Sesión 5

Uso de la memoria virtual

Objetivos

Aprender a liberar la memoria reservada o comprometida por un programa.

Aprender a analizar el estado de la memoria virtual del sistema utilizando el administrador de tareas.

Comprender que uno de los objetivos fundamentales de la memoria virtual es proporcionar a las aplicaciones solamente aquella cantidad de memoria que necesiten mientras se encuentran en ejecución. De esta manera se consigue que las aplicaciones no desperdicien recursos de memoria.

Saber manejar la memoria proporcionada por *VirtualAlloc()*, utilizando punteros definidos de la forma apropiada.

1 Conocimientos previos

En la sesión 4 has aprendido a reservar y comprometer memoria. En esta sesión aprenderás otra operación muy importante que se realiza con la memoria virtual: se trata de *liberar* la memoria que previamente se ha reservado o comprometido. Para ello utilizaremos la función *VirtualFree()* de la API Win32. El prototipo de esta función es el siguiente:

```
BOOL VirtualFree(
   LPVOID lpAddress,
   SIZE_T dwSize,
   DWORD dwFreeType)
```

Comenzaremos por comentar el último parámetro de esta función, *dwFreeType*. Este parámetro indica el tipo de operación a realizar por *VirtualFree()*. Existen dos operaciones posibles: 1) **descomprometer** una zona de memoria comprometida y 2) **liberar** completamente una zona de memoria. Cuando *descomprometemos* una región de memoria, ésta pasa del estado COMMITED al estado RESERVERD. Cuando liberamos completamente una zona de memoria, ésta pasa del estado COMMITED o RESERVED (según se encuentre) al estado FREE. Para indicar la operación a realizar se utilizan las constantes MEM_DECOMMIT y MEM_RELEASE, que se encuentran definidas en los ficheros de cabecera. La primera de ellas se utiliza para descomprometer, y la segunda, para liberar completamente la memoria.

El parámetro *dwSize* debe utilizarse de la siguiente forma:

- Si la operación a realizar es descomprometer (MEM_DECOMMIT), *dwSize* indica la cantidad de memoria a descomprometer expresada en bytes. La cantidad de memoria que se descompromete no tiene por qué ser una región completa: puede ser una parte de la misma.
- Si la operación a realizar es liberar (MEM_RELEASE), *dwSize* debe ser 0. En caso contrario, *VirtualFree()* fallaría. La operación liberar provoca la liberación de una región completa. No se puede liberar completamente sólo una parte de una región. Esta es la causa por la que el parámetro *dwSize* se pone a 0.

Finalmente, el parámetro *lpAddress* se utiliza para pasarle a *VirtualFree()* la dirección base de la región a descomprometer o liberar.

En lo que se refiere al **valor retornado**, *VirtualFree()* devuelve CIERTO o FALSO, según que la función tenga éxito o no.

En los ejercicios que se plantean en el resto de la práctica se utilizará de diversas formas la función *VirtualFree()*.

Desarrollo de la práctica

2 Liberación de memoria

En esta parte de la práctica vamos a aprender a utilizar la función *VirtualFree()* para liberar memoria. Para ello vamos a hacer un programa que reserva y libera memoria según se indica en la figura 1. Este programa llevará a cabo las siguientes operaciones:

1. Reservar una región de memoria de 16 MB a partir de la dirección 01000000 (figura 1-A). Para ello se utiliza *VirtualAlloc()* con el parámetro MEM_RESERVE, tal como se indica en la figura 1-A. Tras reservar la región se envía un mensaje mostrando la dirección devuelta por *VirtualAlloc()*. Así se comprueba si la función tiene éxito o no.
2. Ejecutar un bucle en el que se realizan las siguiente operaciones:
   - Se compromete un determinado número de páginas que el usuario introduce por pantalla. Las páginas se comprometen al comienzo de la región previamente reservada (figura 1-B). Para hacer el compromiso se utiliza *VirtualAlloc()* con el parámetro MEM_COMMIT.
   - Se pide al usuario que pulse una tecla para proceder a "descomprometer" la memoria anteriormente comprometida. Cuando se pulsa la tecla, la memoria se descompromete utilizando la función *VirtualFree()* con el parámetro MEM_DECOMMIT (figura 1-C). Se captura el valor retornado por *VirtualFree()* para sacar un mensaje por pantalla indicando si la función tiene éxito o no.
   - Se pregunta al usuario si desea abandonar el bucle o si desea llevar a cabo una nueva iteración. Si el usuario pulsa la tecla 's', el bucle se abandona. En el caso de que pulse cualquier otra tecla, se comienza una nueva iteración. Gracias a este bucle, el usuario puede comprometer y descomprometer memoria tantas veces como desee.

