

primer pez, y automáticamente queda pescado a través de la variable "caña". Al dar la segunda vuelta, se crea el segundo pez, y lo que hacemos es pescar con nuestra “caña” a este segundo pez. ¿y el primero? Pues ha quedado libre como el viento. Hemos dejado libre al primer pez para pescar al segundo. Así se continuará con el bucle hasta que tengamos la piscina (el montón) llena a rebosar de peces, pero sólo tendremos pescado al último. A la hora de liberar memoria, sólo liberaremos la del último elemento, ya que todos los demás no está localizados. Una vez más, hemos caído en otro "goteo de memoria".

Como ejercicio para manejo de punteros, sugiero que se realice el mismo código, pero corrigiendo el bug (una sugerencia: hace falta una caña de pescar por cada pez).

El montón y los objetos

Ahora vamos realizar lo mismo que antes, pero aplicando la orientación a objetos:

```
void CrearBesugo( void )
{
    CBesugo    *besugo;

    besugo = new CBesugo;
    besugo->Nada;

    delete besugo; // bye bye
}
```

Como vemos, se define una variable (que en realidad es un objeto de una clase), se crea, se llama a un método del objeto, y se destruye.

Esto resulta familiar ¿no?. Pues sí, los objetos, al crearse dinámicamente en tiempo de ejecución, también residen en el montón. Y para acceder a ellos se hace a través de un puntero como no podía ser de otra forma.

Si hubiésemos olvidado liberar el objeto (sin hacer delete), caeríamos de nuevo en otro “goteo de memoria”.

Algún alumno aventajado puede decir lo siguiente: "y si el tamaño de la clase es conocido en tiempo de compilación ¿porqué no se almacena el objeto en la pila, como, por ejemplo, un registro o estructura?"

Bueno, esta es la pregunta del millón.

Es muy cierto que el tamaño de la clase se conoce durante la compilación, y técnicamente es posible almacenar estos objetos en la pila, sin embargo se utiliza el montón por razones de rendimiento y eficiencia. Un objeto suele ocupar bastante memoria (como mínimo la suma del tamaño de sus atributos, más un puntero a la definición de la clase), así que si definimos un par de objetos dentro una función, y estos se almacenasen en la pila, lo más probable es que esta se desbordase, ya que, como todos sabemos, el tamaño de la pila es limitado y fijo.

Por eso, en la mayoría de la ocasiones, los objetos se almacenan en el montón, ya que el tamaño del montón es mucho mayor que el de la pila.

Como habéis visto, he dicho “en la mayoría de las ocasiones” porque siempre hay una excepción. En C es posible hacer esto, aunque en otros lenguaje como Pascal el código equivalente sería imposible:

```
void CrearBesugo( void )
{
    besugo    CBesugo;  // ya no es puntero

    besugo.Nada;
}
```

Y lo que estamos haciendo es almacenar nuestro "besugo" en la pila. Como vemos, ya no definimos una variable puntero, ni utilizamos el operador de indirección "->", sino que tratamos a nuestro besugo como si de una estructura o registro se tratara. Del mismo modo, tampoco es necesaria la creación ni destrucción del objeto, ya que es la pila la encargada de reservar su espacio al iniciar, y descartarlo al finalizar. El que no entienda el porqué de esto, le sugiero que lea el artículo de “La Pila” (www.lawebdejm.com/?id=21102).

Esta técnica es bastante utilizada en programas en C++, siempre que los objetos a crear no sean demasiado grandes, ya que podrían desbordar la pila. La principal razón para el uso de esta técnica es la rapidez de ejecución (la pila es bastante más rápida que el montón) y comodidad (ya no hay que preocuparse de la creación y destrucción de los objetos). Sin embargo, para hacer esto, hay que saber bien lo que se hace (y nosotros ya estamos en condiciones de hacerlo).

Hasta ahora hemos dado una descripción bastante detallada de los usos del montón, sin embargo, toda esta información ha sido genérica, y en ningún momento nos hemos centrado en las peculiaridades de la plataforma Win32.

Los montones en Win32

Como ya vimos, cuando se inicia una aplicación a través de la función CreateProcess, se crea el hilo principal (con su pila) y el espacio de direcciones virtuales. Otra de las tareas que se hacen es crear el **montón por defecto del proceso**.

Es decir, un proceso cuenta con un montón para realizar las operaciones de asignación dinámica de memoria, al igual que cada hilo cuenta con su propia pila para almacenar las variables locales que va necesitando. Esto significa que la memoria del montón está dentro del espacio de direcciones de un proceso, por lo que puede ser accedida desde cualquier punto del mismo (desde el hilo principal, hilos adicionales, DLLs externas, etc.) Pero vamos por partes.

La situación más sencilla es que cada proceso cuente con un sólo montón (el montón por defecto), sin embargo, en cualquier momento se pueden crear montones adicionales (o dinámicos), para un uso más específico. Más adelante explicaremos en qué situaciones es conveniente utilizar montones adicionales y qué ventajas conseguimos con esto.

¿Y por qué se utilizan montones en Win32?

Pues como ya hemos visto en una sección anterior, por la misma razón que en cualquier otra plataforma: para almacenar variables, objetos y otras estructuras creadas en tiempo de ejecución, o dicho de un modo más técnico: para la asignación dinámica de memoria.

Adicionalmente, también sirve para simplificar el trabajo del programador, por las siguientes razones:

1. Se le abstrae de la gestión de la memoria subyacente (en el caso de Win32, la gestión de la memoria virtual, de la que ya hablamos en www.lawebdejm.com/?id=21101).
2. Se le asegura que la memoria estará cuando la necesite, sin tener que preocuparse de conceptos como el compromiso físico.
3. Se le evita entrar en detalles de implementación como el tamaño de página, los redondeos en las peticiones de memoria y otras aspectos que desvían la atención de programador de su problema.

Además, a la hora de gestionar muchos objetos de distintos tamaños, se aprovecha el espacio mucho más eficientemente con montones que con memoria virtual. Más adelante explicamos en profundidad la implementación interna del montón, así que daremos más detalles sobre el rendimiento.

Manos a la obra, o mejor dicho: manos al montón

Bueno, ahora que conocemos el concepto de montón y sus ventajas, vamos a ver con qué funciones podemos manejar los montones.

Vamos allá:

Creación del montón

La creación de cualquier montón (ya sea el montón por defecto o cualquier otro) se realiza a través de la función HeapCreate:

```
HANDLE  HeapCreate(
             DWORD  flOpciones,          // banderas de reserva
             DWORD  dwTamañoInicial,     // tamaño inicial del montón
             DWORD  dwTamañoMaximo );    // máximo tamaño del montón
```

Esta función crea un montón de un tamaño especificado y nos retorna un descriptor (handle) para acceder a él posteriormente.

