PRÁCTICA 2

Arranque del Sistema Operativo (Boot loader)

ÍNDICE

1 OBJETIVOS

Los objetivos de este tutorial son:

- Que el alumno conozca el nivel preciso sobre el que se implementa el sistema operativo de un ordenador real (es decir inmediatamente por encima del nivel de lenguaje máquina y de la BIOS).
- Que el alumno sea capaz de implementar el arranque de un sistema operativo desde un disquete estándar utilizando programación a bajo nivel en C.

2 INTRODUCCIÓN

Al encenderse el ordenador la CPU empieza a ejecutar instrucciones en memoria (ROM) a partir de una dirección prefijada. En la ROM toma el control lo que se denomina la BIOS (Sistema de Entrada/Salida Básico) que es una especie de rudimentario sistema operativo proporcionado por el fabricante del ordenador para poder operar con los dispositivos de E/S de una forma estandarizada. Gracias a que cada fabricante incorpora su propio BIOS a sus modelos clónicos del IBM PC, ofreciendo todos ellos la misma interfaz de llamadas (al sistema BIOS), es posible que un mismo programa funcione correctamente en cualquiera de esos ordenadores, a pesar de contar con dispositivos periféricos que pueden ser completamente diferentes, y sin tener que hacer un reconocimiento exhaustivo del hardware del ordenador.

La BIOS, una vez que toma el control, reconoce e inicializa los controladores de los dispositivos más usuales (teclado, tarjeta de vídeo, controladores de disquete y disco duro, etc.) estableciendo los vectores de interrupción (en la RAM) con las direcciones de las rutinas de tratamiento de interrupción correspondientes (residentes en la ROM). A continuación la BIOS intenta cargar el sistema operativo desde alguno de los dispositivos de E/S que ha reconocido (disquetera, cdrom, disco duro, usb o tarjeta de red). En caso de que ninguno de esos dispositivos aporte el sistema operativo, la BIOS simplemente se queda bloqueado.

En el caso de la disquetera, la BIOS espera que tenga introducido un disquete cuyo primer sector lógico (el sector lógico 0 que está en el sector físico 1, cabeza 0 y pista 0) tenga en algunos de sus 512 bytes ciertos valores. Si tras leer el sector lógico 0 la BIOS encuentra esos valores correctamente establecidos, deduce que ese sector contiene además el código del sistema operativo que se ocupa de realizar su carga desde el disquete, por lo que le cede el control, dando por terminado su cometido.

El código contenido en el sector de arranque tiene en cuenta hasta cierto punto la estructura del sistema de ficheros existente en el disquete (sector de arranque, sectores reservados, la(s) FAT(s), el directorio raíz y los sectores de datos) para realizar la carga del sistema operativo en la memoria RAM, ubicándolo en una zona apropiada para su ejecución. Finalmente el código del sector de arranque cede el control al sistema operativo ya cargado en la memoria.

En la programación del sector de arranque se manejan ya a un nivel elemental todos los conceptos básicos de un sistema operativo: la activación de diferentes programas, la gestión de la memoria, la reubicación de código, el manejo de los dispositivos de E/S, las llamadas al sistema y el sistema de ficheros. Por ese motivo este tipo de ejercicios de programación juega un papel muy importante en un primer curso de sistemas operativos.

3 TRABAJO A REALIZAR

La tarea a desarrollar en este tutorial es justamente la programación en C (con recursos de bajo nivel) del sector de arranque de un disquete de 3 ½ pulgadas de doble cara y doble densidad (1,44 MBytes) cumpliendo las siguientes **especificaciones**:

1) Por supuesto, la BIOS debe reconocer el disquete donde se escriba ese sector de arranque como un disquete correcto con un sector de arranque correcto. Aquí juegan un papel fundamental los bytes 510 y 511 del sector de arranque (es decir la signatura del sector de arranque `55 AA`). Además MS-DOS debe reconocer también el disquete como un disquete correcto con un sistema de ficheros FAT12 correcto, de modo que por ejemplo el comando **`dir a:`** se ejecute sin problemas.

2) Una vez que la BIOS ceda el control al código del sector de arranque, ese código deberá realizar la carga de lo que haya en los primeros 320 sectores lógicos consecutivos siguientes al espacio ocupado por el directorio raíz del disquete. Tras la lectura de cada uno de esos sectores debe escribirse en la pantalla un carácter `'.'`, con el fin de poder seguir el avance de la carga de los 320 sectores.

3) La carga de los 320 sectores mencionados debe tener como destino final las 160 K direcciones de memoria consecutivas que están a partir del tercer KByte de la memoria principal (dirección de comienzo 000600H).

4) La carga de los 320 sectores debe respetar las siguientes zonas de memoria: la tabla de vectores de interrupción (de 000000H a 0003FFH), el área de trabajo de la BIOS (de 000400H a 0004FFH) y las direcciones utilizadas por los controladores y la ROM (de 0A0000H a 0FFFFFH). Dicho de otro modo, durante la carga no debe escribirse en ninguna de las zonas de memoria mencionadas.

Es natural que la especificación anterior plantee al alumno muchas dudas sobre multitud de pequeños detalles que significan una laboriosa tarea de documentación. Por ese motivo vamos a hacer a partir de aquí una exposición pormenorizada de las tareas que es necesario que el alumno lleve a cabo. Dichas tareas son las siguientes:

Actividad 1: Entender cómo debe funcionar el disquete de arranque

Actividad 2: Entender la generación de código necesaria

Actividad 3: Esqueleto para el sector de arranque

Actividad 4: Llamadas a la BIOS de manejo de teclado y pantalla

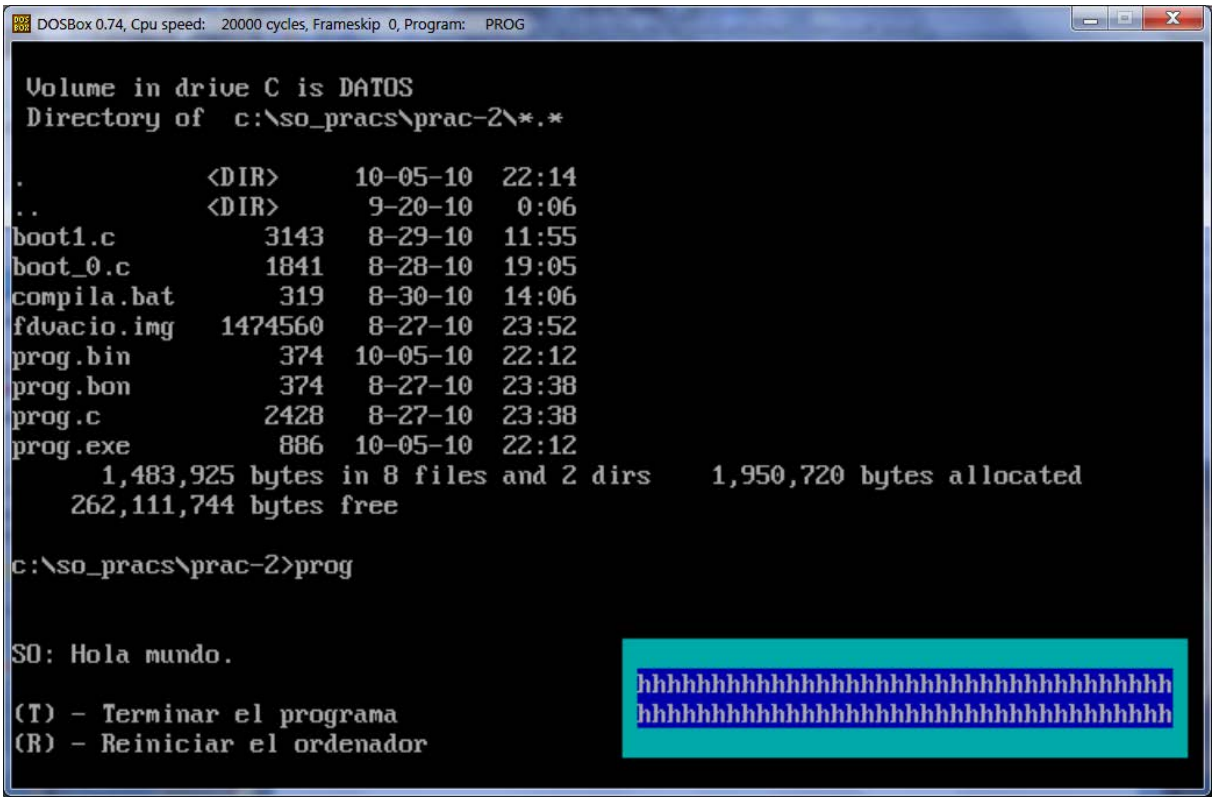
Actividad 5: Llamadas a la BIOS de acceso al disco

Actividad 6: Manejo de direcciones de memoria

Actividad 7: Implementación final del sector de arranque

4 ACTIVIDAD 1: Entender cómo debe funcionar el sector de arranque

Para entender lo que se está pidiendo no hay mejor manera que ver un sector de arranque ya hecho que cumpla las especificaciones del apartado anterior. En el directorio c:\imgs4so se encuentra el fichero “mi_boot.bin” que contiene el código de sector de arranque que cumple las especificaciones del ejercicio. Para comprobar esto, vamos a utilizar un programa que haga las veces de S.O. para ser arrancado desde disquete con dicho cargador (boot loader). En el directorio de la práctica, “c:\practi.cas\prac-2” se encuentra el programa “prog.c”, que una vez compilado (mediante el comando batch: compila progc) y ser ejecutado (programa generado: prog.exe) bajo MS-DOS, se mostraría la siguiente pantalla:



Como puede verse el programa `"prog.exe"` muestra en pantalla un saludo con menú, donde las opciones son: terminar el programa (volviendo a MS-DOS) o reiniciar el ordenador. En la esquina inferior derecha de la pantalla se muestra además un rectángulo de color azul claro cuyo interior se rellena con el último carácter que se haya tecleado.

El programa anterior no es, ni pretende ser, un sistema operativo, pero nos permitirá comprobar el buen funcionamiento del sector de arranque contenido en `"mi_boot.bin"`, a la vez que permitirá que el alumno se dé cuenta de cómo es la programación en el nivel más básico de la implementación de un sistema operativo.

El primer problema que se nos presenta es que el fichero ejecutable `"prog.exe"` no comienza directamente con la primera instrucción a ejecutar, sino que comienza con una cabecera de

512 bytes cuyo formato puede consultarse con el programa "`techelp`" y de la que podemos obtener fácilmente su información si tecleamos:

C:\practi.cas\prac-2\exe2bin /h prog.exe

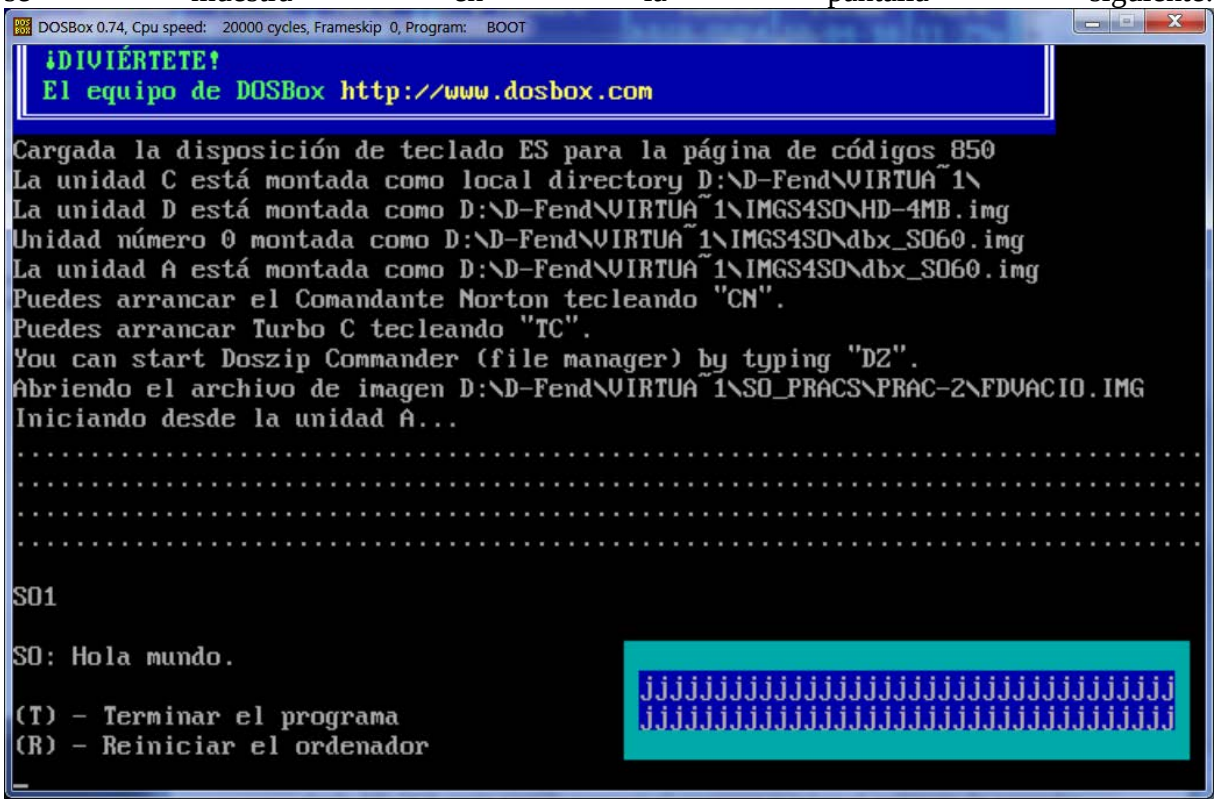
Esta cabecera del fichero es algo necesario para MS-DOS, pero para nuestro sistema, no debe estar en los programas generados, por tanto debe ser suprimida. Existen muchos métodos para hacer esto. En nuestro caso utilizaremos el programa conversor de formato "*exe2bin*", incluido en el directorio "c:\utils". La supresión de la cabecera se realiza de un modo transparente, ya que en el comando "*compila.bat*" que se encuentra en el directorio de la práctica, se encuentra ya incluida la línea que invoca a dicho programa para llevar a cabo tal propósito. Si dentro del directorio "*c:\practi.cas\prac-2"*, ejecutamos el comando: *compila prog*, se generarán los ficheros ***prog.exe*** y *prog.bin*. Este último es la versión compilada de *prog.c*, sin la cabecera. Se verá que el tamaño es justo 512 bytes menos que la versión "exe".

Una vez obtenido el programa `"prog.bin"` por el método que sea, tendríamos ya todos los elementos necesarios para construir nuestro disquete de arranque. Lo primero es dar formato a una imagen de disquete (***Se recuerda que la imagen debe estar montada como unidad física [0 ó 1] ya que DosBox usa también unidades lógicas que no admiten el comando tradicional del DOS "format". Para dicho propósito puede montarse por ejemplo la imagen de disquete del direcotorio de la práctica "fdvacio.img"***). Por rapidez se recomienda utilizar el comando "formatea.bat" que se encuentra en el directorio C:\practi.cas". Este comando es muy simple, únicamente escribe el fichero "formato.bin" en los primeros sectores de un disquete, utilizando el programa "escr_s" ya conocido de la práctica anterior. El fichero escrito, formato.bin, es una imagen binaria de un sistema de ficheros Fat12 de un disquete vacío. Con esto conseguimos tener un disquete libre de información, aunque sobre el resto de sectores no se escribe nada, y no valdría para borrar toda la información de un disquete, pero eso es algo que no nos preocupa.