3. Una vez abandonado el bucle se procede a liberar completamente la memoria, utilizando *VirtualFree()* con el parámetro MEM_RELEASE (figura 1-D). Se captura el valor retornado por *VirtualFree()* para sacar un mensaje por pantalla indicando si la función tiene éxito o no.

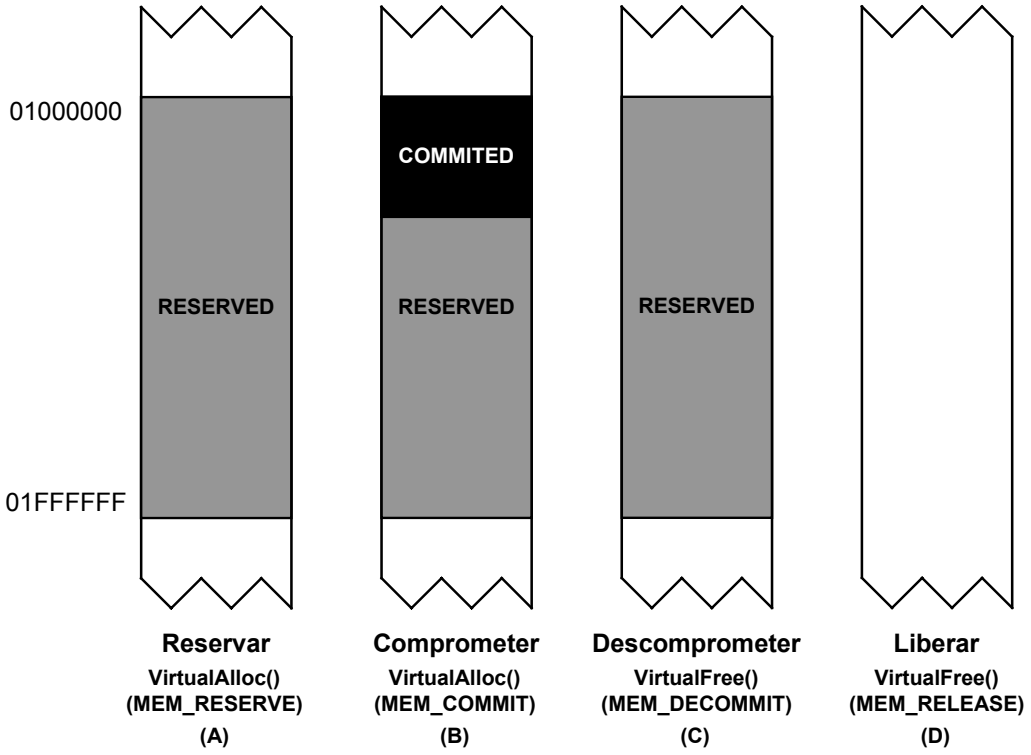

Figura 1: Evolución de la memoria virtual durante la ejecución del programa 2-5prog1

A continuación se muestra el esqueleto del programa perfilado mediante comentarios:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

main()
{
    void *p;       // Para apuntar a la region reservada
    void *q;       // Para apuntar a la region comprometida
    int  num_pag;


    // Reservar region (figura 1-A)
   (...)

    // Sacar por pantalla la direccion retornada por VirtualAlloc()
   (...)

    do
    {
        // Pedir por pantalla un numero de paginas y comprometerlas
        // Usar q para apuntar a la region comprometida
       (...)

        // Esperar hasta que se pulse una tecla y descomprometer
        // la memoria, indicando si se tiene exito o no en la
        // operacion
       (...)
```

```
        // Pedir que se pulse 's' para salir u otra tecla para
        // continuar
       (...)
     }
     while(_getch()!='s');

     // liberar completamente la memoria indicando si se tiene éxito
     // o no en la operacion
    (...)
}
```

Ahora se muestra a modo de ejemplo la salida por pantalla realizada por este programa cuando se comprometen 16 páginas y luego se liberan abandonándose el bucle de procesamiento. Ten en cuenta esta salida por pantalla para determinar las sentencias *printf()* que debes colocar en tu programa.