Vamos a ver para qué sirve cada parámetro:

1. **flOpciones**: Es posible indicar cualquier combinación de las siguientes banderas:
    - HEAP_GENERATE_EXCEPTIONS: cuando cualquier función de manejo de montones falle al acceder a este montón, se generará una excepción del sistema, en vez de retornar NULL. Las posibles excepciones son STATUS_ACCESS_VIOLATION y STATUS_NO_MEMORY.
    - HEAP_NO_SERIALIZE: desactiva el acceso sincronizado al montón. El gestor de montones utiliza un sistema de sincronización de hilos para asegurarse de que los datos son siempre consistentes cuando varios hilos acceden simultáneamente a un mismo montón. Este sistema ralentiza algo las operaciones de reserva y liberación en el montón, por lo que podemos desactivar esta sincronización sólo en las siguientes situaciones:
        - Si el proceso sólo va a tener un hilo.
        - Si el proceso tiene más de un hilo pero sólo uno de ellos accede al montón
        - Si el proceso tiene más de un hilo y varios acceden al montón, pero se gestiona el acceso sincronizado manualmente.
    - HEAP_ZERO_MEMORY: indica que después de todas las asignaciones de memoria realizadas en este montón, se inicializará el contenido con ceros.
    - HEAP_GROWABLE (indocumentado): Indica el montón puede crecer automáticamente cuando su espacio se agote. Para configurar este comportamiento, se debe pasar un 0 en el parámetro dwTamañoMaximo, en vez de emplear esta bandera, que es privada del sistema.
    - HEAP_REALLOC_INPLACE_ONY (indocumentado): indica que cualquier reasignación de memoria de este montón (a través de la función HeapReAlloc) se hará sin mover los bloques de su situación actual.

- HEAP_DISABLE_COALESCE_ON_FREE (indocumentado): Cuando se libera un bloque del motón, automáticamente el gestor intenta refundir bloques libres que estén adyacentes a este, para así crear un bloque más grande y evitar fragmentación. Esta bandera desactiva este comportamiento, lo cual acelera la ejecución.
- HEAP_CREATE_ALIGN_16 (indocumentado): Crea los bloques del montón con una alineación de 16 bytes.
- HEAP_FREE_CHECKING_ENABLED: indocumentado y desconocido.
- HEAP_CREATE_ENABLE_TRACING: indocumentado y desconocido.
- HEAP_TAIL_CHECKING_ENABLED: indocumentado y desconocido.
- HEAP_MAXIMUM_TAG: indocumentado y desconocido.
- HEAP_PSEUDO_TAG_FLAG: indocumentado y desconocido.
- HEAP_TAG_SHIFT: indocumentado y desconocido.

2. **dwTamañoInicial**: indica el tamaño (en bytes) que inicialmente contará con compromiso físico (más adelante veremos que el montón se almacena en el sistema de memoria virtual, por lo que cumple las mismas normas que cualquier otro bloque reservado con VirtualAlloc). Este tamaño podrá ser todo lo grande que queramos (con la limitación física de la máquina) y se redondeará al tamaño de página inmediatamente superior. Cuanto mayor sea este valor, más tardará la función en realizar la creación del montón.

3. **dwTamañoMaximo**: indica el tamaño máximo (en bytes) que podrá albergar el montón. Además, indica el tamaño del bloque que se reservará (pero no comprometerá) al crear el montón. Si en este parámetro se especifica el valor 0, el espacio reservado inicialmente en el montón será el mismo que el indicado en dwTamañoInicial, sin embargo, este crecerá conforme vaya necesitando más espacio, hasta que el gestor de memoria virtual no cuente con más recursos. Si hemos especificado un tamaño máximo (dwTamañoMaximo > 0), contamos con una pequeña limitación: no podemos reservar bloques de memoria en el montón mayores a 524.280 bytes. Esta limitación no existe si el montón es auto-extensible (dwTamañoMaximo = 0).

Si la función tiene éxito, se devuelve un descriptor (handle) del montón, que será utilizado en sucesivas llamadas.

Si ocurre algún error la función retornará nulo o lanzará una excepción del sistema, si se indicó la bandera HEAP_GENERATE_EXCEPTIONS.

Asignación de memoria

Una vez que el proceso ha creado el montón, es posible asignar su memoria para poder ser utilizada.

Esta tarea, como todas las relacionadas con la memoria dinámica, es relativamente lenta, así que tendremos que tener cuidado con su uso.

Del mismo modo, y debido a la implementación interna del montón, debemos tomar ciertas medidas para evitar en lo posible la fragmentación. Una de estas medidas, es reservar siempre bloques de un tamaño que sea potencia de 2: 1 byte, 2 bytes, 4 bytes, 8 bytes, 128 bytes... de este modo nos estamos asegurando que el bloque van a crearse de una serie de tamaños fijos (la lista de todas las potencias de 2), y que podrán ser reutilizados en operaciones posteriores.

Esa tarea la podremos realizar a través de la función HeapAlloc:

```
LPVOID  HeapAlloc(
                HANDLE hMonton,          // descriptor del montón
                DWORD  flOpciones,       // banderas de asignación
                DWORD  dwTamaño);        // bytes a asignar
```

Los parámetros tienen los siguientes significados:

1. **hMonton**: indica el descriptor del montón donde queremos realizar la asignación de memoria. Este descriptor se suele obtener en la llamada a HeapCreate. Más adelante hablamos con detalle cómo conseguir un descriptor de montón por defecto del proceso.

2. **dwOpciones**: indica las banderas con las que se configura la operación. Todas estas banderas tienen el mismo significado que el indicado en la función HeapCreate, pero aplicado sólo a esta llamada:
   - HEAP_GENERATE_EXCEPTIONS
   - HEAP_NO_SERIALIZE
   - HEAP_ZERO_MEMORY

3. **dwTamaño**: El número de bytes a asignar. Si al crear el montón se indicó un tamaño máximo (dwTamañoMaximo > 0), este parámetro tiene que ser menor que 524.280.

Esta función retornará un puntero a la zona de memoria asignada. HeapAlloc nos garantiza que esta zona está reservada y comprometida y es **contigua**.

Del mismo modo, HeapAlloc también nos garantiza que el tamaño del bloque reservado y comprometido es *al menos* del tamaño indicado en dwTamaño. Para averiguar el tamaño real que ha reservado, podemos hacer una llamada a HeapSize:

```
DWORD  HeapSize(
             HANDLE   hMonton,          // descriptor del montón
             DWORD    flOpciones,       // banderas
             LPCVOID  lpBloque);        // puntero al bloque
```

Esta función nos retorna el tamaño real del bloque de memoria apuntado por lpBloque dentro del montón indicado por hMonton. La única bandera que podemos indicar en el parámetro flOpciones es HEAP_NO_SERIALIZE. Normalmente, el parámetro lpBloque no es más que el valor retornado por HeapAlloc y HeapReAlloc, para averiguar el tamaño exacto que asignaron o reasignaron estas funciones.

Re-asignación de memoria

Ya hemos visto como se crea un montón, y se reserva un bloque en él para ser utilizado posteriormente a través del puntero que nos retorna. Puede ser que, en cierto momento, el tamaño que inicialmente hemos asignado, se quede demasiado pequeño, por lo cual debemos ampliar este bloque. A esta operación se le denomina "reasignación".