El siguiente paso sería copiar el fichero `"mi_boot.bin"` (que se encuentra en el directorio c:\imgs4so) en el sector de arranque del disquete (sector 0), utilizando el programa "escr_s" ya visto. Por último, hay que copiar el fichero "prog.bin" en el disquete. *Es crucial que este fichero sea el primer fichero que se copia en el disquete*, ya que así se garantiza que ocupe los primeros sectores del disco (a partir del 33, que es el primer sector de datos tras el directorio raíz) y el disquete pueda arrancar.

Sólo nos falta comprobar que el disquete que hemos preparado arranca (hace *boot*) el programa "`prog.bin`" escrito en él. Con este fin, seleccionamos con "*D-fend*" alguno de los perfiles ya existentes (también se podría crea uno nuevo para tal efecto), y vamos haciendo "click" sobre el botón "editar", opción "Al arrancar" (abajo izquierda), "arrancar imagen de disquete" y seleccionamos nuestra imagen: por ejemplo "fdvacia.img". El resultado es el que

se muestra en la pantalla siguiente:



El alumno debe comprobar que el programa lee y visualiza correctamente los caracteres que se introducen por el teclado, y que pulsando la tecla `'R'` (mayúscula o minúscula) se reinicia el ordenador. Igualmente debe comprobar que pulsando la tecla `'T'` el sistema no funciona. La razón es porque la operación de terminar el programa no tiene ningún sentido en este momento al ser `"prog.bin"` el único programa existente en el sistema. Cuando ejecutábamos `"prog.exe"` desde MS-DOS tenía sentido terminar el programa y volver al intérprete de comandos (haciendo una llamada al sistema operativo MS-DOS), pero ahora el único "sistema operativo" es el propio `"prog.bin"` quien tiene que resolver todos los problemas por sí mismo.

Esa situación de desamparo se da exactamente igual en el código de arranque contenido en `"mi_boot.bin"`. Ese código tiene que leer sectores del disquete que lo contiene, tiene que cargar en memoria dichos sectores, tiene que escribir en la pantalla y tiene que ceder el control al programa una vez cargado. Para hacer todo eso sólo dispone de su propio código máquina y de la BIOS presente en la ROM.

5 ACTIVIDAD 2: Entender la generación de código necesaria

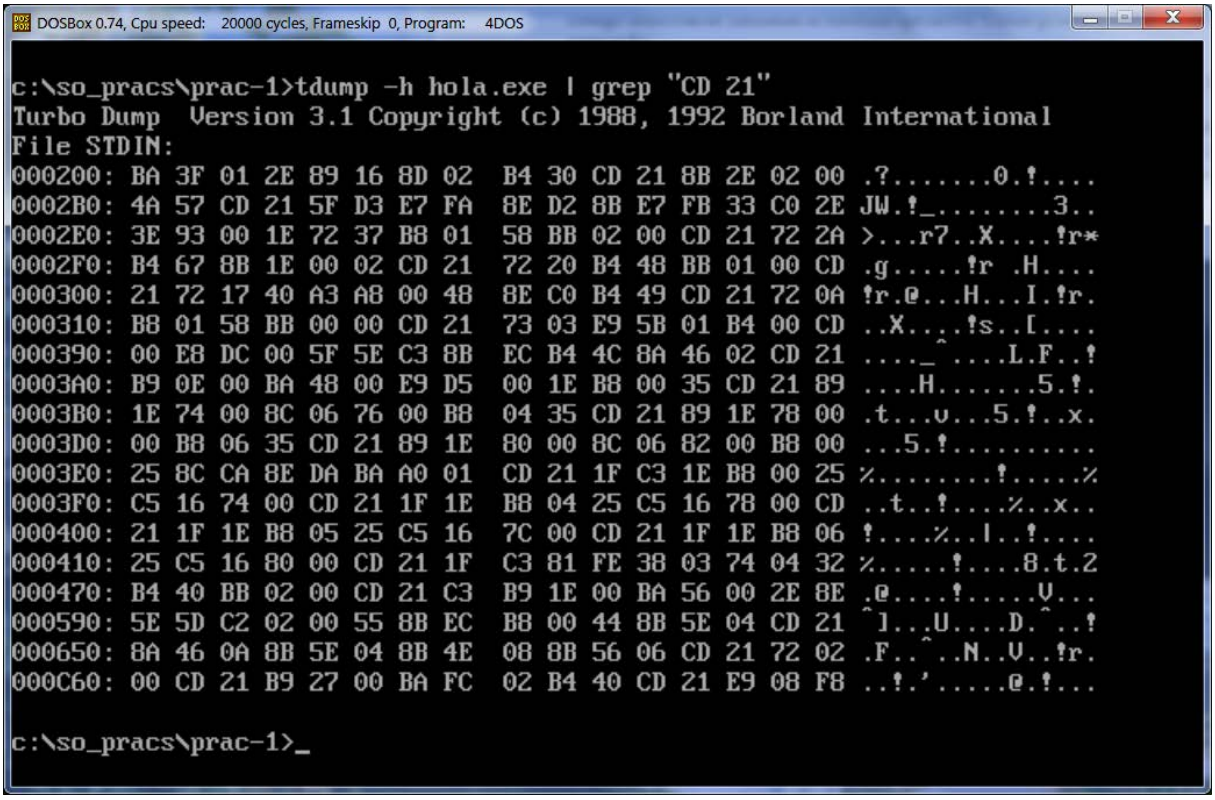
Es muy importante que el alumno se dé cuenta de que tanto el código del sector de arranque `"mi_boot.bin"` como el fichero `"prog.exe"` no son programas de usuario normales escritos en C. En los programas normales el compilador genera código que se apoya sobre el sistema operativo en aspectos como los siguientes:

- la asignación de memoria estática y dinámica al programa
- la entrada/salida y el manejo del sistema de ficheros
- la terminación del proceso asociado al programa en ejecución

Programas aparentemente tan sencillos como el típico `"hola.c"` ya visto incluyen en su código multitud de llamadas al sistema operativo. Como prueba podemos ejecutar el siguiente comando:

```
C:\PRACTI.CAS\PRAC-1>tdump –h hola.exe | grep "CD 21"
```

que muestra las líneas del volcado hexadecimal de `"hola.exe"` que contienen los bytes `"CD 21"`, los cuales corresponden al código máquina de la instrucción `INT 21` (interrupción software o trap 21H) utilizada para invocar las llamadas al sistema operativo MS-DOS:



Como vemos `"so1_hola.exe"` contiene 20 instrucciones `INT 21`, y por tanto ¡invoca 20 llamadas al sistema operativo! De ahí se deduce que `"hola.exe"` nunca podría funcionar tras ser cargado por el sector de arranque del disquete. Por el contrario `"mi_boot.bin"` no contiene ninguna de esas llamadas al sistema, y lo mismo sucede con `"prog.bin"` (exceptuando la llamada al sistema para terminar el programa que se ha dejado a propósito).

Si queremos que el compilador tcc de Turbo C no genere por defecto llamadas al sistema tendremos que establecer una serie de opciones de compilación especiales, como las que utilizaba el comando ***`compila`*** del que debemos echar mano para compilar `"prog.c"`:

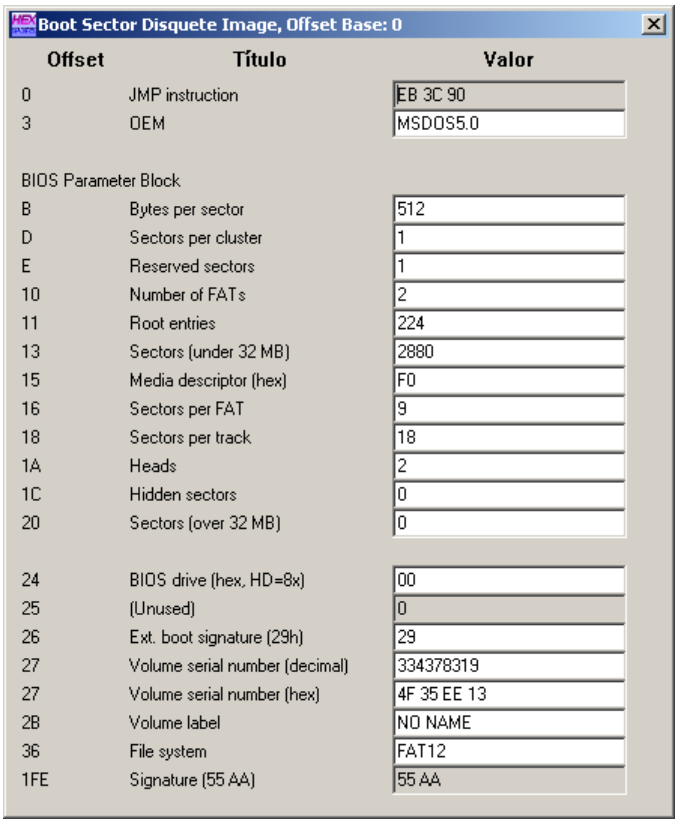
```
C:\PRACTI.CAS\PRAC-2>compila prog
```

Utilizando el comando **`compila`** conseguimos eliminar las llamadas al sistema operativo en el código generado, pero eso tiene como consecuencia que vamos a perder todas las comodidades de que disfrutábamos. Por lo pronto, ahora el punto de entrada del flujo de control no va a ser la función *`main`* correspondiente al programa principal, sino que será la primera instrucción que genere el compilador, es decir la correspondiente a la primera función que se declare en el fichero fuente. Eso no es muy grave ya que desde esa primera función siempre podemos hacer una llamada a la función *`main`*.

Otra incomodidad es que en algunos casos tendremos que ajustar nosotros mismos los segmentos de código, de datos y de pila del programa, cosa que normalmente realiza de una forma automática el compilador haciendo uso del sistema operativo.

6 ACTIVIDAD 3: Esqueleto para el sector de arranque

La estructura del sector de arranque, que se resume en la siguiente plantilla tomada del programa WinHex :



En el sector de arranque coexisten juntos código y datos entremezclados. Concretamente los primeros bytes corresponden a una instrucción de salto `JMP` que evita que se ejecuten los campos de datos del `Bloque de Parámetros de la BIOS`. El salto se dirige al byte situado en el desplazamiento `3E` que es justamente el byte siguiente a la cadena de 8 caracteres que describe el tipo de sistema de ficheros `"FAT12 "`. Por tanto, a partir de ese byte y antes del campo de signatura (`55 AA`) debe haber una serie de instrucciones que realicen la carga del sistema operativo y le cedan el control. En un programa normal el compilador separaría el código de los datos, por lo que vamos a explicar a continuación una manera sencilla de evitar ese problema insertando instrucciones de ensamblador dentro del código C.

```
#define nSectsSO 320          /* Numero de sectores a cargar (S.O.) */
void main () ;                /* declaracion forward de la funcion main */
void start () {
  asm jmp short inicio        /* instruccion JMP */
  asm nop                     /* instruccion NOP (90H) de relleno */
  asm db 'SO v2.00'           /* OEM, 8 caracteres */
  asm dw 512                  /* Bytes por sector */
  asm db 1                    /* Sectores por cluster */
  asm dw 1                    /* Sectores reservados */
  asm db 2                    /* numero de FATs */
  asm dw 224                  /* Entradas del directorio raiz */
  asm dw 2880                 /* numero de sectores en total (16 bits) */
  asm db 0xF0                 /* descriptor de medio */
  asm dw 9                    /* SectPorFAT */
  asm dw 18                   /* Sectores por pista */
  asm dw 2                    /* Cabezas */
  asm dd 0                    /* Sectores ocultos */
  asm dd 0                    /* numero de sectores en total 32 (bits) */
  asm db 0x00                 /* indica la unidad A: */
  asm db 0                    /* Byte que no se usa */
  asm db 0x29                 /* Extension de la signatura */
  asm dw nSectsSO             /* Numero de sectores a cargar */
  asm dw 0x0000               /* Disponible para otros usos */
  asm db 'ETIQUETA   '        /* Etiqueta de volumen, 11 caracteres */
  asm db 'FAT12   '           /* Tipo de sistema de ficheros */
inicio:
  main() ;
}

/* Declaracion de funciones auxiliares */

void main ( ) {
  /* codigo que realiza la carga y cede el control al s.o. */
}
/* Sino se incluyen las directivas _TEXT la directiva org actua sobre
   el segmento de datos y no saldria bien */
asm _TEXT segment byte public 'CODE'
asm     org 01FEh
asm     db 55h, 0AAh
asm _TEXT ends
```

El esqueleto anterior está disponible en `C:\PRACTI.CAS\PRAC-2` como `"boot_0.c"`. La primera línea del programa contiene la directiva `#define` que permite que nos refiramos mediante nombres simbólicos a valores o expresiones (análogamente a como se hace en Pascal definiendo constantes con la cláusula `CONST`). La constante que se define es el número de sectores que ocupa el sistema operativo en el disquete, y por tanto el número de sectores que habrá que cargar. Se estableció en el apartado 3 que serían 320 sectores (120 KBytes).

Tras el `#define` viene una declaración de la función `main` en la que simplemente se muestra su encabezamiento. Esa declaración avisa al compilador de que existe la función `main` por lo que podemos hacer llamadas a ella a pesar de que en ese punto del programa no se haya indicado todavía cuál es su cuerpo.

Luego viene la declaración de una función que hemos llamado a nuestro antojo `start`, y que contiene la instrucción de salto del sector de arranque y el `Bloque de Parámetros de la BIOS`. Es crucial que esa primera función no contenga variables locales, ya que en ese caso el compilador generaría código para asignarles espacio en la pila, y ese código desplazaría la

instrucción de salto inicial a una dirección que ya no sería la cero (dentro del segmento de código).

La palabra reservada `asm` de Turbo C indica que va a insertarse en ese punto del programa el código correspondiente a una instrucción en ensamblador o los datos correspondientes a una directiva del ensamblador. Vemos que primero se inserta la instrucción de salto que requiere el sector de arranque y luego, mediante directivas, se insertan uno a uno todos los campos del `Bloque de Parámetros de la BIOS`. La directiva `db` (define byte) permite introducir una secuencia de caracteres (y cadenas de caracteres) separados por comas. La directiva `dw` (define word) permite introducir una secuencia de palabras (16 bits) separadas por comas. Finalmente la directiva `dd` hace lo mismo con dobles palabras (32 bits).