```
Región reservada a partir de: 01000000

========== Comprometer memoria ==========
Numero de paginas: 16

========= Descomprometer memoria ========
Pulsa una tecla para descomprometer la memoria
Liberacion OK

s = salir; otra tecla = continuar:
liberacion OK
```

H Crea un proyecto llamado `2-5prog1` y agrégale el fichero `2-5prog1.c`. Teniendo en cuanta todas las indicaciones anteriores completa el programa anterior y almacénalo en el fichero `2-5prog1.c`. Obtén el ejecutable de este programa. Ahora prueba tu programa comprobando que, cuando menos, la salida que realiza es igual a la mostrada anteriormente.

Lo que de momento no has podido comprobar es si realmente se está produciendo el compromiso y la liberación de memoria correctamente. Podemos pensar que esto es así a tenor de los mensajes que envía nuestro programa. No obstante, para asegurarnos completamente de que las cosas funcionan bien, vamos a utilizar una herramienta que nos permite analizar en cada momento la memoria virtual utilizada por los programas en ejecución. Dicha herramienta es el *Administrador de tareas*.

H Abre el *Administrador de tareas* del sistema operativo. Elige la ficha *Procesos*. Como has visto en la práctica *Procesos*, esta ficha muestra todos lo procesos que se encuentran en ejecución en el sistema. La información de los procesos se muestra organizada en forma de tabla. En la primera columna de la tabla se indica el nombre de los procesos y en las demás columnas se proporcionan diversos tipos de datos relativos a cada proceso. Ahora vamos a utilizar esta ficha para ver cómo `2-5prog1.exe` utiliza la memoria virtual. Manteniendo abierto el *Administrador de tareas*, ejecuta `2-5prog1.exe`. De momento no hagas ninguna operación con este programa, déjalo esperando en su primera entrada por pantalla. Busca en la ficha *Procesos* la entrada correspondiente a este programa, comprobando así que se encuentra en ejecución.

El *Administrador de tareas* puede mostrar muchos datos diferentes de cada proceso, pero es necesario seleccionar qué es lo que queremos que nos muestre. En nuestro caso nos interesa que nos indique la cantidad de memoria virtual utilizada por cada proceso. Para esto, haz lo siguiente.

H Abre el menú *Ver* y en él elige la opción *Seleccionar columnas*. Entonces se abre una ventana indicando todos los tipos de datos que puede mostrar la ficha *Procesos*. Los campos que ya se encuentren activados, déjalos como están. Activa el campo *Tamaño de memoria virtual*. Pulsa *Aceptar* para volver a la ficha *Procesos*. Busca el proceso `2-5prog1.exe` y pulsa sobre él con el botón izquierdo del ratón para dejarlo remarcado. Ahora busca la columna *Tamaño de memoria virtual*. Escribe a continuación el tamaño de memoria virtual que está siendo utilizada por el proceso[1]:

H Manteniendo bien visibles simultáneamente la consola y la ficha *Procesos* del *Administrador de tareas*, compromete una página y observa cómo el tamaño de la memoria se incrementa en 4 KB. Libera la memoria y observa cómo el uso de la memoria virtual vuelve al valor original. Compromete ahora 25 páginas y observa cómo el uso de la memoria se incrementa en 100 KB. Libera la memoria, observando, de nuevo, el retorno de la memoria virtual utilizada al valor original. Repite este proceso tantas veces como quieras y con el número de páginas que quieras, observando la utilización de la memoria virtual. Por cada página comprometida el uso de memoria deberá incrementarse exactamente en 4 KB.

3 Uso de la memoria virtual

Los programas necesitan memoria para llevar a cabo sus operaciones. La forma habitual utilizada por los programas para conseguir memoria es definir variables o estructuras de datos. Éstas son zonas de memoria en las que el programa puede escribir o leer información. Las variables son la solución idónea para manejar pequeñas cantidades de información. Sin embargo, hay aplicaciones que pueden necesitar manejar cantidades ingentes de información. Un procesador de textos, por ejemplo, puede manejar ficheros de decenas de MB, y además, puede manejar muchos ficheros simultáneamente. ¿Cómo consigue la memoria necesaria una aplicación de este tipo?

El problema de las estructuras de datos estáticas

Una posible solución podría ser el uso de una gran estructura de datos estática, por ejemplo un *array* de bytes definido en el programa, que actúe como *buffer* de almacenamiento. Veamos cómo afecta la utilización de estas estructuras a la cantidad de memoria usada por los programas. Para ello, vas a comenzar escribiendo un programa muy sencillo en el que se defina un *array* de caracteres global. En sucesivas versiones del programa haremos crecer el *array* y veremos cómo afecta esto a la cantidad de memoria utilizada por el programa. El programa que tienes que escribir, perfilado mediante comentarios, se indica a continuación.