La re-asignación es una operación delicada, ya que fragmenta la memoria del montón con mucha facilidad (ver más abajo cuando hablo sobre la implementación). Como todos podréis imaginaros, no es lo mismo reservar un bloque inicial de 2 KB e ir re-asignándolo varias veces, hasta que pueda albergar 100 KB, que reservar directamente un bloque inicial de 100 KB.

Para evitar esta situación, y como norma general, una de las primeras recomendaciones que se hace durante el primer curso de programación, es que siempre se reasigne un valor que sea el doble del tamaño actual del bloque. Es decir: si inicialmente hemos reservado un bloque de 16 KB, a la hora de re-asignarlo, debemos hacerlo de 32 KB, aunque en realidad con 17 KB nos sea suficiente. De este modo, conseguimos mantener controlada la fragmentación del montón, y nunca desperdiciaremos más del 50% del total de la memoria (esto es demostrable matemáticamente).

Para ello existe la función HeapReAlloc. Esta función, además de permitirnos cambiar el tamaño de un montón, nos ofrece la posibilidad de cambiar las opciones de creación.

El definición de la función es la siguiente:

```
LPVOID  HeapReAlloc(
             HANDLE   hMonton,          // descriptor del montón
             DWORD    flOpciones,       // banderas de re-asignación
             LPVOID   lpBloque,         // puntero al bloque
             DWORD    dwTamaño);        // nuevo tamaño del bloque
```

Los parámetros tienen los siguientes significados:

1. **hMonton**: indica el descriptor del montón donde queremos realizar la re-asignación de memoria. Este descriptor se consigue en la llamada a HeapCreate. Más adelante hablamos con detalle cómo conseguir un descriptor de montón del proceso.

2. **flOpciones**: indica las nuevas banderas que establecerán al montón. Estas banderas, sobrescriben a las que se indicaron durante la creación con HeapCreate:

    - HEAP_GENERATE_EXCEPTIONS: igual que en HeapCreate.
    - HEAP_NO_SERIALIZE: igual que en HeapCreate.
    - HEAP_ZERO_MEMORY: Si se amplia el bloque, los bytes adicionales se rellenarán con ceros.
    - HEAP_REALLOC_INPLACE_ONLY: no permite que se mueva el bloque completo cuando se amplía su tamaño. Si no hay bloques contiguos libres para poder ampliar el espacio, la función fallará. Esta bandera es útil cuando dentro de este bloque de memoria hay elementos que están apuntados por distintas variables punteros. Si se permitiese la recolocación del bloque completo, el valor de dichos punteros ya no sería el correcto, por lo que ya no podríamos acceder a esos elementos apuntados.

3. **lpBloque**: es un puntero al inicio del bloque que queremos re-asignar. Normalmente, este valor proviene de una llamada previa a HeapAlloc.

4. **dwTamaño**: Nuevo número de bytes que ocupará el montón. Este tamaño puede ser mayor, menor o igual al tamaño indicado en la creación., sin embargo, si al crear el montón se indicó un tamaño máximo (dwTamañoMaximo > 0), este parámetro tiene que ser menor que 524.280.

Esta función retornará un puntero a la zona de memoria donde comienza el bloque re-asignado. Este puntero, será igual que el parámetro lpBloque si se utilizó la bandera HEAP_REALLOC_INPLACE_ONLY o puede ser completamente distinto si ha necesario resituar el bloque para ampliarlo.

Todo lo reservado debe liberarse…

Como es lógico pensar, toda memoria asignada (o reservada), debe liberarse. Para ello contamos con la función HeapFree:

```
BOOL  HeapFree(
          HANDLE   hMonton,        // descriptor del montón
          DWORD    flOpciones,     // banderas de liberación
          LPVOID   lpBloque);      // puntero al bloque
```

Los parámetros son sencillos: el montón de donde se libera el bloque, las opciones de liberación (sólo se admite HEAP_NO_SERIALIZE) y un puntero al inicio del bloque que se quiere liberar.

Lo más importante que hay que saber de esta función, es que simplemente se marcará el bloque como disponible, pero ocurrirá lo mismo que ocurría con la pila: la memoria ni se libera ni se descompromete, sino que se deja reservada y comprometida para ahorrar tiempo en sucesivas asignaciones. De este modo se consigue que el montón responda rápidamente, excepto cuando se asigne un bloque por primera vez.

En ciertas situaciones de escasez de memoria, Windows se reserva el derecho de eliminar el compromiso físico de aquellos bloques del montón que están reservados. Debido a esto, no tenemos ninguna manera de saber cuando un bloque va a perder su compromiso físico, ya que esta decisión depende del algoritmo que haya utilizando Microsoft (que como era de esperar, no es público).

Lo que sí podemos hacer es eliminar el compromiso físico de todos los bloques libres, a través de la función HeapCompact (descrita más adelante).

La función retornará un booleano indicando si se ha ejecutado correctamente.

… y todo lo creado debe destruirse

Y del mismo modo, si hemos creado un montón a través de la función HeapCreate, debemos destruirlo con la función HeapDestroy.

Hay que tener en cuenta que sólo debemos destruir los montones que creemos nosotros, y *no el montón por defecto del proceso*.

```
BOOL  HeapDestroy(
          HANDLE    hMonton);  // descriptor del montón
```

Esta función, sí que realiza una liberación física de la memoria, tanto los bloques reservados como los que cuentan con compromiso físico. En realidad lo que se hace es una llamada a VirtualFree, indicando las banderas MEM_RELEASE y MEM_DECOMMIT.

Esta función retorna un booleano indicando si el montón se ha destruido correctamente.

Información sobre descriptores de montones

Para realizar cualquier operación con un montón, necesitamos su descriptor. Si se trata de un montón creado por el usuario, simplemente con haber guardado el valor retornado por HeapCreate, ya tenemos el descriptor. Sin embargo, si queremos acceder al montón del proceso, tendremos que hacer uso de la siguiente función:

```
HANDLE GetProcessHeap( void );
```

Esta función es tan sencilla como que se la llama sin parámetros y nos retorna el descriptor del montón por defecto de proceso, o nulo si ocurre un error. Ni más, ni menos.

Sin embargo, como dicen en mi pueblo: no todo el monte es orégano, así que Win32 nos ofrece un método para obtener los descriptores de todos los montones asociados a un proceso (el montón por defecto y cualquier otro que se haya creado).

Esta función es la siguiente:

```
DWORD GetProcessHeaps(
                DWORD    NumeroMontones,    // número máximo montones
                PHANDLE  Montones );        // lista de montones
```

El uso de esta función es como sigue: debemos crear un vector de descriptores, declarándolo de tamaño fijo o bien creándolo dinámicamente. Este vector es el segundo parámetro que pasamos.

En el primer parámetro indicamos el tamaño de este vector, para que la función sepa el número máximo de descriptores que nos puede "retornar" a través del vector. Si el proceso desde que se llama, tiene más montones que el valor indicado por "NumeroMontones", tan sólo nos copiará los que quepan.

Esta función retorna siempre el número total de montones que cuenta el proceso, para que podamos crear un vector dinámico del tamaño exacto.