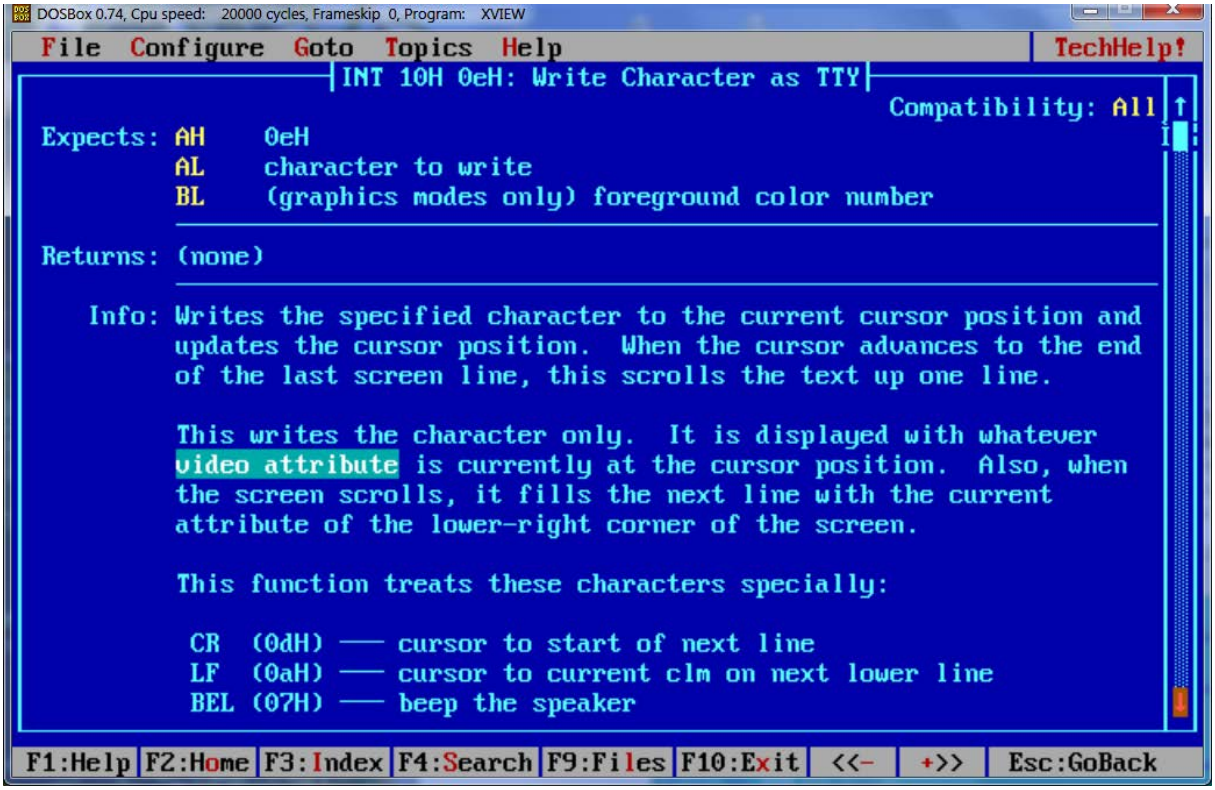
A continuación aparece la etiqueta `inicio:` que es a donde se dirige el salto del principio del programa, llamándose inmediatamente a la función `main` donde en principio debería encontrarse el código que se ocupa de la carga del sistema operativo y de cederle el control. Con ese fin la función `main` puede hacer uso de funciones auxiliares que se declararían antes de ella y después de la función `start`. El formato del sector de arranque impone como requisito que termine con la signatura del sector de arranque (bytes `55 AA`) que debe ocupar exactamente los dos últimos bytes del sector. En otro caso la BIOS rechazaría ese sector de arranque. Por ese motivo hemos introducido las directivas `_TEXT` y `org 01FEH` antes de introducir los dos bytes de la signatura. Habrá que tener mucho cuidado de que el código del programa principal no sea tan grande que esa dirección esté ya ocupada por las instrucciones anteriores del programa principal.

7 ACTIVIDAD 4: Llamadas a la BIOS de manejo de teclado y pantalla

Nuestro sector de arranque va a necesitar escribir en la pantalla algunos caracteres, por lo que vamos a tratar aquí el modo de hacerlo sin utilizar llamadas al sistema operativo MS-DOS y de manera que invirtamos pocos de los preciosos 512 bytes del sector de arranque en el código correspondiente. Una solución es recurrir a la BIOS. El programa `techelp` documenta los servicios que nos ofrece. La BIOS ofrece sus servicios mediante interrupciones software

(*traps*). En el caso de los servicios de manejo de la pantalla el número del vector de interrupción que se utiliza es el 10H (16 en decimal).

Para cada tipo de servicio hemos de indicar como parámetro en el registro AH (byte de mayor peso del registro acumulador AX del microprocesador 8086) un número que identifica la operación concreta que queremos llevar a cabo. En nuestro caso deseamos ser capaces de escribir un carácter por la pantalla como si se tratara de un terminal, por lo que hemos de indicar en AH el código de operación 0E hex. Veamos los detalles sobre esa operación:



Los detalles son mínimos: para escribir un carácter en la pantalla utilizando el servicio 0eH de la interrupción 10H de la BIOS hemos de poner el carácter en el registro AL (byte de menor peso de AX) y el byte de atributo de pantalla en BL (byte de menor peso del registro BX). El atributo determina el color del carácter y el color de fondo del espacio sobre el que se escribe. El atributo normal (carácter blanco sobre fondo negro) corresponde al byte 07H, que es el que vamos a utilizar.

Con lo anterior podemos implementar la siguiente función auxiliar que nos permite visualizar un carácter por pantalla:

```
void printCar ( char car ) {
  asm mov al,car       /* car -> caracter a escribir */
  asm mov bl,07H       /* 07H -> atributo normal */
  asm mov ah,0eH       /* 0eH -> escribir caracter */
  asm int 10H          /* 10H -> servicios BIOS de manejo de la pantalla */
}
```

En cuanto a la lectura de caracteres desde el teclado podemos hacer exactamente lo mismo pero ahora utilizando la interrupción 16H (22 decimal) de la BIOS. En la siguiente pantalla del `techelp` se muestran los servicios ofrecidos a través de esa interrupción. Así,

haciendo uso de la BIOS podemos programar del siguiente modo una función auxiliar que espera a que se pulse una tecla y retorna el carácter correspondiente:

```
char leerTecla ( ) {
  char car ;
  asm mov ah,00H       /* 00H -> leer siguiente tecla pulsada */
  asm int 16H          /* 16H -> servicios BIOS de teclado */
  asm mov car,al       /* El caracter ascii se nos devuelve en AL */
  return(car) ;
}
```

Vamos a ver otra posible función auxiliar que nos permita reiniciar el ordenador desde el programa. En un PC, cuando se enciende el ordenador o se pulsa el botón de reset se cede automáticamente el control a la instrucción que se encuentra en la dirección: 0FFFF0H (1 Mega menos 16 bytes), que expresada en forma de par segmento:desplazamiento del 8086 corresponde (por ejemplo) a la dirección FFFF:0000. Esa dirección es el punto de entrada de la BIOS que está en la ROM, el cual inicia el ordenador. Por tanto en cualquier momento podemos reiniciar el ordenador cediendo el control a esa dirección. Podemos programar del siguiente modo la función de reinicio poniendo la dirección a la que queremos ir en la pila y ejecutando una instrucción de retorno:

```
void reboot ( ) {
  asm push 0FFFFH       /* apilamos el numero de segmento */
  asm push 0000H        /* apilamos el numero de desplazamiento */
  asm retf              /* hacemos un retorno lejano (far) */
}
```

En este momento estamos ya en condiciones de escribir un sector de arranque que se limite a hacer eco en pantalla de los caracteres que se introduzcan a través del teclado. En el directorio `C:\PRACTI.CAS\PRAC-2` se encuentra el programa `"boot_1.c"` que es una primera aproximación al objetivo de este ejercicio y que nos permitirá probar las funciones auxiliares anteriores. Se puede ha añadido la función auxiliar:

```
void finProgDOS () {
  asm mov ah,4ch         /* Llamada al sistema MS-DOS: Terminar Programa */
  asm mov al,00h         /* Codigo de retorno 00h (como con exit(0))     */
  asm int 21h
}
```

con el fin de poder ejecutar primeramente el programa desde MS-DOS terminando el programa normalmente, a la vez que ilustramos cómo se hace una llamada al sistema operativo MS-DOS concreta (véase su documentación en el `techelp`). Esta llamada al sistema deberá suprimirse del código del sector de arranque definitivo, dejando en su lugar la función `reboot`. A continuación mostramos el programa `"boot_1.c"` completo:

```
#define nSectsSO 320          /* Numero de sectores a cargar (S.O.) */
#define CR        13          /* Retorno de carro */
#define LF        10          /* Avance de linea */
#define ESC       27          /* Tecla ESC */

void main () ;                /* declaracion forward de la funcion main */

void start () {
  asm jmp short inicio        /* instruccion JMP */
```

```
  asm nop                     /* instruccion NOP (90H) de relleno */
  asm db 'SO v2.00'           /* OEM, 8 caracteres */
  asm dw 512                  /* Bytes por sector */
  asm db 1                    /* Sectores por cluster */
  asm dw 1                    /* Sectores reservados */
  asm db 2                    /* numero de FATs */
  asm dw 224                  /* Entradas del directorio raiz */
  asm dw 2880                 /* numero de sectores en total (16 bits) */
  asm db 0xF0                 /* descriptor de medio */
  asm dw 9                    /* SectPorFAT */
  asm dw 18                   /* Sectores por pista */
  asm dw 2                    /* Cabezas */
  asm dd 0                    /* Sectores ocultos */
  asm dd 0                    /* numero de sectores en total 32 (bits) */
  asm db 0x00                 /* indica la unidad A: */
  asm db 0                    /* Byte que no se usa */
  asm db 0x29                 /* Extension de la signatura */
  asm dw nSectsSO             /* Numero de sectores a cargar */
  asm dw 0x0000               /* Disponible para otros usos */
  asm db 'ETIQUETA   '        /* Etiqueta de volumen, 11 caracteres */
  asm db 'FAT12   '           /* Tipo de sistema de ficheros */

inicio:
  main() ;
}

void printCar ( char car ) {
  asm mov al,car       /* car -> caracter a escribir */
  asm mov bl,07H       /* 07H -> atributo normal */
  asm mov ah,0eH       /* 0eH -> escribir caracter */
  asm int 10H          /* 10H -> servicios BIOS de manejo de la pantalla */
}

char leerTecla ( ) {
  char car ;
  asm mov ah,00H       /* 00H -> leer siguiente tecla pulsada */
  asm int 16H          /* 16H -> servicios BIOS de teclado */
  asm mov car,al       /* El caracter ascii se nos devuelve en AL */
  return(car) ;
}

void prompt ( ) {
  printCar('S') ;
  printCar('O') ;
  printCar('1') ;
  printCar('>') ;
  printCar(' ') ;
}
void reboot ( ) {
  asm push 0FFFFH       /* apilamos el numero de segmento */
  asm push 0000H        /* apilamos el numero de desplazamiento */
  asm retf              /* hacemos un retorno lejano (far) */
}

void finProgDOS () {
  asm mov ah,4ch         /* Llamada al sistema MS-DOS: Terminar Programa */
  asm mov al,00h         /* Codigo de retorno 00h (como con exit(0))     */
  asm int 21h
}

void main ( ) {
```

```
  char car ;
  prompt() ;
  while ((car = leerTecla()) != ESC) {
    printCar(car) ;
    if (car == CR) {
      printCar(LF) ;
      prompt() ;
    }
  }
  finProgDOS();
  reboot() ;
}
/* Si no se incluyen las directivas _TEXT la directiva org actua sobre
   el segmento de datos y no saldria bien */
asm _TEXT segment byte public 'CODE'
asm     org 01FEh
asm     db 55h, 0AAh
asm _TEXT ends
```

A continuación compilamos el programa del siguiente modo:

```
C:\PRACTI.CAS\PRAC-2>compila boot_1
```

Este comando genera los ficheros “boot_1.exe y boot_1.bin”. Probamos que boot_1.exe se ejecute correctamente y tras comprobar que todo va bien quitamos del fuente la función “finProgDOS” y su llamada y recompilamos, ya que ahora vamos a utilizar el fichero generado “boot_1.c”, que no tiene la cabecera de 512 bytes para probarlo como sector de arranque. El tamaño del fichero debe ser exactamente 512 bytes y si editamos o vemos s interior con cualquier utilidad (cn, tdump..) veremos también que los dos últimos bytes son la signatura 55AA. Por ejemplo:

```
C:\PRACTI.CAS\PRAC-2> tdump –h boot_1.bin | more
```

Tras la comprobación sólo nos queda escribir el sector de arranque en un disquete previamente formateado y reiniciar la máquina virtual DosBOX como se ha hecho en otras ocasiones.

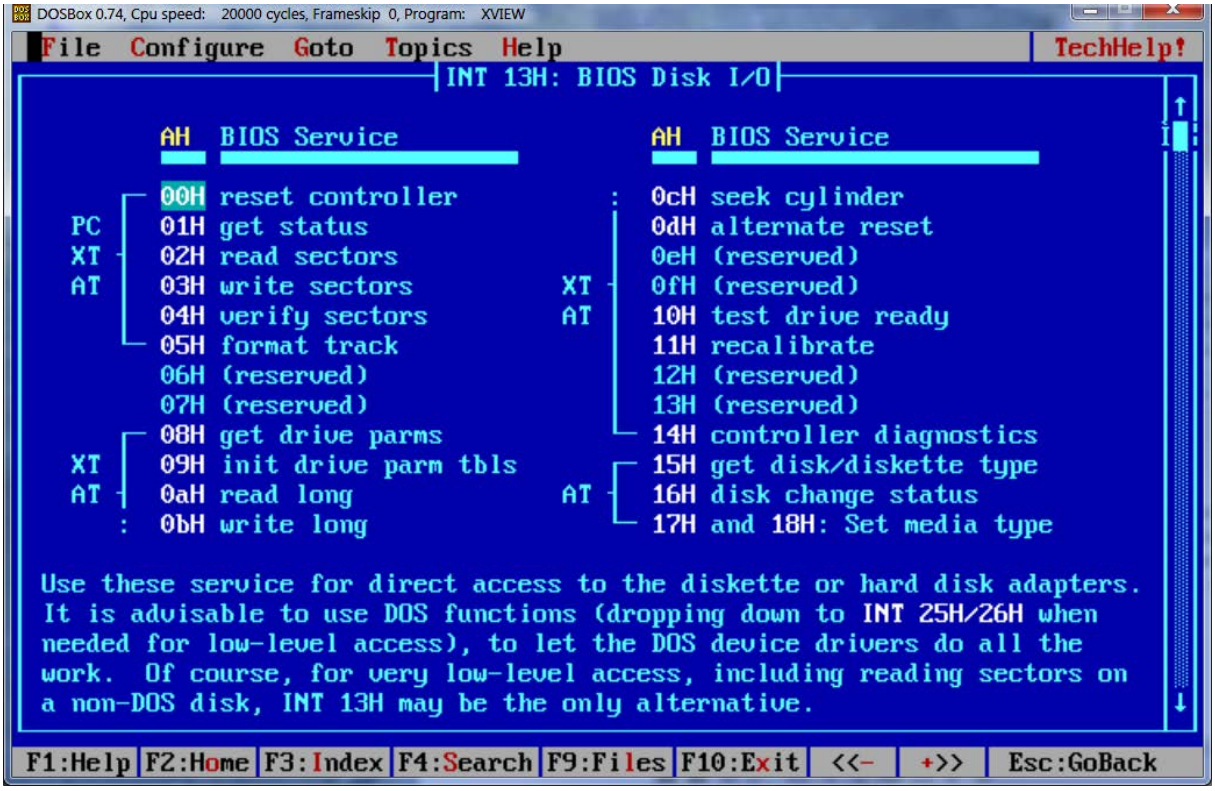
```
C:\PRACTI.CAS\PRAC-2>escr_s boot_1.bin 0 0
```

Se verá que el programa despliega enseguida su prompt `"SO> "` y que realiza perfectamente el eco del teclado, volviendo a mostrar el prompt tras cada pulsación de la tecla de retorno de carro. La única salida posible del programa es presionando la tecla de escape ESC, lo que da lugar al reinicio del ordenador.