```
#include <stdio.h>
#include <conio.h>

// Definir un array de caracteres con 10 elementos
...

main()
{
```

[1] Este valor variará en función de múltiples aspectos de funcionamiento del sistema operativo.

```
    // Inicializar el primer elemento del array con cualquier valor
    ...

    // Imprimir en pantalla el elemento inicializado
    ...

    // Hacer una pausa
    printf("Pulsa una tecla para terminar: ");
    _getch();
}
```

H Crea el proyecto `2-5prog2`. Sin embargo, antes de agregar el programa anterior a este proyecto, vamos a realizar un pequeño cambio en la configuración del mismo.

Por defecto, los proyectos se configuran en el modo *Depuración* (*Debug*). Esto implica que cuando se genera el ejecutable, el sistema de desarrollo introduce en él código y estructuras de datos adicionales con el fin de posibilitar la depuración del programa. Esto hace que sea para nosotros más difícil comprender cómo y cuánta memoria es utilizada por el programa. Para evitar este inconveniente, vamos a configurar el proyecto en modo *Release*, que es la configuración que se utiliza para generar las versiones finales de los programas. Cuando se trabaja con esta configuración, el sistema de desarrollo no introduce ni código ni estructuras de datos adicionales en el ejecutable, y el uso de memoria llevado a cabo por el programa resultará más comprensible.

H Para configura el proyecto en modo *Release*, en el *Explorador de soluciones* pulsa con el botón derecho del ratón sobre el nombre del proyecto y elige *Propiedades*. Se abre entonces la ventana *Página de propiedades* del proyecto. En la parte superior derecha de la ventana, pulsa sobre el botón *Administrador de configuración*. En la ventana que se abre hay una lista desplegable llamada *Configuración de soluciones activas*. Despliega esta lista y selecciona en ella la opción *Release*. Pulsa *Cerrar* para almacenar la nueva configuración.

H Agrega al proyecto el fichero `2-5prog2.c`. Escribe en este fichero el programa anterior debidamente completado. Compila y enlaza este programa. Ahora debes tener en cuenta lo siguiente. Cuando el proyecto está configurado en modo *Debug*, todos los ficheros creados en el proceso de generación del ejecutable, así como el mismo ejecutable se almacenan en una carpeta llamada *Debug*. Pues bien, cuando el proyecto está en modo *Release*, la carpeta en la que se almacena el ejecutable se llama *Release*. Teniendo esto en cuenta, ejecuta `2-5prog2.exe`, comprobando su correcto funcionamiento.

En realidad, este programa no tiene ningún interés en sí mismo. Lo único que queremos saber de él es cuánta memoria utiliza.

H Ejecuta de nuevo `2-5prog2.exe`, pero no lo termines, mantelo en pausa. Abre el *Administrador de tareas*, ficha *Procesos*, y mira en la columna *Tamaño de memoria virtual* la cantidad de memoria utilizada por el proceso. Anótala a continuación[2].

Aunque este programa es muy pequeño, requiere una cantidad nada despreciable de memoria virtual. Esto es debido a que el sistema operativo necesita generar una serie de estructuras de datos en memoria para controlar el proceso.

[2] Este valor variará en función de múltiples aspectos de funcionamiento del sistema operativo.