El método correcto de llamar a esta función es el siguiente:

1. Llamar a la GetProcessHeaps indicando NumeroMontones = 0 para que nos retorne el total de montones que cuenta el proceso.
2. Direccionar un vector del tamaño obtenido por la anterior llamada a GetProcessHeaps.
3. Volver a llamar a GetProcessHeaps indicando ahora el NumeroMontones correcto y pasando el vector dinámico.
4. Recorrer el vector para ir obteniendo cada uno de los descriptores obtenidos. El primer elemento de este vector corresponde al montón por defecto, así que será el mismo valor que el devuelto por GetProcessHeap.

Este algoritmo se implementa del siguiente modo:

```
{
    HANDLE  *vector = NULL;
    DWORD    i;
    DWORD    NumeroMontones;
    char     DescMonton[255] = "";
    char     descriptor[255] = "";

    // obtener el número total de montones
    NumeroMontones = ::GetProcessHeaps( 0, vector );

    // crear un vector dinámico a través de punteros
    vector = new HANDLE[NumeroMontones];

    // se obtienen la lista de descriptores
    ::GetProcessHeaps( NumeroMontones, vector );

    for ( i = 0; i < NumeroMontones; i++ )
    {
        if ( i == 0 )
            strcpy( DescMonton, "Montón por defecto" );
        else
            wsprintf( DescMonton, "Montón número %i", i + 1);

        wsprintf( descriptor, "%lu", vector[i] );
        ::MessageBox( GetActiveWindow(),
                      descriptor, DescMonton, MB_ICONINFORMATION );
    }

    delete [] vector;
}
```

De este modo nos aparecerá en pantalla un MessageBox por cada uno de los montones del proceso, siendo el primero de ellos el descriptor del montón por defecto.

Una vez que hemos obtenido el descriptor de cada uno de los montones, podemos obtener cierta información de ellos a través de las funciones Heap32ListFirst y Heap32ListNext, o bien alguna de sus evoluciones: HeapQueryInformation (a partir de Window XP) o HeapWalk (a partir de Windows NT/2000) para obtener más información sobre el montón o sobre cada uno de los bloques que lo componen.

Otras funciones

Ya hemos visto las funciones más típicas para el uso de montones. De todas formas, existen otras, que vamos a explicar a continuación:

HeapCompact

Se recorre la lista de bloques libres (*free chain*) intentando fundir los bloques pequeños que han sido fragmentados, en los bloques originales más grandes. Esta función nos retorna el tamaño del bloque reservado, comprometido y contiguo más grande que existe en el montón, aunque la gente de Microsoft no nos garantiza que una asignación de este tamaño (con la función HeapAlloc) se ejecute correctamente.

Además, esta función elimina el compromiso físico de todos los bloques libres, por lo que en siguientes asignaciones de memoria, se tendrá que comprometer nuevamente espacio físico.

HeapValidate

Recorre la lista enlazada de bloques de memoria que compone un montón (o un solo bloque), verificando que cada uno de estos bloques sea consistente.

HeapLock y HeapUnlock

Bloquea o desbloquea el montón especificado para que sólo pueda ser accedido por el hilo que llama a estas funciones. Estas funciones permiten realizar la sincronización manual entre hilos.

Las funciones GlobalX y LocalX

Estas funciones se incluyen por compatibilidad con las versiones de 16 bits de Windows. El sistema de gestión de montones es, quizá, el que más cambios sufrió en su implementación en Win32. Uno de los principales fue que en Windows 3.1, existía un montón en el proceso (el montón local) y otro común a todos los procesos (el montón Global). Para manejar estos montones se incluyeron dos grupos de funciones: las que manejaban el montón local (cuyo nombre comenzaba con Local) y las que manejaban el montón compartido (cuyo nombre comenzaba con Global).

En Win32 no tiene sentido esta separación, por lo que ambas funciones realizan la misma tarea: llamar a las funciones HeapXX.

La equivalencia en Win32 es sencilla de encontrar:

```
GlobalAlloc    ==   LocalAlloc      ==    HeapAlloc
GlobalReAlloc  ==   LocalReAlloc    ==    HeapReAlloc
GlobalFree     ==   LocalFree       ==    HeapFree
GlobalX        ==   LocalX          ==    HeapX
```

El uso de la memoria fija (LMEM_FIXED), movible (LMEM_MOVEABLE), descartable (LMEM_DICARDABLE) o compartida (GMEM_SHARED) sería algo bastante largo de explicar, por lo que no voy a entrar en ese tema, ya que esta serie no trata sobre programación en Windows 3.1.

Nos valdrá con saber que una llamada a LocalX o GlobalX, se transformará en una llamada a HeapX.

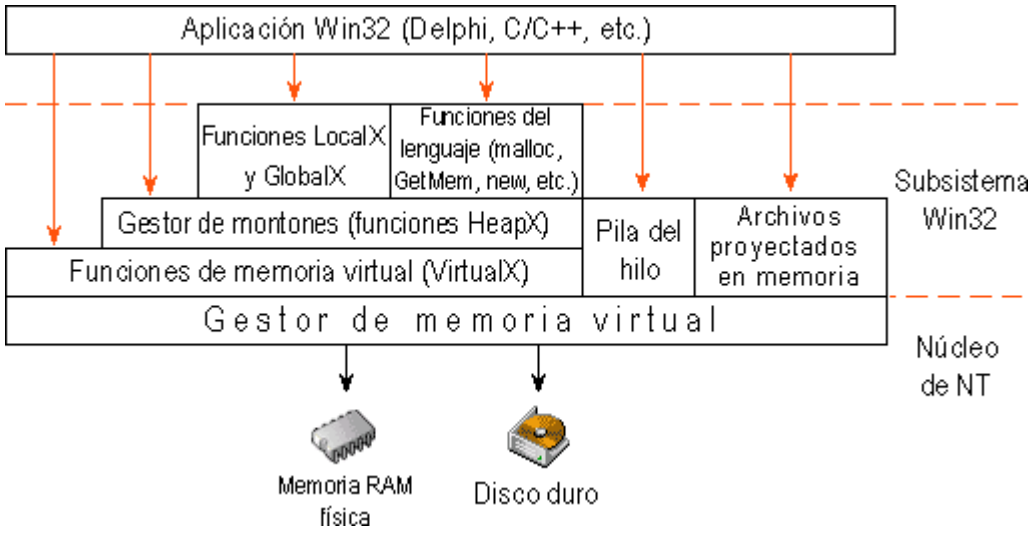
Bien, ya conocemos las funciones para manejar uno o varios montones, pero... ¿para qué queremos varios montones? Antes de dar razones, vamos a profundizar la implementación del montón en Win32, para así comprender mejor el funcionamiento interno.

Implementación de los montones en Win32

Ahora que ya hemos visto cómo se utiliza, vamos a ver que está pasando por dentro cada vez que hacemos una llamada al gestor de montones.

Esquema general

Como nuestro conocimiento del sistema de memoria ha mejorado mucho durante los últimos artículos de *"Los rincones del API Win32"*, vamos a ver una panorámica general, en la siguiente figura, de los niveles que utiliza Win32 para gestionar la memoria.