8 ACTIVIDAD 5: Llamadas a la BIOS de acceso al disco

Vamos a tratar ahora la forma en la que debe realizarse la lectura de sectores del disquete desde el código del sector de arranque. En la práctica 1 utilizamos la función `biosdisk`, ofrecida por la biblioteca `bios.h`, para leer sectores. Ahora vamos a implementar nosotros mismos el acceso al disquete utilizando directamente llamadas a la BIOS. Comenzamos echando un vistazo a la documentación que nos proporciona el `techelp`:

La interrupción 13H de la BIOS ofrece las siguientes operaciones:

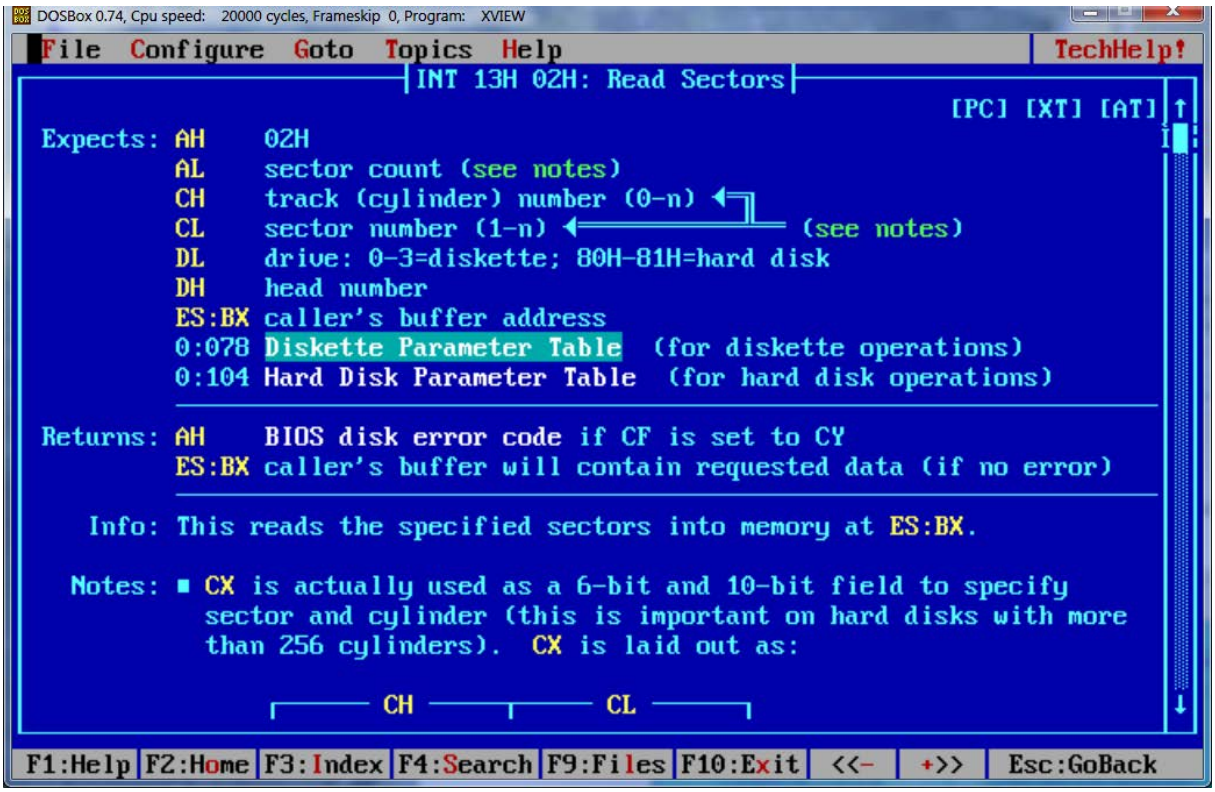


El sector de arranque necesita tan solo dos operaciones: el reset del controlador (subfunción 00H) y la lectura de un sector del disco (subfunción 02H). El número de la subfunción debe escribirse en AH antes de ejecutar la instrucción INT 13H. Por su simplicidad vamos a comenzar implementando el reset del controlador de disquete.
En el arranque es necesaria la operación de reset porque el controlador podría haberse quedado en un estado incorrecto tras la ejecución de un programa que deje colgado el ordenador. La documentación dice que aparte de poner 00H en AH, es necesario poner en DL (parte baja del registro DX) la unidad sobre la que va a realizarse el reset (0 = A:, 1 = B:, 80H = C:, 81H = D:).

```
int resetDisquete ( char unidad ) {
  asm mov dl,unidad
  asm mov ah,00H      /* 00H -> reset del controlador */
  asm int 13h         /* BIOS: E/S de disco */
```

```
  asm jc resetError   /* el flag de acarreo C se activa en caso de error */
  return(0) ;
resetError:
  return(1) ;
}
```

Pasamos a ocuparnos de la operación de lectura de un sector. La documentación nos dice lo siguiente:



Vamos a explicar un poco cada uno de los parámetros de la llamada a la BIOS para leer un sector. Está claro que en el registro DL hemos de poner el número de la unidad. En los registros DH, CH y CL hay que codificar la información correspondiente al número de cabeza, pista y sector físico del sector lógico que se desea leer. El número de cabeza (0 o 1) se pone tal cual en DH. El número de sector físico (de 1 a 18) debe ponerse en CL. Finalmente el número de pista, que sería más correcto denominar número de cilindro (de 0 a 79) debe ponerse en CH. Se apreciará que la explicación anterior parece diferir de lo que señala el `techelp`. El motivo es que el `techelp` describe el caso general en el que el número de cilindro de un disco duro puede requerir hasta 10 bits, debiendo aprovecharse los dos bits superiores de CL para contener los dos bits que no caben en CH. En el caso de un disquete, con 80 cilindros, no tenemos ese problema, de ahí que todo sea mucho más sencillo.

Siguiendo con los parámetros de la llamada a la BIOS para leer un sector, vemos que en AL hay que indicar el número de sectores consecutivos que queremos leer (sin exceder un cilindro), por lo que pondremos en AL un 1. En cuanto a la dirección de memoria donde debe quedar el sector leído, vemos que hay que indicarla en los registros ES (registro de segmento extra) y BX. En ES debe ponerse el número de segmento de la dirección, y en BX el desplazamiento. Con esto hemos terminado de explicar la información que es estrictamente necesario indicar a la BIOS en esta llamada. Como sucedía con la operación de reset, si la BIOS detecta algún error en la lectura del sector, activa el flag C de acarreo para indicarnos

ese hecho. A continuación mostramos el esqueleto de la función que realiza la lectura de un sector lógico del disquete, dejándose como tarea al alumno el completar los detalles:

```
/* leerSector: sect es el numero de sector logico                        */
/*             unidad es el numero de la unidad (0 = A:, 1 = B:)         */
/*             dir es la direccion del bufer de memoria                  */

int leerSector (int sect, char unidad, void far * dir) {

  char sector ;            /* numero de sector fisico (1..nSectPorPista) */
  char cabeza ;            /* numero de cabeza (0 o 1) */
  char pista ;             /* numero de pista (cilindro) */

  sector = ... ;           /* calculo de (s,c,p) a partir de sect */
  cabeza = ... ;
  pista = ... ;

  asm les bx,dir           /* pone en ES:BX la direccion del bufer */
  asm mov dl,...
  asm mov dh,...
  asm mov ch,...
  asm mov cl,...
  asm mov al,...
  asm mov ah,02h           /* 02H -> lectura de un sector */
  asm int 13h              /* BIOS: E/S de disco */
  asm jc errorAlLeer  /* el flag de acarreo C se activa en caso de error */
  return(0) ;
errorAlLeer:
  return(1) ;
}
```

Con el fin de comprobar que la función anterior se comporta correctamente el alumno puede ir al programa `"l_s.c"` que implementó en la práctica 1, y reemplazar la llamada a la función `biosdisk`:

```
biosdisk(cmd_read_sector, unidad, c, p, s, 1, &bufer) ;
```

por una llamada a la función `leerSector`:

```
leerSector(sectorlogico, unidad, &bufer) ;
```

A continuación debe recompilarse el programa con el `tcc` (ya que `"l_s.c"` utiliza bibliotecas como `stdio.h` dependientes de MS-DOS) y comprobarse que el programa `"l_s.c"` sigue funcionando bien a pesar del cambio realizado.

9 ACTIVIDAD 6: Manejo de direcciones de memoria

Tras el apartado anterior sabemos ya cómo podemos leer los 320 sectores lógicos correspondientes al sistema operativo. Esos sectores ocuparán 160K en memoria principal, por lo que el búfer de memoria donde leemos cada uno de esos sectores debe ir desplazándose 512 bytes tras la lectura de cada sector. Los punteros que se manejan en el 8086 normalmente son desplazamientos de 16 bits relativos al segmento de datos. Con tan solo 16 bits no es posible direccionar los 160K de memoria donde va a cargarse el sistema operativo, por lo que necesitamos direcciones completas formadas por un número de segmento y un desplazamiento, las cuales requieren 32 bits en total. En Turbo C el tipo de datos correspondiente a esas direcciones son los punteros lejanos (*far*) como por ejemplo los siguientes punteros `origen` y `destino` que apuntan a un carácter:

```
char far * pOrigen;
char far * pDestino;
```

A estos punteros lejanos se les puede asignar punteros normales (*near*), puede escribirse su valor con `printf` y el formato `%08lX`, y puede operarse con ellos con sumas y restas. Por ejemplo podemos copiar las 1000 posiciones de memoria que comienzan en la dirección segmentada `8765H:4321H` (que corresponde en el 8086 a la dirección lineal `8B971H = 87650H+4321H`) llevándolas a partir de la dirección `8900H:1234H` mediante el siguiente bucle:

```
pOrigen  = (char far*) 0x87654321L;
pDestino = (char far*) 0x89001234L;
contador = 1000 ;
while (contador-- > 0) *destino++ = *origen++ ;
```

Aunque en este caso concreto no se da el problema, hay que tener mucho cuidado de que en las sumas con punteros no se desborden los 16 bits correspondientes al desplazamiento, ya que por ejemplo si un puntero far vale `0x0000FFFF` y le sumamos 1, lo que se obtiene en Turbo C es `0x00000000`, en vez del valor esperado en el 8086 que es `0x10000000`. El alumno puede comprobarlo con el siguiente programa:

```
void main ( ) {
  char far * ptr;
  ptr = (char far *)0x0000FFFFL;
  printf(" ptr = %08lX antes de incrementarse\n", ptr++);
  printf(" ptr = %08lX despues de incrementarse\n", ptr);
}
```

Con el fin de solucionar el problema anterior al copiar zonas de memoria extensas, vamos a proporcionar una función de incremento de la dirección contenida en un puntero lejano.

```
void inc (void *p, unsigned i) {
  if (i > 0xFFFF - *(unsigned *)p) ((unsigned *)p)[1] += 0x1000;
  *(unsigned *)p += i;
}
```

Con esta función podemos realizar correctamente el incremento de la dirección contenida en un puntero lejano 'p', en una cierta cantidad 'i' (de 16 bits). Ahora el programa equivalente al anterior funciona como se espera en un 8086:

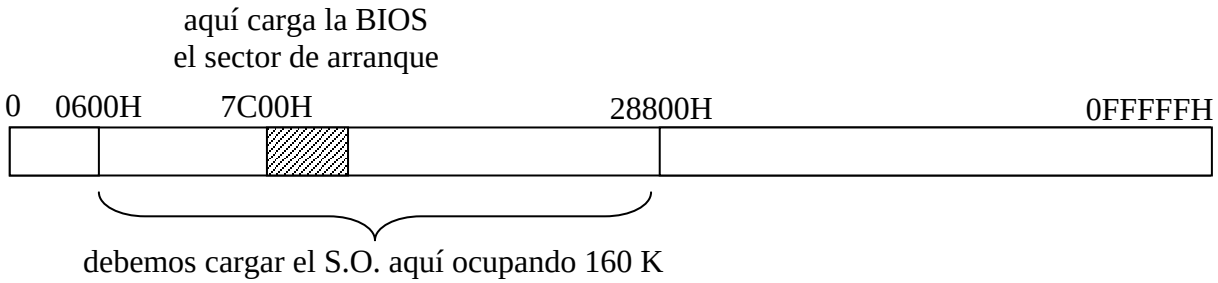
```
void main ( ) {
  char far * ptr ;
  ptr = (char far *) 0x0000FFFF ;
  printf(" ptr = %08lX antes de incrementarse\n", ptr) ;
  inc(&ptr, 1) ;
  printf(" ptr = %08lX despues de incrementarse\n", ptr) ;
}
```

Una vez que se dispone de la función auxiliar 'inc' podemos programar la carga desde el disquete de los 320 (nSectSO) sectores del sistema operativo de la siguiente manera, donde utilizamos un puntero para indicar en cada momento la dirección de comienzo del búfer:

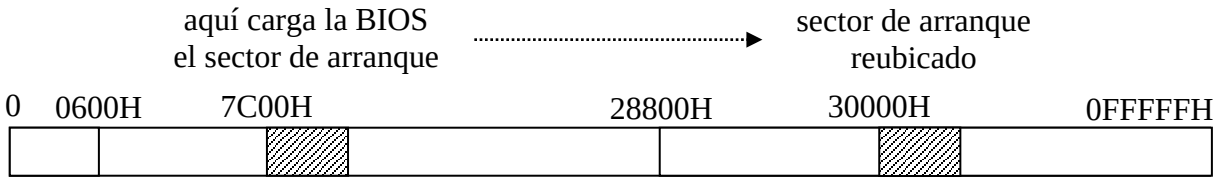
```
#define PRI_SEC_DATOS 33
char far * ptr = (char far *) 0x00600000L;    /* 0060:0000 = 00600H */
int i;
...

...
for (i = PRI_SEC_DATOS; i < nSectsSO+PRI_SEC_DATOS; i++ ) {
   leerSector(i++, 0, ptr);
   printCar('.');
   inc(&ptr, 512);
}
```

10 ACTIVIDAD 7: Implementación final del sector de arranque

Un hecho que complica la carga del sistema operativo es que la BIOS carga los 512 bytes del sector de arranque a partir de la dirección de memoria **007C00H**. El problema es que esa dirección cae dentro del rango de direcciones que queremos que ocupe el sistema operativo, como muestra la figura siguiente. Por tanto llegaría un momento durante la carga en el que el sistema operativo sobrescribiría el código del sector de arranque, lo que tendría consecuencias desastrosas.



La solución al problema consiste en que el sector de arranque se reubique desde su posición inicial a una zona de memoria segura que no sea recubierta por el sistema operativo. Esa nueva zona de memoria puede ser cualquiera a partir de la dirección 28800H, que es donde acaba el sistema operativo. Vamos a elegir para reubicar el sector de arranque la dirección 30000H que está suficientemente alejada.



Además de reubicar el sector de arranque, necesitaremos que la pila que se utilice tampoco sea sobrescrita durante la carga del sistema operativo, lo que sería también catastrófico (obsérvese que no utilizamos variables globales). Con ese fin el sector de arranque debe establecer una nueva pila en una zona segura. Vamos a elegir como valor para el segmento de pila SS el mismo que donde se copia el código del sector de arranque, es decir el 3000H. En cuanto al puntero de pila SP vamos a darle también el mismo valor 3000H, de manera que la cima de la pila estará en la dirección SS:SP = 3000:3000. El establecimiento de la pila debe hacerse en la función `start` antes de llamar a la función `main`, ya que la función **`main`** necesita la pila para asignar memoria a sus variables locales. El código que se necesita al final de la función `start` es el siguiente:

```
inicio:

  asm cli                       /* inhibimos las interrupciones */
  asm mov ax,3000h              /* Establecemos la pila en la dir. 33000H */
  asm mov ss,ax                 /* Segmento de pila SS = 3000H */
  asm mov sp,ax                 /* SS:SP = 3000:3000 = 33000H */
  asm sti                       /* permitimos las interrupciones */
```

```
  main();
```

La razón por la que inhibimos temporalmente las interrupciones (intrucción `cli`) es que la pila interviene decisivamente en la gestión de las interrupciones, que no deben aceptarse mientras modificamos la ubicación de la pila.