Por el contrario, las estructuras de datos del propio programa no necesitan casi memoria. Sin embargo, imagina ahora que este programa necesitase un gran *buffer* de bytes para llevar a cabo un determinado procesamiento. Por ejemplo, un *buffer* de 1 MB. Veamos en este nuevo caso cómo se modifican los requisitos de memoria del programa.

H Modifica la definición del *array* de caracteres realizada en el programa `2-5prog2.c` para que contenga 1024*1024 elementos. El tamaño de este *array* será por tanto de 1 MB. Compila el programa y ejecútalo manteniéndolo en pausa. Observa ahora en la ficha Procesos el tamaño de memoria virtual utilizado por el proceso. ¿En cuánto se ha incrementado dicho tamaño respecto a la versión anterior de este programa?

H Repite la prueba anterior, definiendo en el programa un *array* de 2 MB (2*1024*1024), y comprueba que los resultados obtenidos son coherentes.

H Deja la ficha *Procesos* con su configuración original. Para ello abre el menú *Ver*, opción *Seleccionar columnas* y desactiva el campo *Tamaño de memoria virtual*. Pulsa *Aceptar* y cierra el *Administrador de tareas*.

La conclusión de las pruebas anteriores es que cuando se define una estructura de datos estática (global) en un programa, el sistema tiene que proporcionar memoria virtual para la totalidad de dicha estructura. En muchas ocasiones, un programa no conoce a priori la cantidad de memoria que necesita. Así por ejemplo, podría darse el caso de que un programa necesitase un máximo de 64 MB para llevar a cabo un determinado procesamiento. Pero eso sería un máximo, podría haber ejecuciones en las que el programa utilizase todo o una gran parte de ese espacio de almacenamiento, pero también podría haber otras ejecuciones en las que no necesitase nada de esa memoria. En este último caso estaríamos imponiendo al sistema gestionar 64 MB de memoria virtual que no van a ser utilizados, lo cual no tiene ningún sentido. La solución a este problema no está en el uso de estructuras de datos estáticas, sino en las funciones de manejo de memoria virtual proporcionadas por el sistema operativo.

Cuando una aplicación necesita manejar grandes cantidades de memoria y no se conoce a priori cuánta memoria va a necesitar en cada ejecución, la aplicación debe diseñarse para que solicite al sistema operativo la memoria que necesita en cada momento, para lo cual se utilizarán las funciones de manejo de memoria virtual proporcionadas por el sistema operativo.

Ejemplo de programa que usa memoria proporcionada por el sistema operativo

Ahora vas a hacer un ejercicio simple para practicar sobre este concepto. Es decir, vas a trabajar sobre un programa que va a solicitar al sistema operativo la cantidad justa de memoria que requiere para satisfacer sus necesidades. Así cuando este programa necesita memoria se la pide al operativo, y cuando la memoria ya no le es útil, la libera para que pueda ser utilizada por otros programas.

Se trata este ejercicio de hacer un programa que procesa cadenas de caracteres. El programa opera de una forma muy simple: recibe las cadenas de caracteres por pantalla, las almacena en una región de memoria y después las visualiza. Vamos a suponer que las cadenas pueden tener cualquier tamaño entre 1 y 1024 caracteres (incluido el

terminador). El esqueleto de este programa perfilado mediante comentarios se proporciona a continuación:

```
#include <windows.h>
#include <stdio.h>

main()
{
    char (*p)[1024];   // Para apuntar a la region comprometida
                       // y manejar en ella arrays de 1024 caracteres
    int  num_cad;      // Numero de cadenas a procesar
    int  i;            // Contador


    // Pedir por pantalla el numero de cadenas a procesar
   (...)

    // Reservar y comprometer la memoria necesaria para almacenar
    // las cadenas. Hacer que el S.O. elija la region
   (...)

    // Introducir cadenas
   (...)

    // Visualizar cadenas
   (...)

    // liberar completamente la memoria indicando si la operación
    // tiene exito o no
   (...)
}
```

Ahora se muestra, a modo de ejemplo, la salida por consola realizada por este programa, cuando se le pide que procese las cadenas “aaa”, “bbbbbbb” y “ccccc”. Ten en cuenta esta salida por pantalla para determinar las sentencias *printf()* que debes colocar en tu programa.

```
Numero de cadenas a procesar: 3

========== Introducir cadenas ==========

Cadena[0]: aaa
Cadena[1]: bbbbbbb
Cadena[2]: ccccc

========== Visualizar cadenas ==========

Cadena[0]: aaa
Cadena[1]: bbbbbbb
Cadena[2]: ccccc

Liberacion OK
```

A continuación se indican algunas pautas importantes que debes tener en cuenta cuando completes el programa:

- Las cadenas manejadas por el programa son de tamaño variable, pero con un límite máximo de 1024 caracteres. Debido a esto, para cada cadena que haya que procesar

habilitaremos un área de memoria de 1024 caracteres. Así aseguramos que hay sitio suficiente para la cadena, sea cual sea su tamaño.

- Una vez que el programa ha recibido el número de cadenas a procesar, se procede a habilitar una zona de memoria para albergar dichas cadenas. Fíjate que por cada cuatro cadenas será necesario utilizar una página. En este ejercicio se puede reservar y comprometer la memoria simultáneamente, usando los *flags* adecuados unidos mediante el operador '|'. *VirtualAlloc()* no debe recibir directamente la dirección de reserva (como se hizo en todos los ejercicios anteriores), sino que se dejará al sistema operativo que elija la dirección. Esto se consigue pasando en el primer parámetro de *VirtualAlloc()* el valor NULL.
- Otro aspecto, de crucial importancia, que tenemos que tratar en este ejercicio es cómo se maneja la memoria proporcionada por *VirtualAlloc()*. La memoria obtenida mediante *VirtualAlloc()* es una memoria no tipificada, es decir, no está definida para contener ningún tipo de dato determinado[3]. El programador podrá elegir a voluntad qué tipos de datos almacena en esta memoria. Debido a ello, esta memoria siempre se trata mediante punteros que deben definirse para apuntar a los tipos de datos adecuados. En nuestro ejemplo tenemos que manejar *arrays* de 1024 caracteres, por tanto, el puntero que definamos para manejar la memoria proporcionada por *VirtualAlloc()* deberá apuntar a *arrays* de 1024 caracteres. Esta es la razón de que dicho puntero se define de la siguiente forma:

  ```
  char (*p)[1024];
  ```

  Asegúrate de que entiendes perfectamente la declaración anterior. Si tienes dudas pregúntale a tu profesor.
- Para introducir cadenas mediante *scanf_s()* y visualizarlas mediante *printf()*, utilizar el parámetro de sustitución %s. Recordar que cuando se utiliza *scanf_s()* para introducir una cadena de caracteres por consola, hay que indicarle a la función el tamaño del *buffer* en el que se almacenará la cadena de caracteres.

H Crea el proyecto `2-5prog3` y agrégale el fichero `2-5prog3.c`. Escribe en este fichero el programa anterior, debidamente completado, teniendo en cuenta todas las indicaciones planteadas anteriormente. Obtén el ejecutable y pruébalo procesando 3, 7 y 11 cadenas. Para tres cadenas el programa necesitará reservar y comprometer una página, para 7 cadenas, dos páginas, y para 11 cadenas, 3 páginas. Así se demuestra cómo el programa sólo utiliza la cantidad de memoria que necesita. Comprueba que el programa funciona en todos los casos de la forma esperada.

Para terminar esta parte de la práctica, ahora vas a llevar a cabo una mínima modificación del programa anterior. Se trata de hacer que cuando el programa muestre las cadenas almacenadas, también indique la dirección de memoria en la que cada una de estas cadenas se encuentra almacenada. A continuación se indica la salida que debe generar el programa cuando se le pide que procese las cadenas "aaa", "bbbbbbb" y "ccccc". Sólo se muestra la parte del programa que visualiza las cadenas almacenadas.

[3] Esto contrasta totalmente con la memoria asignada a las variables definidas en un programa. Así cuando se define una variable, se le asigna una porción de memoria y se especifica que esa zona de memoria va a contener datos de un determinado tipo.

```
========== Visualizar cadenas ==========

Cadena[0] (Dir: 00360000): aaa
Cadena[1] (Dir: 00360400): bbbbbbb
Cadena[2] (Dir: 00360800): ccccc

Liberacion OK
```

H Crea el proyecto `2-5prog4` y agrégale el fichero `2-5prog4.c`. Copia en este fichero el programa almacenado en `2-5prog3.c`. Haz las modificaciones indicadas en el programa para que se muestren las direcciones. Obtén el ejecutable del programa. Ejecuta el programa procesando 11 cadenas. Esto hará que el programa utilice 3 páginas. Observando la salida llevada a cabo por el programa, indica a continuación las direcciones en las que comienzan las tres páginas utilizadas por el programa:

Dirección primera página:
Dirección segunda página:
Dirección tercera página:

4 Ejercicios adicionales

E Haz un programa que pida por pantalla una dirección de memoria y que conteste con la página en la que se encuentra dicha dirección. La página se indica mediante 5 dígitos hexadecimales. Para ello, puedes utilizar en la cadena de formato de *printf()* el parámetro de sustitución %05x ('*x*' indica hexadecimal; '*5*' indica campo de 5 caracteres; '*0*' indica completar con ceros por la izquierda). Para hacer este programa crea el proyecto `2-5prog5` y almacena el programa en el fichero `2-5prog5.c`.