Analizando este esquema, de abajo a arriba, encontramos:

- **Soporte físico**: la memoria RAM y el disco duro es donde físicamente se soportan los datos en memoria.

- **Gestor de memoria virtual**: este gestor es la base del sistema de memoria en Win32. Toda petición de memoria pasará por este gestor, para así asegurarnos de que siempre se maneja memoria virtual. Del mismo modo, las conversiones entre direcciones virtuales y físicas se realizan en este nivel. Este nivel pertenece al Kernel del sistema, por lo que ciertas instrucciones del procesador podrán realizar llamadas. Por ejemplo, si desde un programa en ensamblador hacemos una reserva de memoria, internamente estamos llamando a funciones de este nivel. Al pertenecer al Kernel, las funciones no son públicas, sino que sólo pueden ser accedidas por otros niveles del Subsistema Win32. En el artículo "La Memoria Virtual" (www.lawebdejm.com/?id=21101), hablamos sobre este nivel.

- **Funciones de memoria virtual**: estas funciones nos permiten trabajar con el Gestor de Memoria Virtual. En el artículo "La Memoria Virtual" (www.lawebdejm.com/?id=21101), hablamos sobre este nivel.

- **La Pila**: utiliza directamente el Gestor de Memoria virtual a través de llamadas a instrucciones del procesador. En el artículo "La pila" (www.lawebdejm.com/?id=21102), hablamos sobre este nivel.

- **Archivos proyectados en memoria**: es un sistema que utiliza Win32 para manejar archivos como si fueran bloques de memoria virtual. En el artículo "Archivos proyectados en memoria" (www.lawebdejm.com/?id=21140), hablamos sobre este nivel.

- **Gestor de montones**: este gestor permite manipular bloques de memoria mucho más pequeños que si utilizáramos directamente el gestor de memoria virtual. Este es el artículo donde estamos profundizando sobre la gestión de montones.

- **Funciones LocalX y GlobalX**: estas funciones se incluyen por compatibilidad con versiones anteriores de Windows. En Win32, lo único que hacen es realizar llamadas a al gestor de Montones. No vamos a profundizar sobre estas funciones porque están obsoletas, aunque durante este artículo hemos explicado un poco su funcionamiento.

- **Funciones y operadores del lenguaje (malloc, calloc, new, etc.)**: realizan llamadas al Gestor de Montones. En esta serie de artículos no vamos a profundizar en ellas, ya que nuestra misión es abarcar la programación con el API Win32, y no con el propio lenguaje de programación, aunque todo lo dicho en este artículo, es aplicable a estar funciones, ya que normalmente hacen una llamada interna a las funciones del montón.

Las flechas rojas del esquema indican los puntos de entrada a la memoria que tendría un programador desde su aplicación:

- Utilizando funciones propias del lenguaje.
- Utilizando las funciones LocalX y GlobalX.
- Utilizando las funciones de manipulación de montones.
- Utilizando las funciones de manipulación de memoria virtual.
- Utilizando la pila de un hilo (variables locales).
- Utilizando archivos proyectados en memoria.

Implementación

Internamente, un montón no es más que una lista doblemente enlazada soportada en un bloque de memoria virtual (www.lawebdejm.com/?id=21101). En principio, este bloque estará reservado (por consiguiente, será un bloque contiguo), aunque parte de él contará además con compromiso físico.

El montón de divide en dos secciones:

- **Cabecera:** contiene toda la información necesaria para gestionar el montón, como la lista de bloques libres, tamaño total, objeto de sincronización, etc.

- **Bloque**: una lista enlazada de todos los bloques que componen el montón. Estos bloque se crean durante cada llamada a HeapAlloc y se componen a su vez de:

    - **Cabecera**: información específica del bloque.

    - El **bloque** de memoria donde se almacena la información.

En la cabecera del montón, uno de los datos que se almacena es un puntero a una lista enlazada que almacena los bloques libres. Esta lista, llamada *free chain* nos proporciona una cadena de

bloques "listos para usar", que serán extraídos del *free chain* e insertados en la lista del "bloque" cada vez se reserven. Bueno, mejor vamos poco a poco y explicaremos qué ocurre cada vez que reservamos un bloque de memoria en el montón:

¿Cómo funciona la reserva de bloques?

Cada vez que llamamos a la función HeapAlloc pidiendo un bloque de tamaño X, el sistema se recorre la lista de libres (*free chain*) en busca de un bloque del tamaño buscado, o mayor. En caso de no encontrarlo, nos retornará el mayor posible (por eso debemos usar HeapSize). En caso de que exista, el sistema divide el bloque encontrado en dos: uno del tamaño que nosotros le hemos pedido, y el otro del tamaño restante. Es decir, si hemos hecho una petición de un bloque de 16 KB, y el sistema ha encontrado que el más cercano es de 20 KB, enconces dividirá este bloque en dos: uno de 16 KB (el que nos retornará) y otro de 4 KB. El bloque sobrante (el de 4 KB) se volverá a insertar en la lista de libres, y el otro (el de 16 KB) se inserta en la otra lista doblemente enlazada, formada por los bloques reservados, y se retornará su dirección de memoria.

Una vez que sabemos esto, es fácil imaginarse que un montón puede fragmentarse rápidamente, ya que los bloques que lo componen se van troceando en bloques más pequeños. Para evitar esto, y como ya hemos comentado, es muy recomendable utilizar la regla de las potencias de 2: *todo bloque que reservemos debe ser un un tamaño que sea potencia de 2*.

Dentro de la cabecera del montón, también se almacena un puntero al siguiente montón creado en el proceso, para poder recorrer los montones de un proceso a través de la función GetProcessHeaps.

Cuando se hace la llamada a HeapCreate, se reserva el espacio total del montón a través de la función VirtualAlloc, por lo que todo el espacio de un montón está direccionado en un bloque contiguo de memoria virtual.

Pero vamos a ver un ejemplo concreto:

```
{
    HANDLE    monton;

    monton = ::HeapCreate(
                    0,           // sin banderas: por defecto es sincronizado
                    15 * 1024,   // compromiso inicial: 15 KB
                    25 * 1024 ); // reserva inicial y tamaño máximo: 25 KB

    if ( !monton )
        ::MessageBox( ::GetActiveWindow(),
                      "No se ha podido crear el montón",
                      "Error", MB_ICONERROR );
}
```

Cuando ejecutamos este código ocurre lo siguiente:

- Se reserva una región de memoria de 25.600 + X bytes a través de la función VirtualAlloc. Los X bytes representan el tamaño que ocupará la cabecera del montón. Como ya explicamos en su momento, esta función redondeará al múltiplo inmediatamente superior del tamaño de página (4 KB en procesadores x86). Teniendo en cuenta esto, la reserva que se hará es:

  ```
  25.600 bytes = 25,00 KB -> Redondeado a 28 KB = 7 páginas de 4 KB cada una.
  ```