El punto más crítico del código del sector de arranque es el momento en el que el código del sector de arranque (cargado a partir de 07C00H) debe ceder el control a su copia (a partir de la dirección 30000H) por lo que vamos a indicar esos pasos en detalle:

```
void main ( ) {

    char far *pOrigen = (char far*)0x07C00000;  /* 0000:7C00 = 07C00H */
    char far *pDestino= (char far*)0x30000000;  /* 3000:0000 = 30000H */
    int i;

    . . .   /* Declaracion de otras variables locales que se necesiten */

    for (i=0; i < 512; i++ )  /* reubicacion del sector de arranque */
      *destino++ = *origen++;

    asm push 3000H /* Cedemos el control al sector de arranque reubicado */
    asm push OFFSET($+4)         /* justo despues de la instruccion retf */
    asm retf    /* $ = dirección a la que apunta el contador de programa */
    asm push cs     /* establecemos el segmento de datos DS = CS = 3000H */
    asm pop ds

    ...                                 /* Hacer un reset de la unidad A: */

    ...     /* Cargar el S.O. a partir de la dirección 0600H escribiendo */
            /* un punto tras la lectura de cada sector                   */

    ... /* Escribir en pantalla los caracteres: CR y LF, seguidos de SO */

    asm push 0060H  /* Cedemos el control al S.O. en 0060:0000 = 000600H */
    asm push 0000H
    asm retf

}
/* Si no se incluyen las directivas _TEXT la directiva org actua sobre
   el segmento de datos y no saldria bien */
asm _TEXT segment byte public 'CODE'
asm     org 01FEh
asm     db 55h, 0AAh
asm _TEXT ends
```

Vemos en el programa principal que lo primero es reubicar el sector de arranque. Tras la reubicación se cede el control a la copia del sector de arranque justo después de la instrucción `retf`. El signo $ hace referencia en ensamblador al contador de programa, por tanto si sumamos 4 al CP obtenemos un dirección 4 bytes posterior, que es justo la dirección de la siguiente instrucción a retf. Esto puede comprobarse examinando el código maquina con el debugger del DOS, o bien consultando un manual de ensamblador, en el que se especifica que la instrucción push OFFSET X es 68XXXX, ocupando 3 bytes y la instrucción retf es CB, ocupando un byte.

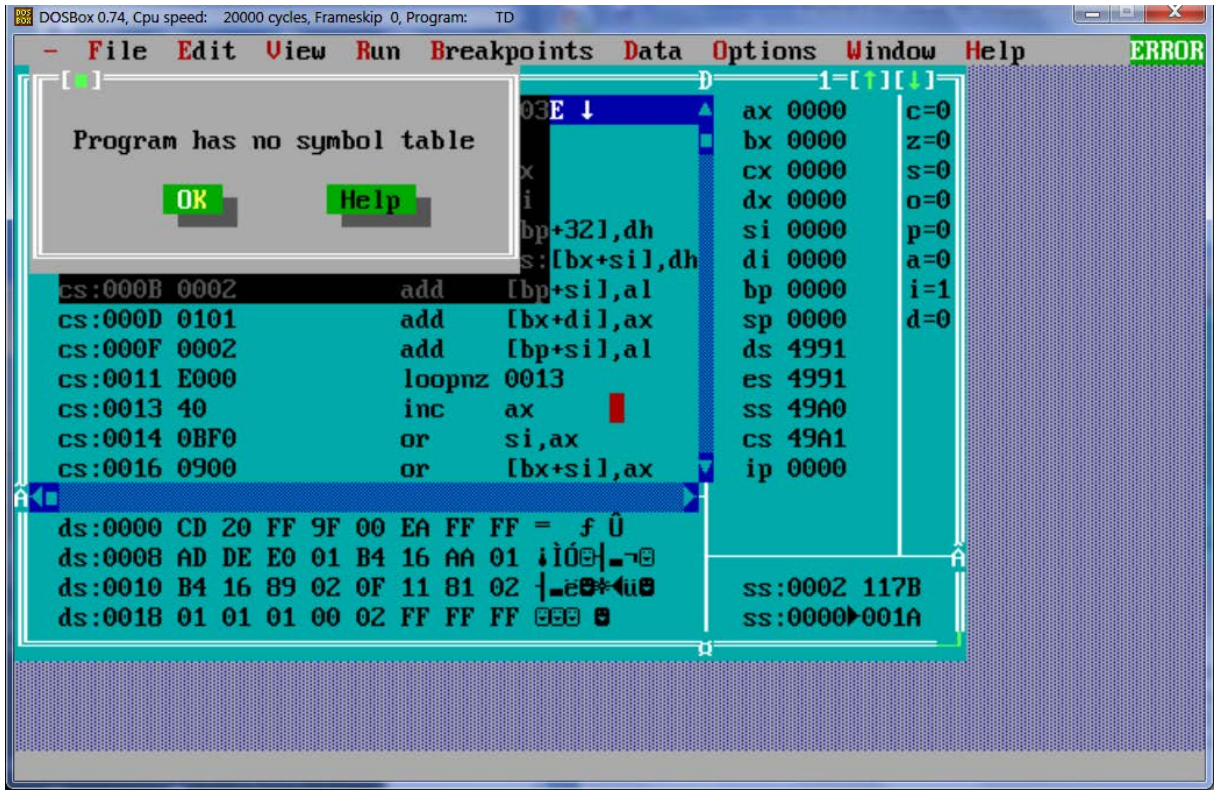
El resto de tareas que debe realizar el sector de arranque están indicadas como comentarios y se dejan como tarea para el alumno. Al final se cede el control al sistema operativo en la

dirección 00600H como se exigía en la especificación. Para ello se apila (`push`) primero el valor del segmento (0060H), luego el valor del desplazamiento (0000H) y después se ejecuta otra instrucción `retf`. La instrucción `retf` saca de la pila la primera palabra (0000H) poniéndola en el contador de programa (IP) y saca luego la otra palabra (0060H) poniéndola en el registro del segmento de código (CS), de manera que la siguiente instrucción a ejecutar será la de la dirección CS:IP = 0060:0000 que es precisamente la primera instrucción del sistema operativo. Obsérvese que hemos ajustado el valor del segmento de código a 60H para que la primera instrucción del sistema operativo corresponda al desplazamiento 0000H.
Con esto hemos terminado de explicar el código del sector de arranque compuesto por la función `start`, las funciones auxiliares (actividad 5 y 6) y la función `main`. Con todo ese código el alumno deberá crear un fichero llamado `"boot_2.c"` que incluso sin depurar va a permitirnos ilustrar el flujo de control que se produce durante el arranque. Con ese fin procedemos a compilar el programa:

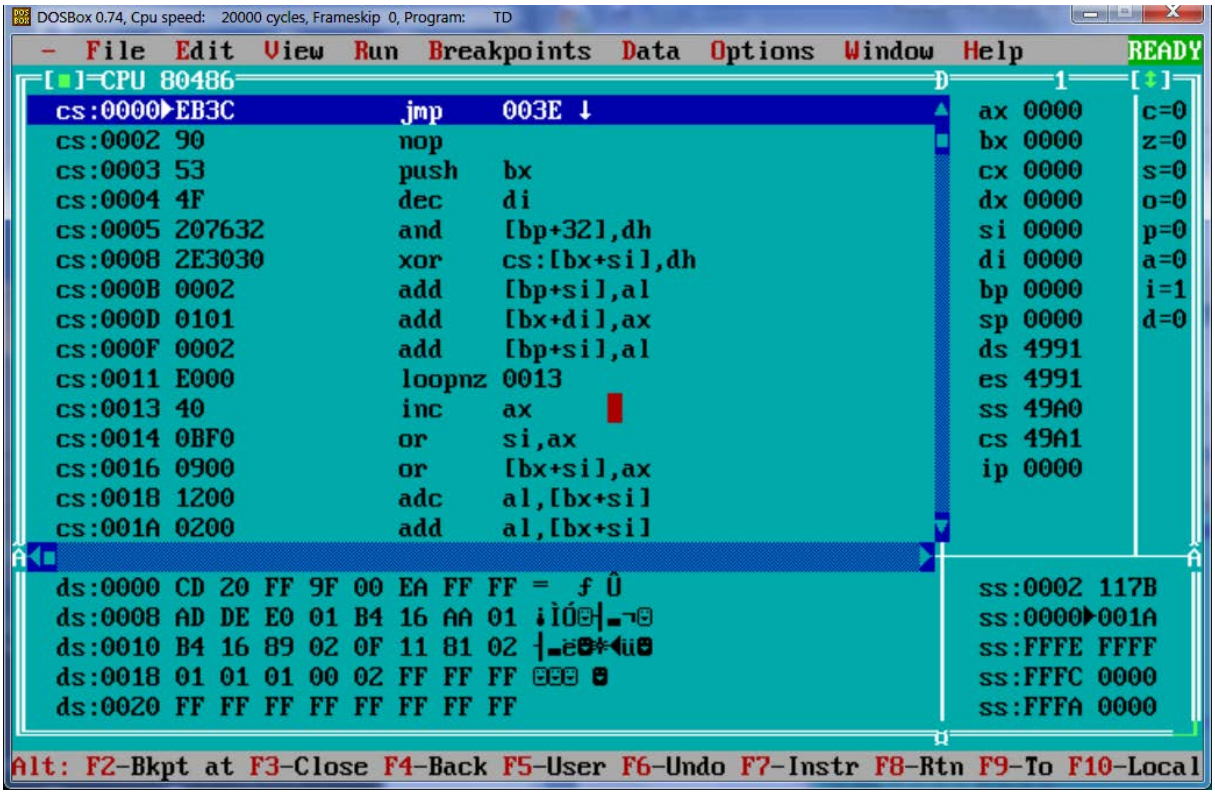
```
C:\SO1\PRACT2> compila boot_2                                    [tcc -v boot_2.c]
```

Ahora ejecutamos el depurador `td` (Turbo Debugger):

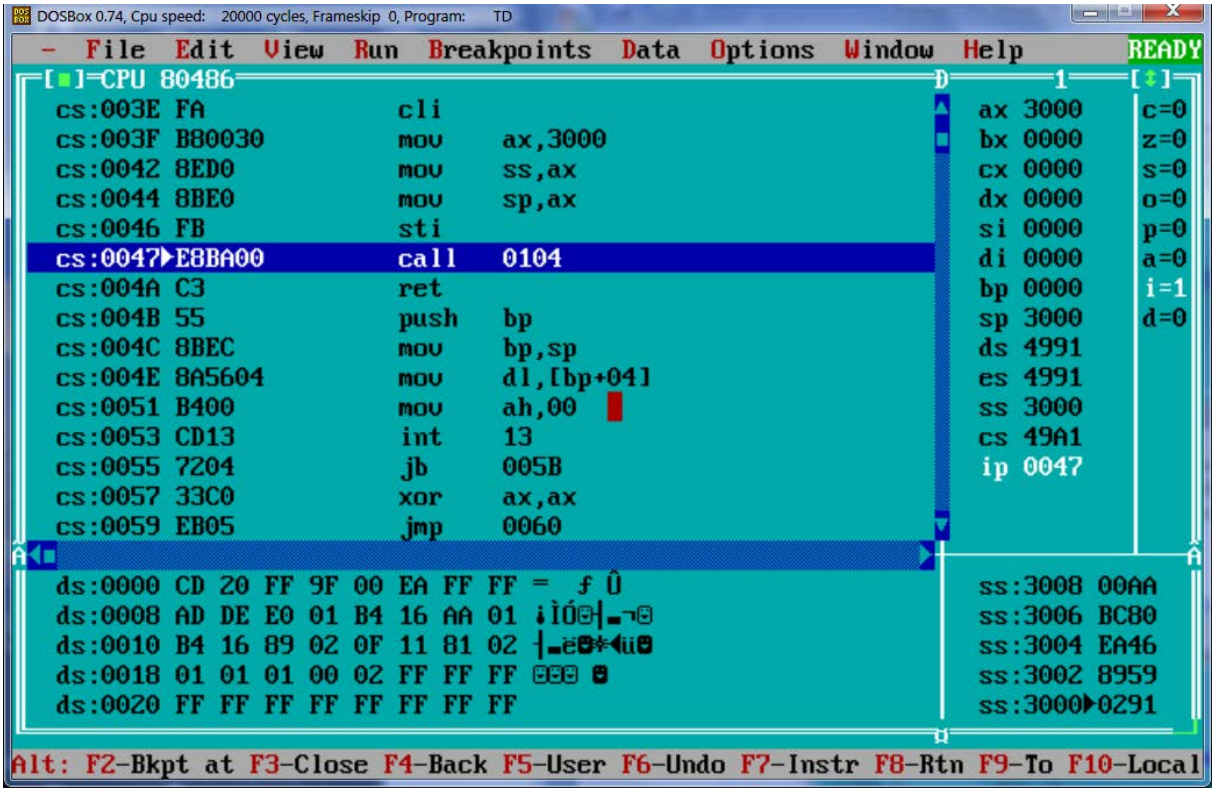
```
C:\SO1\PRACT2> td boot_2.exe
```



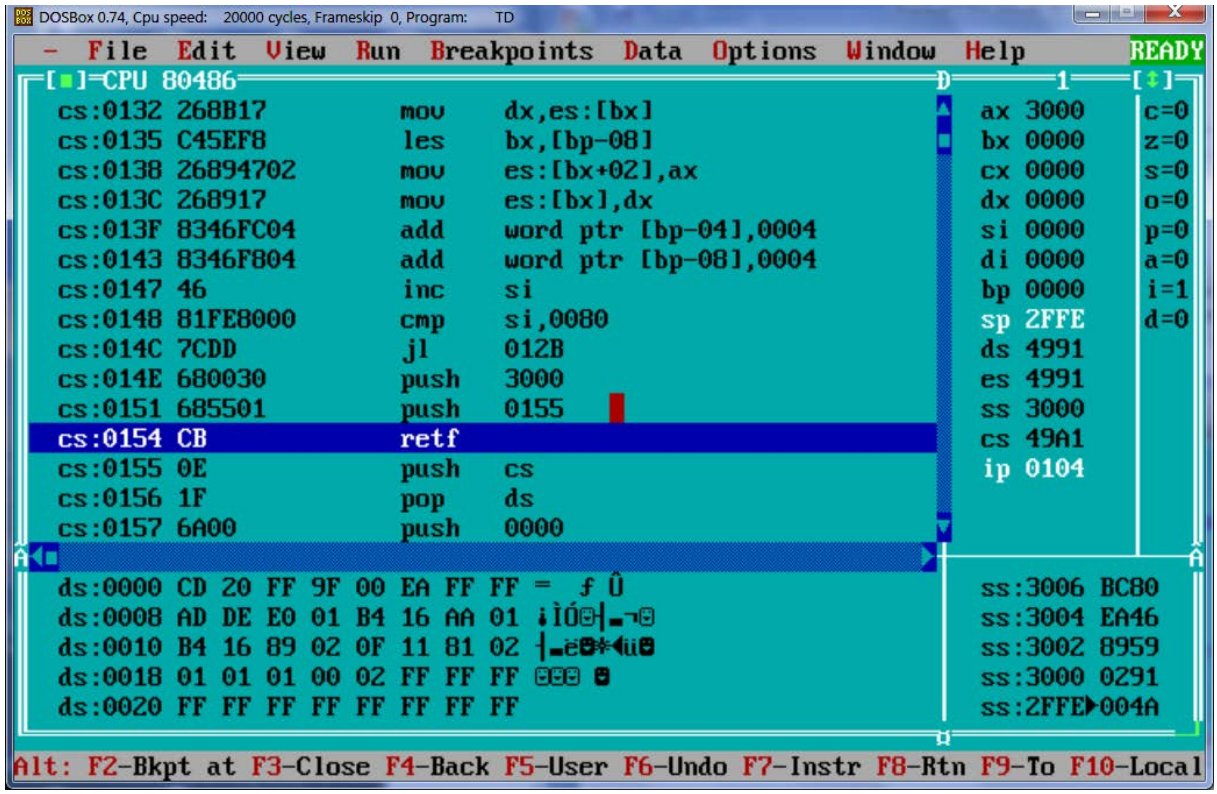
Pulsando la tecla INTRO aceptamos el mensaje. A continuación presionamos F5 para agrandar la ventana. Veremos que todo está listo para seguir la ejecución del programa a partir de la dirección 0000 del segmento de código (CS:0000). Dicha instrucción es precisamente la instrucción de salto que está al principio del sector de arranque.



