A **Apellidos** _____ **Nombre** _____ **DNI**_____

Examen de Arquitectura de Computadores. (Telemática.) **Convocatoria de septiembre: 02-09-2004**

A continuación se muestra el listado de un programa ensamblador cuyo objetivo es el procesamiento de un conjunto de cadenas de caracteres que deben estar terminadas con el byte 00h. El procesamiento a realizar es la *inversión* de las cadenas. Para llevar a cabo este procesamiento se han diseñado dos procedimientos:

Procedimiento **`longitud`**: recibe un único parámetro a través de la pila. Este parámetro es la dirección de la cadena a procesar. Como resultado devuelve la longitud de la cadena de caracteres en el registro `edx`. Si, por ejemplo, se definiese en la sección de datos

```
cadena DB “abc”, 0
```

el procedimiento devolvería a través de `edx` el valor `00000003`. (El byte 00 utilizado como terminador de cadena no se cuenta).

Procedimiento **`invierte`**: recibe dos parámetros a través de la pila, que deben ser apilados por el procedimiento llamador en el orden en que son comentados a continuación:

El primer parámetro es la dirección de la cadena de caracteres a invertir.

El segundo parámetro es la longitud de la cadena anterior, especificada como un entero sin signo de 32 bits.

El procedimiento recorre la cadena, tomando inicialmente el primer carácter e intercambiándolo con el último, para después continuar con el segundo e intercambiarlo con el penúltimo, y así consecutivamente hasta llegar a la mitad de la cadena. En ese instante, se detiene el recorrido de la cadena pues ya estarán intercambiados todos los caracteres que la conformaban inicialmente.

Los procedimientos *longitud* e *invierte* se utilizan en el programa principal para procesar un array de cadenas de caracteres definido en la sección de datos del programa. En dicho array se reservan 16 bytes para cada cadena a almacenar. Cuando la cadena no ocupa los 16 bytes (incluido el byte de terminación), las posiciones restantes se rellenan con ceros, hasta que se ocupen los 16 bytes.

El programa principal utiliza un bucle controlado por contador para procesar las cuatro cadenas almacenadas en el array de la sección de datos `array_cadenas`. Sin embargo, solamente realizará la inversión de una cadena si la longitud de ésta es estrictamente mayor que 10.

Como información adicional se conoce que la primera posición de la sección de datos del programa es 00402000, y que justo al comienzo de la ejecución del programa el valor del registro ESP es 0012FFC4. También se conoce la dirección de memoria en la que se ubican ciertas instrucciones del programa. Dichas direcciones se indican en el listado a la derecha de las instrucciones, separadas de éstas mediante puntos.

```
.386
.MODEL FLAT
```

```
EXTERN ExitProcess:PROC

.DATA
array_cadenas DB "123 probando", 0, 0, 0, 0
              DB "Hola mundo", 0, 0, 0, 0, 0, 0
              DB "Examen AC", 0, 0, 0, 0, 0, 0, 0
              DB "2-septiembre-04", 0

.CODE
longitud PROC
  push ebp
  mov ebp, esp
  push edi
  push ebx
  mov edi, [ebp+8]
  xor edx, edx

bucle_1:
  mov bl, [edi]
  cmp bl, 0
  je SHORT fuera
  inc edx
  inc edi
  jmp SHORT bucle_1

fuera:
  pop ebx
  pop edi
  pop ebp
  (--1--)
longitud ENDP

invierte PROC
  push ebp          ..................................... 0040101B
  mov ebp, esp
  push edx
  push ecx
  push edi
  push esi          ..................................... 00401021

  mov  ecx, [ebp+8]
  mov  edi, [ebp+12]

  ; Incializar ESI de la forma apropiada
  (--3--)
```

A

```
  ; Dividir entre dos la longitud de la cadena para saber cuándo
  ; parar de intercambiar caracteres
  shr ecx, 1        ..................................... 0040102D

  ; Bucle principal de intercambio  de caracteres
intercambiar:
  ; Intercambiar los 2 caracteres que toquen
  mov dl, [esi]
  mov dh, [edi]
  mov [esi], dh
  mov [edi], dl

  inc edi
  dec esi

  loop intercambiar       ................................. 00401039

  pop esi
  pop edi
  pop ecx
  pop edx
  pop ebp
  (--2--)

invierte ENDP

; Programa principal

inicio:
  mov  ebx, OFFSET array_cadenas       ..................... 00401043
  mov  ecx, 4

bucle_2:
  push ebx
  call longitud                        ..................... 0040104E
  cmp  edx, 10
  jbe  SHORT sigue
  ; se invierte la cadena
  (--4--)
  call invierte           ................................. 0040105A

sigue:
  add  ebx, 16            ................................. 0040105F
  loop bucle_2            ................................. 00401062

  ; Retorno al sistema operativo
  push 0
  call ExitProcess
END inicio
```