- Se compromete el espacio de la cabecera y un bloque inicial de 15.360 bytes a través de la función VirtualAlloc. Del mismo modo, se redondeará del siguiente modo:

  ```
  15.360 bytes = 15,00 KB -> Redondeado a 16 KB = 4 páginas de 4 KB cada una
  ```

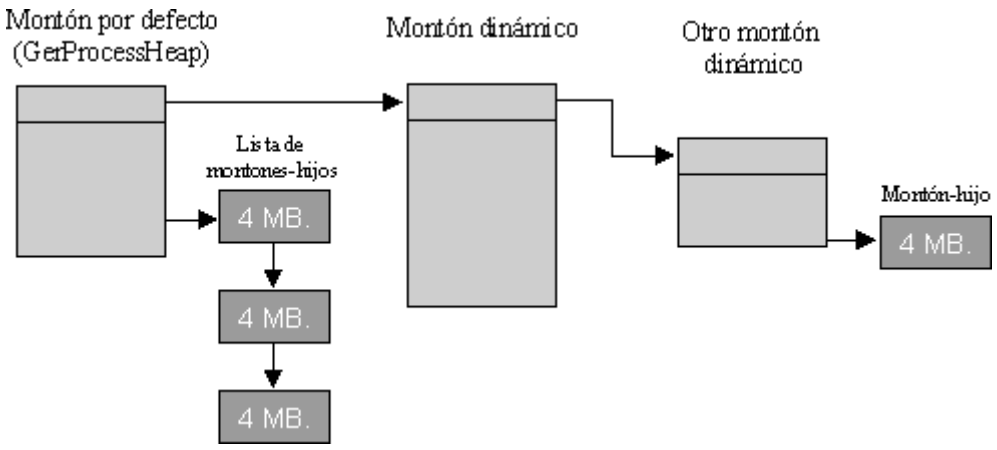
- Se retorna el descriptor (handle) del motón (que no es más que un puntero al inicio de la cabecera).

El aspecto final del montón será el que se muestra en la figura de la izquierda.

Cuando se realiza una llamada a HeapAlloc, se crea un bloque dentro del montón, y se compromete almacenamiento físico para este bloque. Este bloque cuenta con una limitación de 524.280 bytes si el montón no es auto-extensible.

Si el montón es auto-extensible y el tamaño requerido es mayor que el tamaño disponible, Windows crea un sub-montón (o montón-hijo) para acomodar este nuevo bloque de memoria. Es decir, supongamos que tenemos un montón de 1 MB. y que tras varias reservas nos quedan tan sólo 30 KB. libres. Si el montón es autoextensible, se llamará a HeapCreate para crear un montón adicional, normalmente de 4 MB, y se añadirá a una lista de montones hijos. El nuevo bloque se reservará en este nuevo montón, aunque esta operación es transparente al programador. De este modo, si un montón crece constantemente, se tendrón que crear varios sub-montones, lo que es una operación bastante lenta. Microsoft recomienda evitar esta situación, intentando que el espacio inical del montón sea lo suficientemente grande para que no sea necesario crear sub-montones.

En la siguiente figura se puede ver un esquema de un proceso, con el montón por defecto y dos montones adicionales. También puede verse como ciertos montones cuentan con sub-montones asociados.



Cuando se realiza una llamada a HeapFree, el bloque se marca como libre, pero no se anula el compromiso físico, sino que se mantiene para sucesivas asignaciones, aunque Windows se reserva el derecho de anular este compromiso si la memoria es escasa. Una llamada a HeapCompact, nos asegura que se anula el compromiso físico de todos los bloques libres.

Montones de baja fragmentación

Ahora que ya sabemos que los montones son propensos a fragmentarse, vamos a explicar una nueva característica que ha introducido Microsoft en Windows XP y Windows .NET Server: los montones de baja fragmentación (*low fragmentation heaps*: LFH).

Los montones normales están optimizados para recibir muchas peticiones de bloques de tamaño pequeño. Sin embargo, si hacemos peticiones de bloques grandes, el rendimiento no será el mejor posible.

Los LFH son un tipo especial de montones, que se comportan de forma distinta a la hora de reservar memoria de su interior. ¿Recordáis cuando hemos dicho que es recomendable reservar siempre bloques de un tamaño que sea potencia de 2? Pues precisamente esa es la filosofía de estos montones.

Cuando se hace una petición de memoria a un LFH (con HeapAlloc, como ya sabemos), el sistema no hará la búsqueda en el *free chain* como ya hemos explicado, sino que retorna un bloque de un tamaño dentro de una serie de tamaños prefijados.

Microsoft han definido 128 tamaños distintos, y siempre se retornará un bloque de alguno de esos tamaños, independientemente del espacio que pidamos con HeapAlloc o HeapReAlloc. Con esta técnica, no evitamos que el montón esté fragmentado, aunque lo que sí conseguimos es que los trozos sean, como máximo, de 128 tamaños distintos. ¿Y para qué queremos limitar los posibles tamaños de los bloques? pues para que cuando se busquemos un bloque de un tamaño X (y X siempre será un valor dentro de los 128 prefijados), tendremos más posibilidades de encontrar un bloque contiguo de nuestro tamaño sin tener que dividir otro bloque en dos (como hacíamos con los montones normales).

La mala noticia es que, el mayor de estos 128 tamaños prefijados, es de 16.384 bytes, de decir: 16 KB. Así que, con los montones LFH, el bloque más grande que podemos reservar es de 16 KB. Si pedimos un valor mayor, se utilizarás la técnica que ya conocemos: la búsqueda en la cadena de libres y la división del bloque encontrado en dos trozos.

Puede ser interesante el uso de montones LFH, aunque, por ahora, estamos limitados a Windows XP y Windows .NET Server. Para versiones anteriores, podemos simular esta técnica utilizando las dos reglas que ya hemos visto: **utilizar bloques de un tamaño múltiplo de 2** y **reasignar bloques siempre al doble de su tamaño original**.
Por cierto, para el que quiera utilizar los LFH, llamar a la nueva función HeapSetInformation, del siguiente modo:

```
{
    ULONG  lfh = 2;

    // hacer que el montón por defecto sea un LFH
    SetHeapInformation( GetProcessHeap(),              // montón a cambiar
                        HeapCompatibilityInformation, // enumerado en "winnt.h"
                        &lfh,
                        sizeof(lfh) );
}
```

Ventajas del uso de montones

Utilizar los montones frente a la memoria virtual directamente, nos proporciona muchas ventajas:

- Se abstrae al programador de tareas de bajo nivel, como la manipulación de páginas y compromiso físico.
- A la hora de gestionar muchos objetos de distintos tamaños, se aprovecha el espacio mucho más eficientemente con montones que con memoria virtual, ya que ésta última está orientada a grandes bloques de memoria. Recordemos que la cantidad mínima que se podía direccionar con la función VirtualAlloc era de una página, 4 KB. Si no contásemos con la gestión de montones, cada variable (independientemente de su tipo y tamaño) ocuparía una página en memoria.
- Gestionar múltiples bloques de memoria con montones es más rápido que hacerlo directamente con las funciones de memoria virtual, ya que el gestor de montones intenta mantener el mayor número de páginas en RAM, ahorrándose así el tiempo de movimiento de páginas entre la memoria física y el archivo de intercambio en el disco duro.
- Con montones es posible situar ciertas variables dentro de un pequeño rango de direcciones virtuales, aumentando así el rendimiento y disminuyendo la fragmentación de la memoria.