Pulsando F7 se provoca la ejecución de la instrucción actual. Pulsando F8 se consigue ejecutar la instrucción actual o toda una llamada a un procedimiento (instrucción `call`) o toda una interrupción software (instrucción `int`). Vamos a ir presionando F7 hasta llegar al primer `call` que corresponde a la llamada a la función `main`.

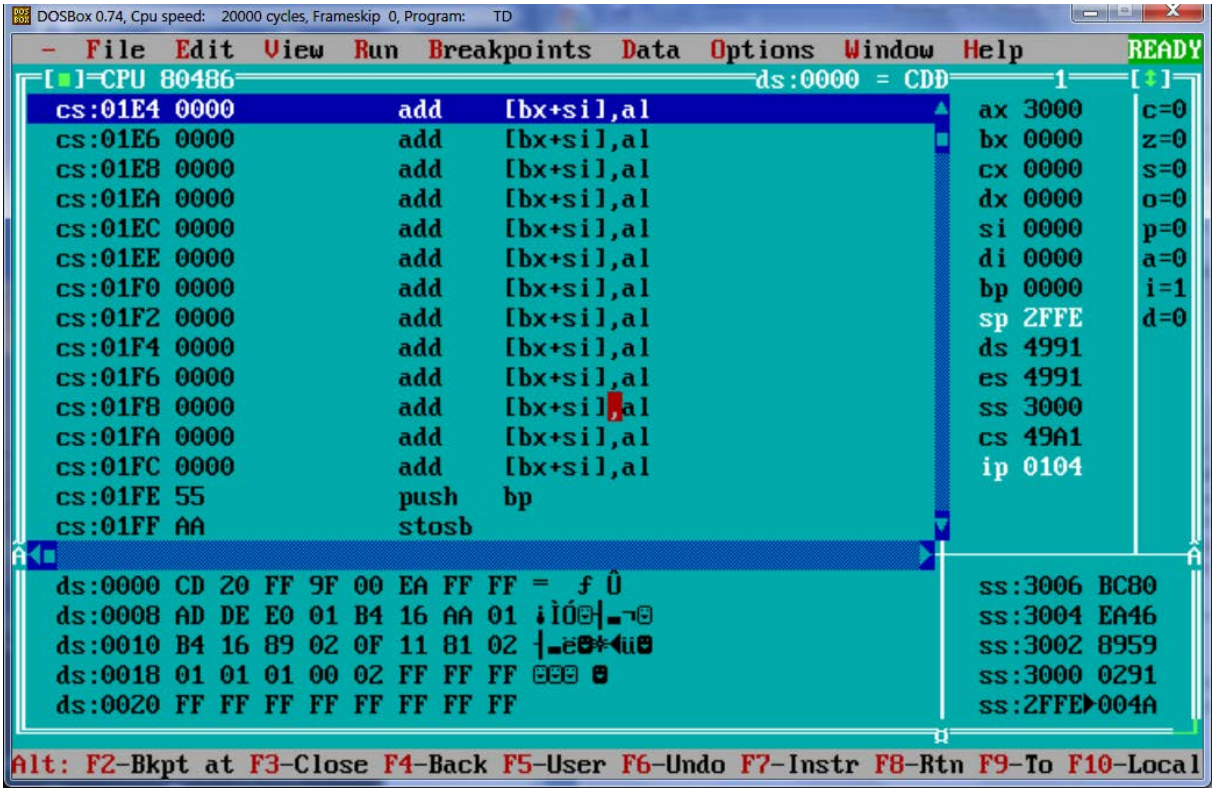


Pulsamos otra vez F7 para saltar al programa principal en la dirección CS:0104. Ya desde esa posición lo mejor es que nos limitemos a buscar la primera instrucción `retf` del programa principal utilizando la tecla de cursor abajo o `Avance de Página`.



Una vez localizada la primera instrucción `retf` vemos que la instrucción siguiente está en la dirección CS:0155, que efectivamente coincide con el segundo de los valores que se apilan (que en el programa expresamos mediante `OFFSET($ + 4)`)

Obsérvese que algo más adelante todo son ceros 0000, hasta llegar a la signatura (55 AA) del sector de arranque:



Tras las explicaciones anteriores el alumno ya puede desarrollar el sector de arranque cuya especificación se dio en el apartado 3. Se advierte de que hay que limitar al máximo el tamaño del código que se genere, ya que el sector de arranque no puede ocupar más que los 512 bytes del primer sector. En particular debe suprimirse cualquier función auxiliar que no se utilice desde el programa principal, aunque de eso ya se encarga automáticamente el compilador. Los requisitos para poder evaluarse de este ejercicio son:

- haber desarrollado completamente el fichero `"boot_2.bin"` cumpliendo todas las especificaciones del apartado 3

- haber comprobado que `"boot_2.bin"` funciona correctamente cuando se graba en un disquete de arranque con el fichero `"prog.bin"` (o cualquier otro) similar

- tener suficientemente claros todos los aspectos que recoge esta práctica

Por otra parte hay que decir que sería conveniente (si queda espacio libre en el sector de arranque) realizar alguna detección de errores a la hora de hacer el reset de la unidad A:, o a la hora de leer los sectores que contienen el sistema operativo. En caso de detectarse un error sería deseable escribir algunos caracteres indicándolo y esperar a la pulsación de una tecla para reiniciar el ordenador. En la entrega del trabajo para su evaluación podrán plantearse (en caso de duda sobre los conocimientos del alumno) cuestiones relativas a esas mejoras.

PRÁCTICA 3

Interrupciones, Excepciones y Llamadas al Sistema

ÍNDICE

1 OBJETIVOS

Los objetivos de este tutorial son:

- Que el alumno conozca en detalle cómo se implementa el modelo de los procesos sin entrar en algoritmos de planificación complejos.
- Que el alumno entienda el funcionamiento de las interrupciones y excepciones como parte del sistema operativo.
- Que el alumno entienda cómo funciona sobre una máquina concreta el mecanismo de las llamadas al sistema.
- Que el alumno entienda el funcionamiento de la interrupción de reloj al nivel necesario para implementar la llamada al sistema ***sleep***.
- Que el alumno sea capaz de implementar en un sistema operativo una herramienta de sincronización sencilla (semáforos con las operaciones: ***iniSemaforo***, ***bajaSemaforo*** y ***subeSemaforo***).
- Que el alumno escriba algún programa de usuario que haga uso de las llamadas al sistema disponibles y conozca todos los pasos necesarios para su ejecución.

2 INTRODUCCIÓN

En la primera práctica el alumno aprendió a utilizar las llamadas al sistema de gestión de ficheros desde un programa escrito en C tomando conciencia de las operaciones básicas de implementación del sistema de ficheros, que son la lectura y escritura de sectores lógicos del disco.

En la segunda práctica el alumno se enfrentó con la ingrata situación de tener que escribir un programa capaz de ejecutarse sin recurrir a un sistema operativo, y desarrolló un sector de arranque capaz de cargar desde un disquete o CDROM cualquier sistema operativo sencillo.

Una vez superados estos desafíos el alumno está preparado para dar el salto y meterse de lleno en lo que realmente es un sistema operativo. Dada la complejidad del tema no podemos pretender comenzar abordando un sistema tan acabado como MINIX. Por el contrario hemos de conformarnos con un sistema más humilde y transparente que nos permita alcanzar los objetivos expuestos en el apartado anterior. El sistema 'SO' que va a utilizarse es lo suficientemente sencillo como para poder trabajar en él con la máquina virtual "DosBox" sin necesidad de ninguna herramienta adicional, aparte del entorno de Turbo C utilizado desde la primera práctica.

3 TRABAJO A REALIZAR

Para empezar hemos de entender el funcionamiento del sistema operativo 'SO', tanto a nivel de usuario, introduciendo comandos a través de la consola o haciendo llamadas al sistema desde un programa, como a nivel de la estructura de dicho programa 'SO' en correspondencia con la de un *sistema monolítico*. En relación con este segundo nivel se mostrará la implemen-

tación concreta de los procesos (descriptores, tabla de procesos, cola de preparados, planificador y despachador) y de la gestión de memoria (particiones fijas).

A continuación se emprenderá el camino de las ampliaciones del sistema 'SO' original. Comenzaremos ilustrando la gestión de las interrupciones mediante la escritura de una rutina de tratamiento para la interrupción del teclado generada por la combinación de teclas Ctrl-C. Seguidamente trataremos brevemente las excepciones, escribiendo las rutinas de tratamiento de las excepciones de división por cero y de desbordamiento aritmético (instrucción `INTO`). A partir de ahí vamos a concentrarnos en las interrupciones software (instrucción `INT`) que son el mecanismo mediante el cual los programas de usuario invocan las llamadas al sistema.

La primera llamada al sistema que se implementará será sleep, que permite a un proceso de usuario bloquearse durante un cierto intervalo de tiempo. En su implementación el alumno deberá alterar la declaración de los descriptores de proceso, modificar la rutina de tratamiento de la interrupción de reloj y escribir el manejador de la nueva llamada al sistema.

Lo siguiente será implementar el grupo de llamadas al sistema: ***iniSemaforo***, ***bajaSemaforo*** y ***subeSemaforo***, que corresponden a las operaciones del tipo abstracto de datos de los semáforos, dotando así al sistema 'SO' de sus primeras primitivas de sincronización. Será necesario definir en el núcleo del sistema las estructuras de datos que representen a los semáforos. Los manejadores de las llamadas al sistema ***bajaSemaforo*** y ***subeSemaforo*** invocarán al planificador y dispatcher del sistema con el fin de bloquear y desbloquear correctamente a los procesos.

4 RELACIÓN DE ACTIVIDADES A DESARROLLAR

Las actividades a desarrollar son las siguientes:

Actividad 1: El sistema 'SO' a nivel de usuario

Actividad 2: El sistema 'SO' a nivel de programación de aplicaciones

Actividad 3: Estructura e implementación del sistema 'SO'

Actividad 4: Implementación de la interrupción Ctrl-C.

Actividad 5: Compilación del sistema 'SO'

Actividad 6: Implementación de excepciones

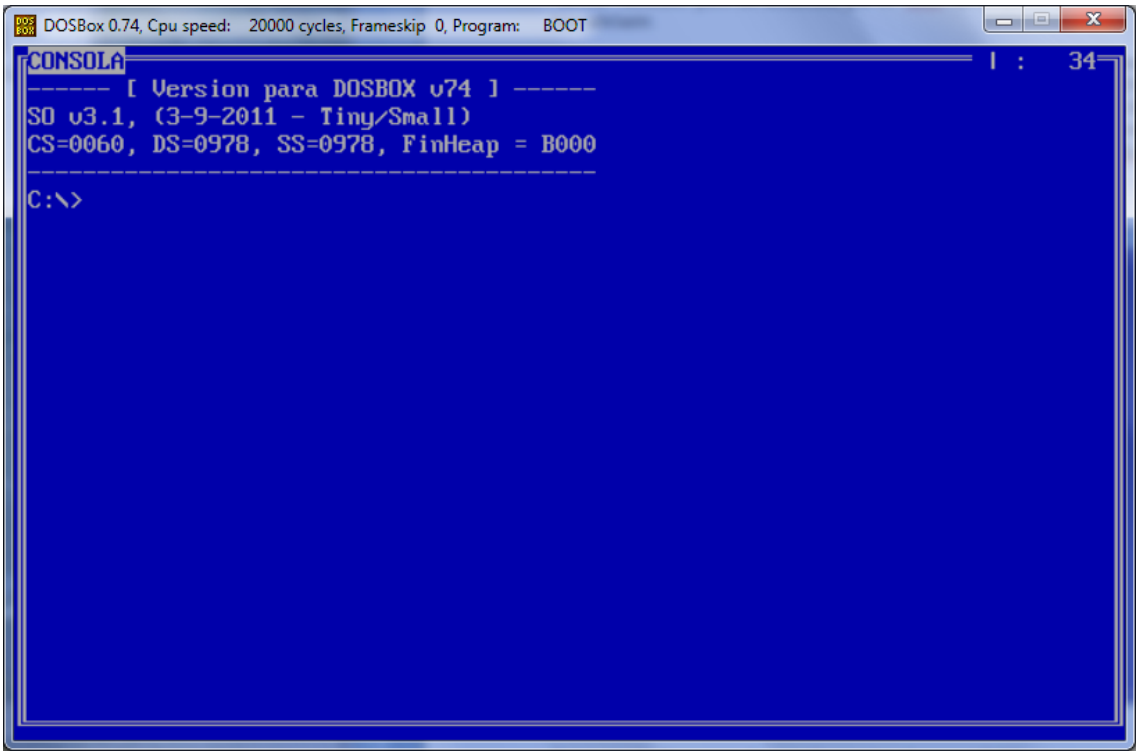
Actividad 7: Implementación de la llamada al sistema ***sleep***

Actividad 8: Implementación de los semáforos

5 ACTIVIDAD 1: El sistema ‘SO’ en el de nivel de usuario

Los fuentes del sistema ‘SO’ se encuentran en el directorio virtual “`c:\miso`". La generación del fichero ejecutable para DOS “so.exe”, en sus modalidades de depuración integrada, depuración para el Turbo Debugger y sin depuración, así como la creación del fichero binario “so.bin” utilizado para ser arrancado por el “boot” de disquete, es muy sencilla y se explicará mas adelante.

Vamos a ejecutar por primera vez el sistema ‘SO’, para lo cual primeramente arrancamos el entorno D-Fend y ejecutamos cualquiera de los dos perfiles “DBxBoot-SO” o MiBoot_SO”. Ambos perfiles inician el sistema ‘SO’ contenido en una imagen de disquete, en la cual además se encuentran algunos programas de usuario. La única diferencia entre ambas consiste en el boot de inicio del disquete. En una de ellas es el estándar de DosBox y en la otra es el boot de prácticas. Aparecerá la siguiente pantalla:



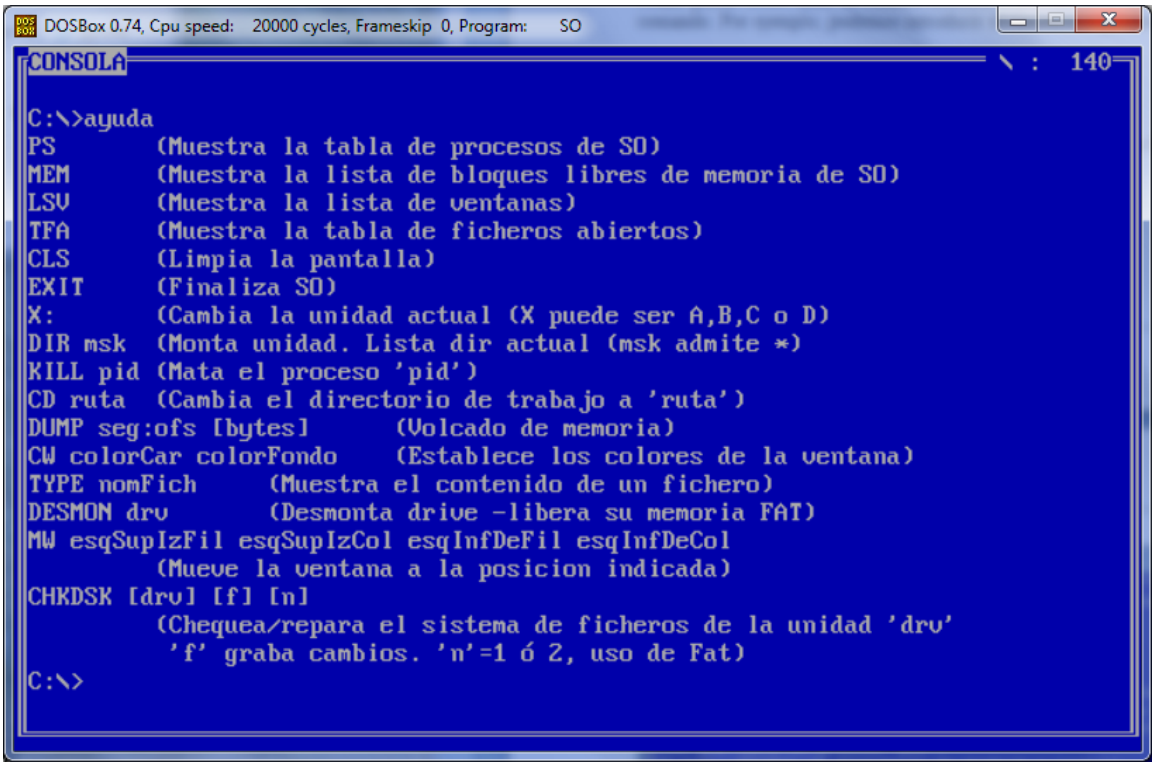
Como puede observarse, la consola muestra el *prompt* C:\> y espera a que se introduzca un comando. Por ejemplo, podemos introducir el comando siguiente:

```
C:\>cls <┘
```

que permite borrar la pantalla de la consola (*clear screen*).