— Indica las instrucciones que faltan en los huecos (—**1**—) y (—**2**—) del listado. **0,5**

```
Hueco (--1--): RET 4

Hueco (--2--): RET 8
```

— ¿Qué instrucciones son necesarias en el hueco (—**3**—) del listado? **0,5**

```
mov esi, edi
add esi, ecx
dec esi
```

— ¿Qué instrucciones son necesarias en el hueco (—**4**—) del listado? **0,5**

```
push ebx
push edx
```

— Determina el valor máximo que alcanza el registro EDI durante la segunda ejecución del procedimiento `invierte`. **0,5**

```
00402037
```

— Indica el dato de 32 bits que se encontrará almacenado en la dirección de memoria 0012FFB8, durante la ejecución del procedimiento `invierte`. Contesta con 8 dígitos hexadecimales. **0,5**

```
0040105F
```

— Codifica la instrucción `mov edi, [ebp+12]`. **0,5**

```
8B 7D 0C
```

— Imagina que estás escribiendo un nuevo programa que tiene una sección de datos idéntica a la del programa anterior. Entonces, utilizando una sola instrucción, escribe sobre la 'u' de la cadena "`Hola mundo`" el carácter 'X'. Indica a continuación la instrucción necesaria. **0,5**

```
mov [array_cadenas + 22], 'X'
```

A **Apellidos** _____ **Nombre** _____ **DNI** _____

Examen de Arquitectura de Computadores. (Telemática.) **Convocatoria de septiembre: 02-09-2004**

A continuación se muestra el listado de un programa cuyo objetivo es probar el funcionamiento de una nueva función, denominada `CopyPages()`. El cometido de dicha función es copiar un conjunto de páginas de memoria de una región a otra región. Esta función recibe como parámetros las direcciones de las regiones origen y destino de copia y el número de páginas a copiar. La función `CopyPages()` puede recibir cualquier dirección origen o destino, pero internamente truncará las direcciones recibidas a direcciones que sean múltiplos de página. Para llevar a cabo la copia de memoria, la función `CopyPages()` utiliza la función `CopyMemory()` de la API Win32. El prototipo de esta función es el siguiente:

```
VOID CopyMemory(
  void *Destination, // puntero a la dirección destino de la copia
  void *Source,      // puntero a la dirección origen de la copia
  int   Length       // tamaño en bytes del bloque a copiar
);
```

En el listado que se proporciona a continuación se muestra el código de la función `CopyPages()`.

Para probar `CopyPages()` se ha preparado una función `main()` en la que se reserva y compromete una región de memoria virtual. Entonces se llama a `CopyPages()` para copiar las dos primeras páginas de la sección de código del programa en la región reservada. Finalmente, para verificar que la copia se ha realizado correctamente, el programa imprime en pantalla los 16 primeros bytes de la sección código del programa y los 16 primeros bytes de la región reservada.

Rellena los huecos existentes en el listado, prestando atención a los comentarios.

NOTA: Recordar que la sección de código de un programa C empieza siempre con la función `main()`.

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

void CopyPages( void * , void * , int );

main()
{
    void *p;           // Para apuntar a la region de memoria a
                       // reservar
    unsigned char *q;  // Para tratar la memoria como posiciones
                       // de un byte
    int i;             // Contador auxiliar

    // Reservar y comprometer una region de 128K de memoria
    // Hacer que el sistema operativo elija la direccion de reserva
    // Salvar la direccion de reserva en p

    p=VirtualAlloc( NULL,                                   0,5
                    32*4096,
                    MEM_RESERVE | MEM_COMMIT,
                    PAGE_READWRITE );

    // Llamar a CopyPages para copiar las dos primeras paginas
    // de la seccion de codigo del programa en la region reservada

    CopyPages( p,  (void *) main  , 2 );                    0,5

    // Mostrar los 16 primeros bytes de la seccion de codigo

    q = ( unsigned char * ) main;                           0,25

    for( i=0; i<16; i++)
       printf("%2x\n", *(q+i));

    // Hacer una pausa
    printf("\n\nPulsa una tecla");
    getch();