¿Uno o varios montones?

Como ya hemos dicho, un proceso cuenta al menos con un montón: el montón por defecto, aunque es posible crear otros montones para almacenar datos en ellos. Sin embargo, el montón por defecto es especial y tiene algunas peculiaridades. Vamos a ver cuales.

El montón por defecto

Aunque tengamos varios montones adicionales, el más importante es el montón por defecto. Esto es porque, internamente, cualquier reserva de memoria utilizando funciones del lenguaje como malloc, calloc, etc., se hace en el montón por defecto.

Siendo prácticos podemos decir que el siguiente bloque de código:

```
{
    void    *buffer;

    buffer = malloc( 1024 );

    // aquí se hace lo que sea

    free( buffer );
}
```

Es equivalente a este otro código:

```
{
    void    *buffer;

    buffer = ::HeapAlloc( GetProcessHeap(), 0, 1024 );

    // aquí se hace lo que sea

    ::HeapFree( ::GetProcessHeap(), 0, buffer );
}
```

Además, cualquier reserva que haga el sistema para un proceso, también la hace en el montón por defecto. Por ejemplo, cuando hacemos una llamada a la función FindFirstFile, el sistema creará una pequeña zona de memoria para su uso interno, en el montón por defecto del proceso llamante. Esa zona de memoria se liberará con otra función, en nuestro caso con FindClose.

Debido a esto, es muy importante que los accesos al montón por defecto se hagan sincronizados, sin utilizar la bandera HEAP_NO_SERIALIZE, ya que existirán múltiples hilos que accedan a él.

Este montón, tiene un tamaño por defecto de 1 MB, de los cuales tan sólo de comprometen 4 KB.

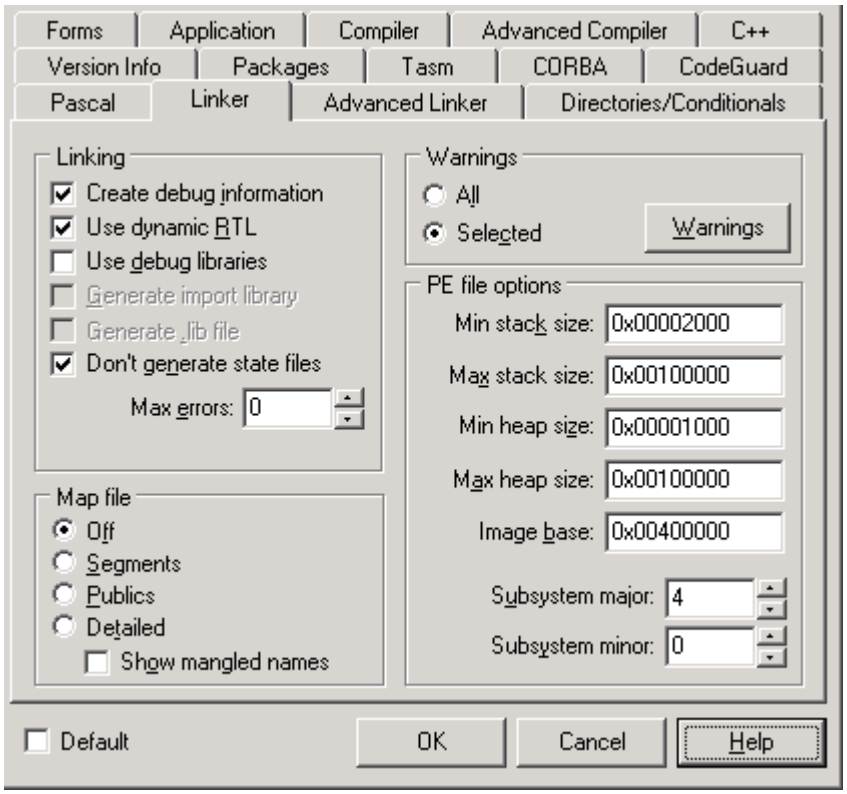
La configuración de este parámetro depende del entorno en el que trabajemos, por poner algún ejemplo:

Visual C++

A través de la opción del enlazador **/HEAP:reservar[,comprometer]**

C++ Builder

Desde la opción de menú *Project - Options - Linker - PE File Options*. En esta sección vemos, tanto la configuración del almacenamiento de la pila, como del montón por defecto:



> Los campos MinHeapSize y MaxHeapSize indican el espacio a reservar y comprometer respectivamente.

Cuando explicamos el parámetro dwTamañoMaximo de la función HeapCreate, dijimos que si contenía un valor mayor que 0, el montón no podría crecer automaticamente. Esto se falso para el montón por defecto, ya que éste sí podrá crecer aunque se haya definido un tamaño máximo. De hecho, el montón por defecto crece en segmentos de 1 MB cada vez que necesita memoria.

La gente de Microsoft recomienda que el tamaño máximo sea lo suficientemente grande como para que no se llegue a una situación en que el montón por defecto deba crecer. Para ello debemos definir un tamaño máximo en el que quepan todos los objetos que vamos a crear dinámicamente durante la ejecución.

El tamaño mínimo nos indica cuánto será comprometido al crear el montón, por lo que si indicamos un tamaño muy grande, la carga de la aplicación se ralentizará (pero durante la ejecución, las reservas serán más rápidas).

Supongamos que nuestra aplicación va a manejar una lista enlazada (o un objeto TList, CList, etc.). Cada elemento de esta lista será un puntero de 32 bits, que apuntará a una estructura creada dinámicamente. Cada una de las estructuras ocupará 234 bytes, que aplicando el alineado de campos pasará a ocupar 256 bytes. Si hacemos una estimación, y decimos que en el caso más desfavorable tendremos en memoria 10.000 estructuras, aplicamos la siguiente ecuación:

```
dwTamañoMaximo = (256 + 32) * 10.000 ) = 2.880.000 bytes = 2812,5 KB
```

En nuestro ejemplo, y para curarnos en salud, yo definiría un tamaño máximo para el montón de 3 MB.

Para calcular el tamaño mínimo, podemos hacer una estimación de la ocupación que se hará del montón durante el arranque (los objetos que van a estar en memoria durante toda la vida de la aplicación).
Si suponemos que de la lista anterior, tan sólo 1000 van a estar en memoria continuamente, entonces aplicamos la siguiente fórmula:

```
dwTamañoMinimo = (256 + 32) * 1.000) = 288.000 = 281,25 KB
```

Yo utilizaría un tamaño mínimo de 300 KB, para asegurarnos que el montón tendrá memoria comprometida inicial para albergar todos los objetos que se cargan en el arranque.

Este (junto con el tamaño de pila), puede ser un buen método para optimizar el tiempo de carga de una aplicación.

¿Cuántos montones debo crear?