    // Mostrar los 16 primeros bytes de la region
    // apuntada por p

    q = ( unsigned char * ) p;

    for( i=0; i<16; i++)
       printf("%2x\n", *(q+i));

    // Liberar completamente (compromiso y reserva) la región de
    // memoria virtual utilizada

    VirtualFree( p, 0, MEM_RELEASE );                       0,25
}

void CopyPages( void *p_destino, void *p_origen, int num_pag )
{

    // Se truncan las direcciones de origen y destino
    // a multiplos de pagina (4K)
```

A

```
    p_destino = (void *)( (int) p_destino & 0xFFFFF000 );
    p_origen = (void *)( (int) p_origen & 0xFFFFF000 );

    // Llamada a CopyMemory() para copiar la region de memoria
    CopyMemory( p_destino, p_origen, num_pag*4096 );
}
```

0,5

— Determina el número máximo de páginas que pueden ser reservadas mediante *VirtualAlloc()* en la región de memoria determinada por el intervalo de direcciones [0052A000 – 00536FFF]. Debe tenerse cuenta que el funcionamiento de *VirtualAlloc()* se encuentra sujeto a una granularidad de reserva de 64K.

7 0,5

~~La jerarquía de memoria de un computador está formada por una memoria principal de 64 KB y una *cache* de 2 KB. Se trata de una *cache* de 4 vías que cuenta con un total de 64 bloques. Teniendo en cuenta esta información, contesta a las preguntas A, B y C.~~

— A) ~~Determina cuántos bloques diferentes de la memoria principal se pueden ubicar en cada conjunto de la *cache*. Contesta en decimal.~~

~~128~~ 0,5

— B) ~~Supón que la jerarquía de memoria recibe una petición de lectura sobre la dirección AB13, y que como resultado de dicha petición, el bloque de memoria en el que se encuentra dicha dirección resulta *cacheado*. ¿Qué combinación de bits se copiaría en el campo etiqueta del bloque de la *cache* escrito en esta operación? Contesta en binario.~~

~~1010 101~~ 0,5

— C) ~~Indica el número de bloque de memoria principal más pequeño que se puede almacenar en el conjunto 9 de la *cache*. Contesta en decimal.~~

~~9~~ 0,5

— Se sabe que las direcciones físicas manejadas por un computador son de 20 bits y que la memoria instalada en dicho computador ocupa el 25% del espacio de direcciones físico. El tamaño de las páginas manejadas por el sistema es de 4K. Por otro lado, se conoce que la tabla de páginas del proceso que se encuentra en ejecución tiene 16 entradas con el bit de presencia en estado 'SI'. Se desea conocer qué porcentaje de la memoria física es ocupado por el proceso.

0,5

25%

— Indica cuál o cuáles de las siguientes afirmaciones son CIERTAS. Contesta ninguna si crees que ninguna lo es.

A) Las posiciones del espacio de direcciones de E/S de la arquitectura IA-32 son de 16 bits.

B) Para enviar un dato a un puerto de E/S se utiliza la instrucción OUT.

C) Las interrupciones NMI son aceptadas por el procesador IA-32 enviando una señal a través de la línea INTA.

D) El número de interrupción asociado a las interrupciones NMI es el 3.

B 0,5

— ~~Explica la estrategia utilizada por la memoria cache de escritura diferida (write back) para mantener la coherencia entre la información almacenada en la cache y en la memoria principal.~~

0,5

~~Se añade a cada bloque de la cache un bit llamado "dirty", que se activa en el momento que el bloque es escrito. Los bloques marcados como "dirty" se envían a memoria principal antes de ser reemplazados~~

— ~~Explica el concepto de tablas de páginas globales frente a locales, indicado para que se usan cada una de ellas.~~

0,5

~~Las tablas globales son compartidas por todos los procesos de un sistema. Las tablas locales son exclusivas de cada proceso.~~

~~Las tablas globales se usan para mapear las páginas ocupadas por el S.O. Las tablas locales se usan para mapear las páginas de código y datos de cada proceso.~~

— Indica y define brevemente los diferentes estados por los que puede pasar un proceso durante su ejecución.

0,5

Nuevo: Estado del proceso durante su creación.

Listo: El proceso está esperando ser asiganado al procesador para su ejecución.

En ejecución: El proceso tiene la CPU y ésta ejecuta sus instrucciones.

En espera: El proceso está esperando a que ocurra algún suceso, como por ejemplo la terminación de una operación de E/S.

Terminado: Estado del proceso durante su eleminación