Un programador principiante, las únicas variables que utilizará son las almacenadas en la pila (locales) o las variables estáticas (globales). Según su conocimiento va aumentando, aprenderá a utilizar la asignación dinámica de memoria, sin embargo, lo que suele hacer es realizar todas las reservas en el montón por defecto, ya sea utilizando funciones del lenguaje (malloc, calloc, etc.), como funciones para montones locales y globales (LocalAlloc y GlobalAlloc). Un programador experto debe ir más allá, y detectar las situaciones en que es recomendable crear montones adicionales.

Utilizar múltiples montones puede incrementar el rendimiento, sobre todo poniendo énfasis en los siguientes puntos:

- **Un montón por cada hilo**
  En situaciones de acceso masivo al montón, se puede producir un cuello de botella cuando múltiples hilos acceden al montón por defecto repetidas veces (miles o millones). En estas situaciones es recomendable crear un montón por cada hilo, y realizar todas las peticiones de memoria al montón privado de cada hilo. Además, en este caso se puede (y se debe) desactivar el mecanismo de sincronización de hilos (utilizando la bandera HEAP_NO_SERIALIZE), ya que será un solo hilo el que haga accesos a cada montón, y este mecanismo ralentiza la ejecución.

  Suponiendo que estamos programando un programa servidor, podríamos crear un hilo que gestione las peticiones de cada cliente que se conecta a nuestro servidor. Además, si estos hilos hacen un uso intensivo de la memoria dinámica, es muy recomendable crear un montón para cada uno de ellos, utilizando la bandera HEAP_NO_SERIALIZE en la llamada a HeapCreate.

- **Un montón para cada tipo de dato**
  Si utilizamos el montón por defecto para almacenar estructuras de datos, lo más probable, como ya hemos explicado, es que después de las primeras asignaciones/liberaciones, tengamos un montón con bloques de memoria fragmentados. Para aclarar esto, nada mejor que un ejemplo: supongamos que tenemos un montón 140 KB y hacemos las siguientes reservas:

  1. Reservar 20 KB
  2. Reservar 10 KB
  3. Reservar 50 KB
  4. Reservar 40 KB

  Después de estas reservas, el aspecto del montón será el de la siguiente figura:

  Como puede verse, los bloques libres (aunque en realidad la memoria virtual que los soporta está reservada) se representan en blanco y los bloques reservados, en gris. Si después de estas operaciones liberamos el bloque de 10 KB, el aspecto del montón será el como se muestra a continuación:

Este montón está fragmentado, ya que el espacio libre total es de 30 KB, pero si intentamos hacer una reserva con este tamaño, no lo conseguiremos, porque que no hay bloques contiguos suficientemente grandes. Si fuera un montón auto-extensible (como el montón por defecto), y se realizase una reserva de 30 KB, se tendría que crear un montón-hijo para acomodar este espacio, lo cual es una operación muy lenta.

Esta situación, se podría evitar si utilizamos un montón por cada tipo de dato a almacenar.
Supongamos que necesitamos almacenar dos listas enlazadas: la primera de elementos de 5 KB y la segunda de elementos de 7 KB. Si ambas listas se almacenan en el mismo montón, podríamos llegar fácilmente a situaciones como la descrita. Sin embargo, si utilizamos un montón para cada lista, los bloques siempre serán lo suficientemente grandes, porque los "huecos" serán siempre de un tamaño múltiplo del espacio requerido (5 y 7 KB respectivamente) y los nuevos bloques siempre "encajarán" en estos "huecos". En esta figura

se muestra un montón fragmentado, pero que acomodaría perfectamente cualquier petición de 7 KB (representa el montón adicional para la segunda lista enlazada).

- **Situar los bloques de memoria próximos**
  Es conveniente que los bloques de memoria que vayan a ser utilizamos a la vez se reserven dentro de un rango de direcciones virtuales lo más pequeño posible. Esto es debido a que, cuando el sistema necesita más memoria, vuelca cierto rango de páginas al archivo de intercambio para dejar espacio libre en RAM. Si los bloques de memoria que necesitamos no están próximos entre sí, puede darse el caso de que nuestros datos hayan sido volcados al archivo de intercambio, con lo cual sería necesario volverlos a recuperar de disco y proyectarlos en memoria, lo cual es una operación muy lenta. Utilizando un montón para cada estructura de datos, conseguimos que los bloques de memoria que se van a utilizar a la vez se direccionen juntos, con lo que minimizamos el riesgo de que algunos de ellos sean volcados al archivo de intercambio.

  En el ejemplo que pusimos anteriormente, es recomendable que los bloques de ambas listas enlazadas, se sitúen próximos entre sí dentro del sistema de memoria virtual, y esto se consigue utilizando un montón para cada una de ellas

- **Proteger componentes**
  Si en el mismo montón, mezclamos bloques de memoria de distintas estructuras, corremos el riesgo de que una escritura errónea en la manipulación de una de ellas, pueda afectar a los datos de la otra. En nuestro ejemplo, si cometemos un error al manipular la primera lista enlazada, podemos sobrescribir datos de la segunda lista, lo cual sería difícil de depurar, máxime si ambas estructuras se utilizan desde partes muy distintas del programa. Es mucho más conveniente aislar cada una de las estructuras en su propio montón, para evitar así que un error en una parte de un programa, afecte a sus datos, y no a los datos de otros objetos.

  Este método es especialmente recomendable para proteger componentes encapsulados dentro de una DLL, ya que así nos aseguramos que no corromperemos la memoria del programa, sino la de nuestra propia DLL.

Conclusión

En este artículo hemos entrado en profundidad (y mucha) sobre este aspecto, tan importante como desconocido, de la arquitectura de memoria en Win32.

Hemos visto la importancia de los montones para la asignación dinámica de memoria, así como el uso interno que se hace de esta estructura desde cualquier lenguaje de programación.

También, hemos explicado la importancia y el modo de crear montones dinámicos, las funciones para su manipulación y las situaciones en las que es recomendable hacer uso de esta característica.

Los ejemplos

Todo lo que hemos ido explicando, se utiliza de modo práctico en los siguiente ejemplos:

Visual C++ 6
Los archivos Heap.h y Heap.cpp, donde se define las clases CHeap, CHeapBlock, CHeapList y CHeapBlockList, las cuales hacen una implementación orientada a objetos del sistema de montones en Win32.
Además, se incluye el proyecto de Visual C++ 6 (www.lawebdejm.com/?id=21132) para compilar estas clases dentro de Heap.dll y Heap.lib. Para más detalles, se puede consultar la documentación dentro del código fuente.
En breve incluiré un ejemplo de uso desde Visual C++.

C++ Builder 6
Los archivos HeapObject.h y HeapObject.cpp, donde se define la clase base CHeapObject. Los objetos de esta clase se almacenarán en un montón dedicado a tal efecto. Para más detalles, se puede consultar la documentación dentro del código fuente.
Se incluye también un programa de ejemplo (www.lawebdejm.com/?id=21131) escrito en C++ Builder 6, que hace una demostración del uso de estas clases.

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