Puede obtenerse la lista completa de los comandos internos de ‘SO’ introduciendo el comando “*ayuda*” ante el *prompt*:

```
C:\>ayuda <┘
```

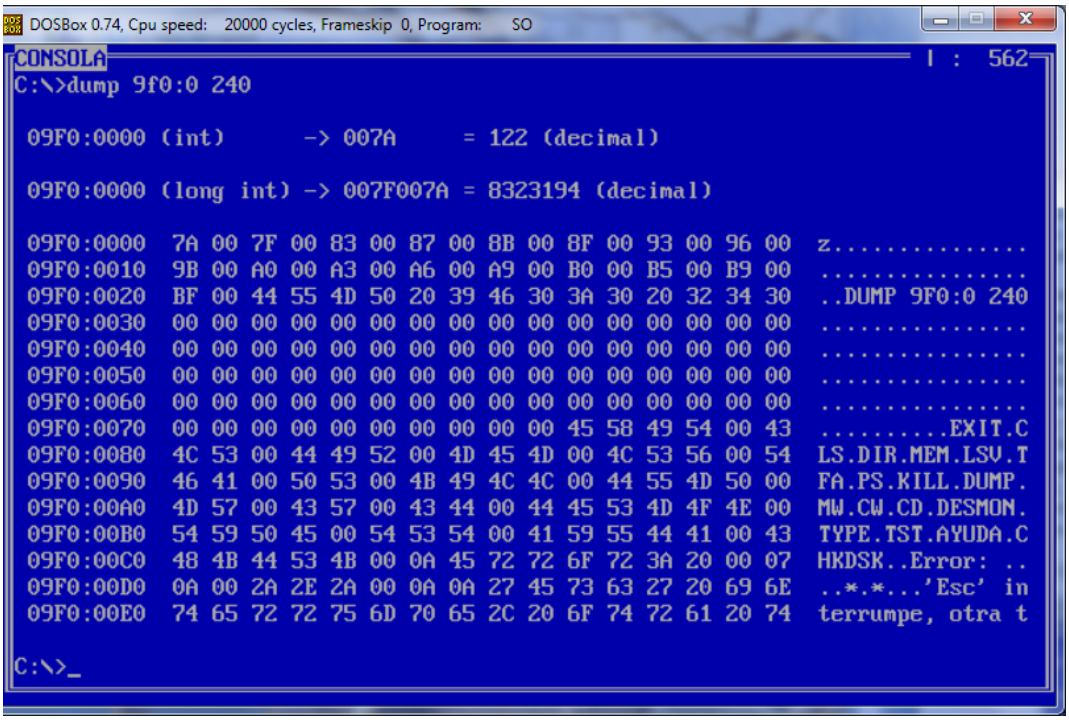


La denominación de *comandos internos* alude a que son comandos que el intérprete ejecuta cediendo el control a una función que forma parte del código del intérprete. Un *comando externo* sería uno que el intérprete ejecuta leyendo un fichero ejecutable, cargándolo en la memoria y cediéndole el control. La desventaja de los comandos externos es que tienen un peor tiempo de respuesta, al tener que leerse su código desde el disco. Su gran ventaja es que el código correspondiente a esos comandos puede actualizarse inmediatamente sin más que sustituir el fichero correspondiente por la última versión del comando, evitando así tener que recompilar el intérprete. En Linux podemos encontrar los comandos externos más utilizados en los directorios `/bin, /sbin y /usr/bin`.

'SO' es un sistema con multiprogramación, lo que queda patente ejecutando el comando `ps`, el cual muestra el estado de los procesos que están residiendo en la memoria. En este momento sólo hay dos procesos, que se corresponden con el proceso servidor (servicios relacionados con ficheros) y al intérprete de comandos de la consola de 'SO', los cuales forman parte del propio sistema 'SO' cargado en la dirección 0x00600, o segmento **0x60**.

El comando '*mem*' muestra el espacio disponible de memoria para la carga de programas en forma de una lista encadenada llamada "de huecos". Inicialmente toda la memoria a partir de donde finaliza 'SO' hasta el segmento dirección A000, estará disponible para programas y se reportará en forma de un solo hueco, por ejemplo: 2A5D (comienzo segmento), 85A3 (tamaño del hueco en "*paragraphs*" - unidades de 16 bytes) y por último la dirección del siguiente hueco: NIL.

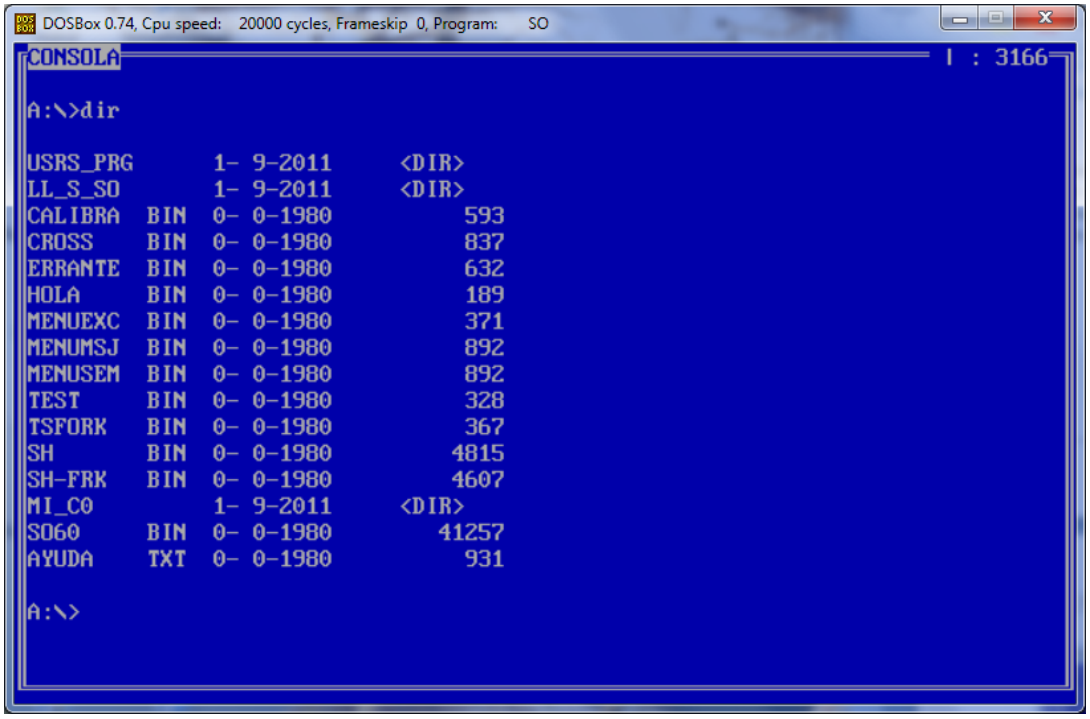
El comando '*dump*' (volcado de memoria) permite visualizar partes de la memoria del mismo modo que el programa "*ver_fich.exe*" visto anteriormente mostraba partes de un fichero binario. Si tecleamos por ejemplo el comando '*dump*' siguiente, obtendremos una pan talla similar a la siguiente:



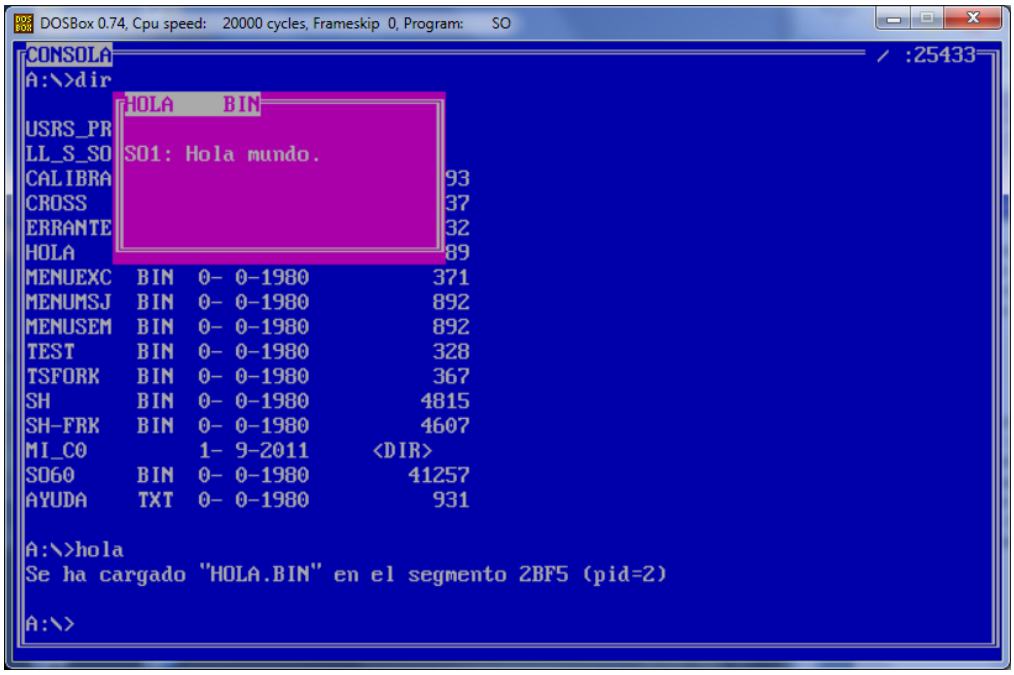
La dirección **`09F0:0`** es el principio del segmento de datos (DS) del programa 'SO'.

Como curiosidad en la columna de la derecha se reconocen varias cadenas de caracteres (cuyo fin está indicado por el byte 00) como son las correspondientes a los comandos internos mostrados por el comando "ayuda".

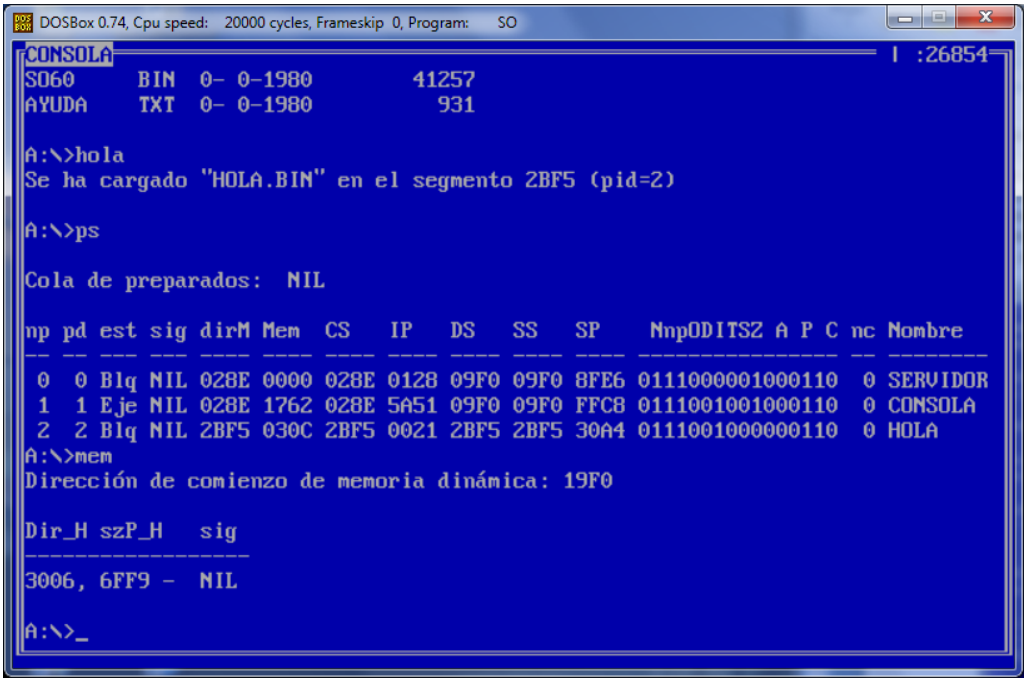
Veamos ahora cómo podemos ejecutar algún programa presente en el sistema de ficheros, el cual podría implementar algún comando externo de 'SO'. Con ese fin lo más rápido es cambiar primero la unidad actual por la "A", para ello tecleamos A:<intro> y seguidamente usamos el comando interno *dir* el cual nos mostrará el contenido del disquete:



En este caso vemos que hay varios ficheros en el disquete, uno de ellos: "so60.bin", se corresponde con el sistema 'SO' cargado durante el arranque, los demás ficheros son programas de usuario que hacen uso de diferentes llamadas a sistema, algunas de las cuales habrá que realizar en estas prácticas. Si ejecutamos el programa "hola", obtenemos lo siguiente:

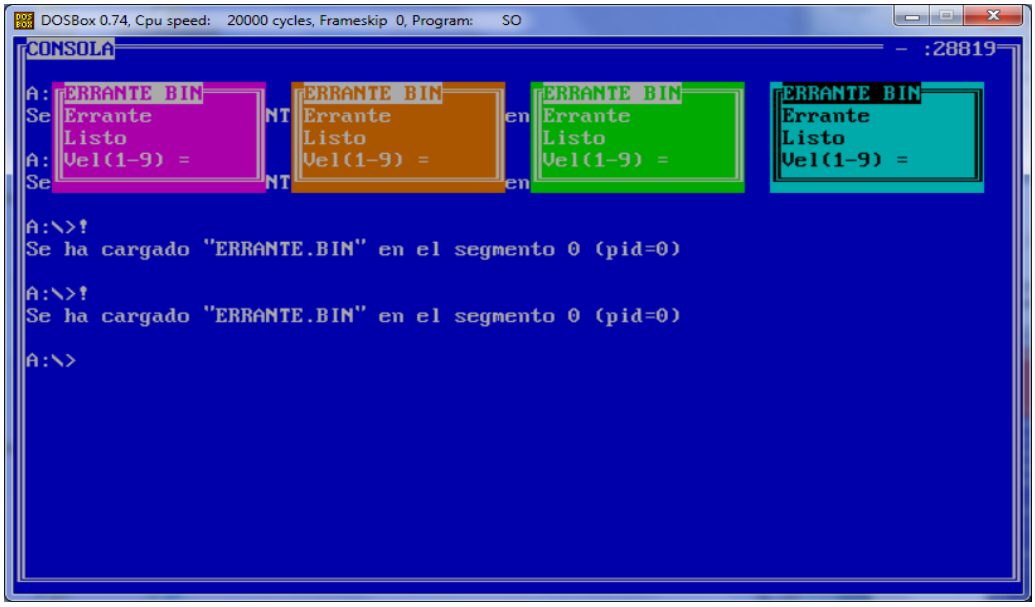


Vemos que el sistema 'SO' ha cargado el fichero "*hola.bin*" desde el disquete y lo ha ubicado en el segmento **2BF5**, creando un nuevo proceso cuyo ***pid*** (identificador de proceso) es 2. Además se ha abierto una ventana (violeta en este caso) que sirve de vía de comunicación entre el usuario y el proceso. La ventana no tiene el foco del teclado ya que el cursor "_" lo sigue mostrando la consola. Para tomar el foco del teclado pulsamos la tecla **TAB** que sirve para cambiar dicho foco entre todas las ventanas (procesos) en ejecución. Una vez retomado el foco, si ejecutamos los comandos '*ps*' y '*mem*'. Veremos la siguiente pantalla:

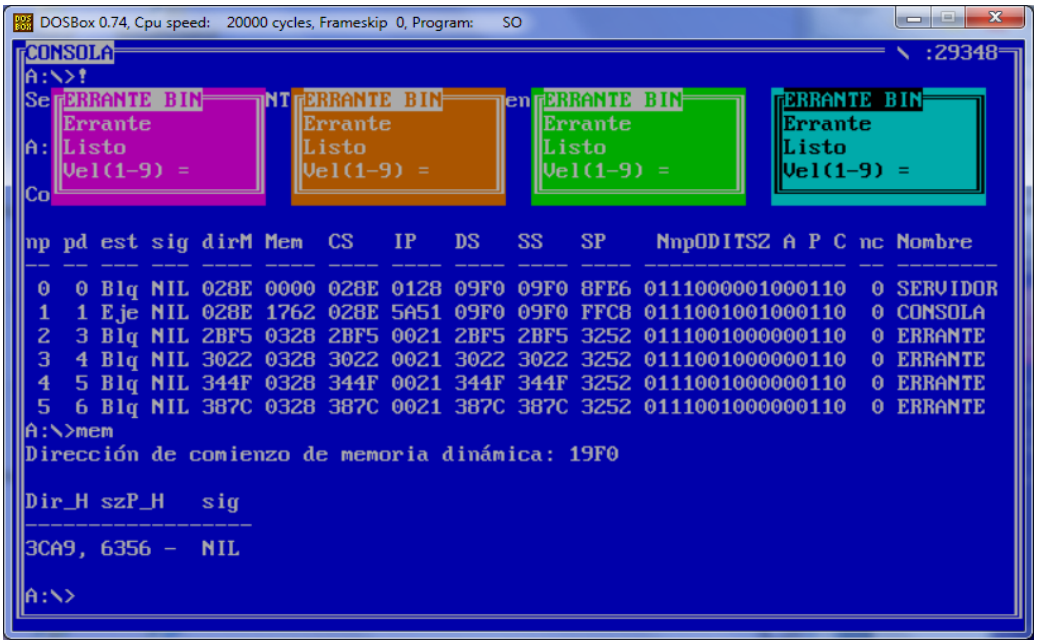


Las combinaciones de teclas **SHIFT IZDA-TAB** y/o **SHIFT DCHA-TAB** sirven para cambiar el plano de la ventana sin cambiar el foco del teclado. Si arrancamos mas procesos podemos probar estas teclas y comprobar su funcionamiento. El programa "*hola*" finaliza con la pulsación de una tecla, pudiendo comprobarse con '*ps*' cómo desaparece el proceso y con '*mem*' si se ha generado un nuevo hueco en el lugar que ocupaba dicho proceso o se ha fusionado con alguno ya existente. Por último, respecto al uso de teclas especiales, hay que decir que las teclas de cursor "⇦", "⇨", "⇧" y "⇩", sirven para mover la ventana que aparece en primer plano y si se usan conjuntamente con la tecla **SHIFT** *(Mayus)* redimensionan la ventana.

Vamos a mostrar ahora la ejecución de varios procesos concurrentemente. Con este fin introducimos desde la consola cuatro veces el comando externo "*errante*" (el carácter '**!**' tiene la función especial de repetir el último comando). Recuérdese el uso de **TAB** para retomar el foco y las teclas de flecha de cursor para mover las ventanas:



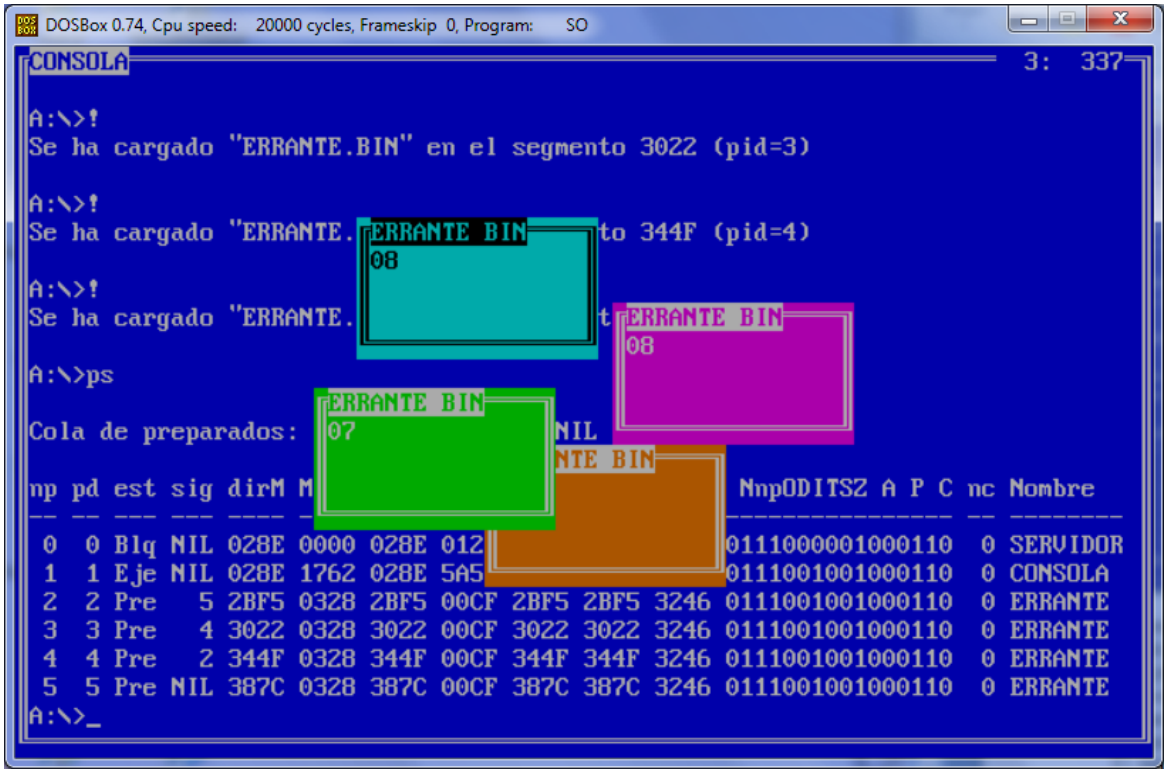
Si ejecutamos los comandos '*ps*' y '*mem*' obtenemos:



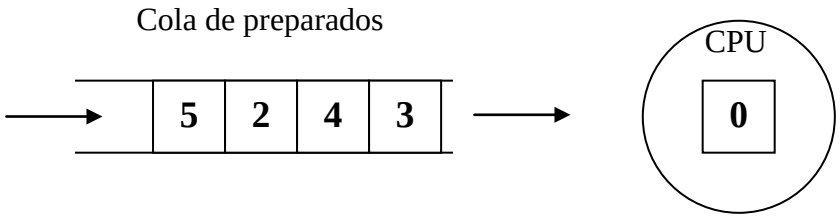
Pueden apreciarse los espacios de memoria que han ido ocupando los procesos "*errante*", la memoria que ocupa cada uno de ellos y el hueco de memoria final resultante.

La función del programa "errante" es muy simple, se limita a solicitar un número entre 1 y 9 y a continuación comienza a desplazarse por la pantalla a una determinada velocidad que viene indicada por el número introducido, cuanto más alto más rápido se mueve. Este programa hace uso reiterado de la llamada a sistema "*moverWindow*", y tras realizar una serie de vueltas, acaba parado en la esquina inferior izquierda, muestra un mensaje y finaliza.

Con los programas "errante" cargados en memoria y listos para empezar su función, tecleamos en cada uno de ellos un número entre 1 y 9 y seguidamente vamos a la consola y ejecutamos el comando '*ps*', obteniendo algo similar a esta pantalla:



Es interesante observar el estado de los procesos mientras se ejecutan haciendo uso del comando '*ps*' desde la consola. En la pantalla anterior se aprecia que se estaba ejecutando la consola (procesando el comando '*ps*') y que el resto de procesos estaban preparados para ejecutarse, para proseguir con la ejecución de su movimiento. Podemos deducir el estado de la cola de preparados a partir del valor de los campos '`sig`' de los descriptores de los procesos. Vemos que el siguiente del proceso ***np = 5*** es NIL, lo que indica que se trata del último proceso preparado. El siguiente del proceso 2 es 5, lo que indica que el proceso 2 es el penúltimo de la cola. El siguiente del proceso 4 es 2, por lo que el proceso 4 es el segundo proceso de la cola. Finalmente el proceso 3 tiene como siguiente al proceso 4, por lo que el proceso 3 es el primer proceso de la cola de preparados.



El comando '*kill*' permite destruir un proceso identificado por su '*pid*'. Por ejemplo desde la consola podemos matar cualquier proceso errante antes de que finalice éste, ejecutando por ejemplo: `c:\>kill 3 <↵`

6 ACTIVIDAD 2: El sistema 'SO' en el nivel de programación de aplicaciones

Esta actividad se realizará usando principalmente el perfil de D-Fend "DBOX-SO", en el que tenemos el entorno de desarrollo ya familiar de TC30 y demás utilidades, y con el que se han desarrollado las prácticas anteriores, aunque en esta ocasión utilizaremos fundamentalmente el directorio *c:\miso* donde se encuentra el fuente del sistema 'SO', así como también el subdirectorio *usrs_prg*, en el que se encuentran los programas de usuario necesarios y el subdirectorio *ll_s_so* con los ficheros que forman la librería de interfaz de llamadas al sistema.

A nivel de programación de aplicaciones, el sistema 'SO' nos ofrece el repertorio de llamadas contenido en los ficheros del directorio "*c:\miso\ll_s_so*", que podemos utilizar desde cualquier programa diseñado para ejecutarse en el sistema 'SO' como un proceso de usuario. Como ejemplo a continuación se muestra el contenido del fichero "*bsic-ifz.c*", el cual contiene un conjunto de llamadas básicas generales:

```
/* -------------------------- finProceso() ----------------------------
   Esta llamada al sistema es llamada cuando un proceso quiere finalizar
   ---------------------------------------------------------------------- */
void finProceso (void) {
  asm { MOV AH,0; INT VINT_SO }
}

/* ------------------------ matarProceso()-----------------------------
   Esta llamada al sistema elimina al proceso cuyo identificador de
   proceso es pid. No se permite matar al proceso con pid == 0 que
   corresponde a la consola del sistema SO. La ventana del proceso se
   cierra y la memoria que ocupa el proceso queda libre para cargar otro.
   Si se detecta algun error la funcion devuelve un -1, sino 0.
   ---------------------------------------------------------------------- */
int matarProceso (word_t pid) {
  asm { MOV AH,1; MOV BX,pid; INT VINT_SO }  return _AL;
}

/* ------------------------ leerTecla() -------------------------------
   Esta llamada al sistema lee un caracter del teclado virtual del
   proceso. Cada proceso tiene su propio teclado virtual que corresponde
   al teclado fisico cuando la ventana del proceso esta seleccionada.
   Si al ejecutar leerTecla() no hay ningun caracter disponible en el
   bufer del teclado fisico o en el bufer del teclado virtual, el proceso
   que hace la llamada se bloquea hasta que se presione una nueva tecla
   estando la ventana del proceso seleccionada. La funcion devuelve como
   resultado el caracter ascii correspondiente a la tecla pulsada.
   ---------------------------------------------------------------------- */
```

```
char leerTecla (void) {
  asm { MOV AX,0x0200; INT VINT_SO } return _AL;
}

char leerTeclaLista (void) {
  asm { MOV AX,0x0201; INT VINT_SO } return _AL;
}

/* ------------------------- printCar() ------------------------------
   Esta llamada al sistema escribe un caracter ascii en la ventana del
   proceso en la posicion actual del cursor.
   -------------------------------------------------------------------*/
void printCar (char car) {
  asm { MOV AH,3; MOV BL, car; INT VINT_SO }
}

#define printDec(num,l) printBase(num,10,l)
#define printHex(num,l) printBase(num,16,l)
#define printStr(str)   printStrHasta(str,0xFFFF)

void printStrHasta (char far *str, word_t lon) {
  asm { MOV AH,4; LES SI,str; MOV CX,lon; INT VINT_SO }
}

void printBase (word_t num, byte_t base, byte_t lon){
  asm { MOV AH,5; MOV AL,base; MOV BX,num; MOV CL,lon; INT VINT_SO }
}

/* ------------------------- moverWindow() ---------------------------
   Esta llamada al sistema cambia los limites de la ventana del proceso
   correspondientes al numero de la primera fila, el numero de la primera
   columna, el numero de la ultima fila y el numero de ultima columna.
   Los numeros de fila deben de estar en el rango 0..24 y los numeros de
   columna deben estar en el rango 0..79. En otro caso o si la ventana
   resulta ser demasiado peque¤a, la funcion no tiene ningun efecto.
   Tras moverse la ventana se borra su contenido. Devuelve 0 si se pudo
   pudo llevar a cabo la operacion con exito o 0 en caso contrario.
   ------------------------------------------------------------------- */
void moverWindow (char esqSupIzF, char esqSupIzC,
                  char esqInfDeF, char esqInfDeC) {
  asm { MOV AH,6;
        MOV BL,esqSupIzF;  MOV BH,esqSupIzC;
        MOV CL,esqInfDeF;  MOV CH,esqInfDeC; INT VINT_SO }
}

/* ------------------------- colorWindow() ---------------------------
   Esta llamada al sistema cambia el color que se utiliza al visualizar
   los caracteres en la pantalla, tanto el color del caracter en si como
   el color de fondo utilizado. Los valores concretos de los colores
   pueden encontrarse en el fichero "colores.h". Esta funcion no devuelve
   ningun resultado.
   ------------------------------------------------------------------- */
void colorWindow (char colorCar, char colorFondo) {
  asm { MOV AH,7; MOV BH,colorCar; MOV BL,colorFondo; INT VINT_SO }
}

void fotoWin (bool salvar) {
  asm { MOV AH,8; MOV BL,salvar; INT VINT_SO }
}

// --- Funciones auxiliares de SO. Utilizan otro vector ----
```

```
void leerLinea (char far *lin, word_t size, bool mayus) {
  asm { MOV AH,0; LES BX,lin; MOV CX,size; MOV AL, mayus; INT VINT_SO+1 }
}
```

La implementación de estas funciones de interfaz de llamadas al sistema se limita a:

- Pasar los parámetros que reciben en la pila a los registros donde el sistema operativo 'SO' espera encontrarlos,
- Pasar al sistema operativo el código de la llamada al sistema en cuestión en el registro `AH` (byte alto del registro acumulador `AX`) del 8086,
- Realizar la llamada al sistema operativo 'SO' mediante la correspondiente instrucción `INT` del 8086, y
- Devolver, tras el retorno de la llamada, los resultados que corresponda.

Es conveniente que el alumno relacione esto con lo visto en la teoría, por lo que se reproduce aquí la figura 1-17 del libro de *Tanenbaum*.

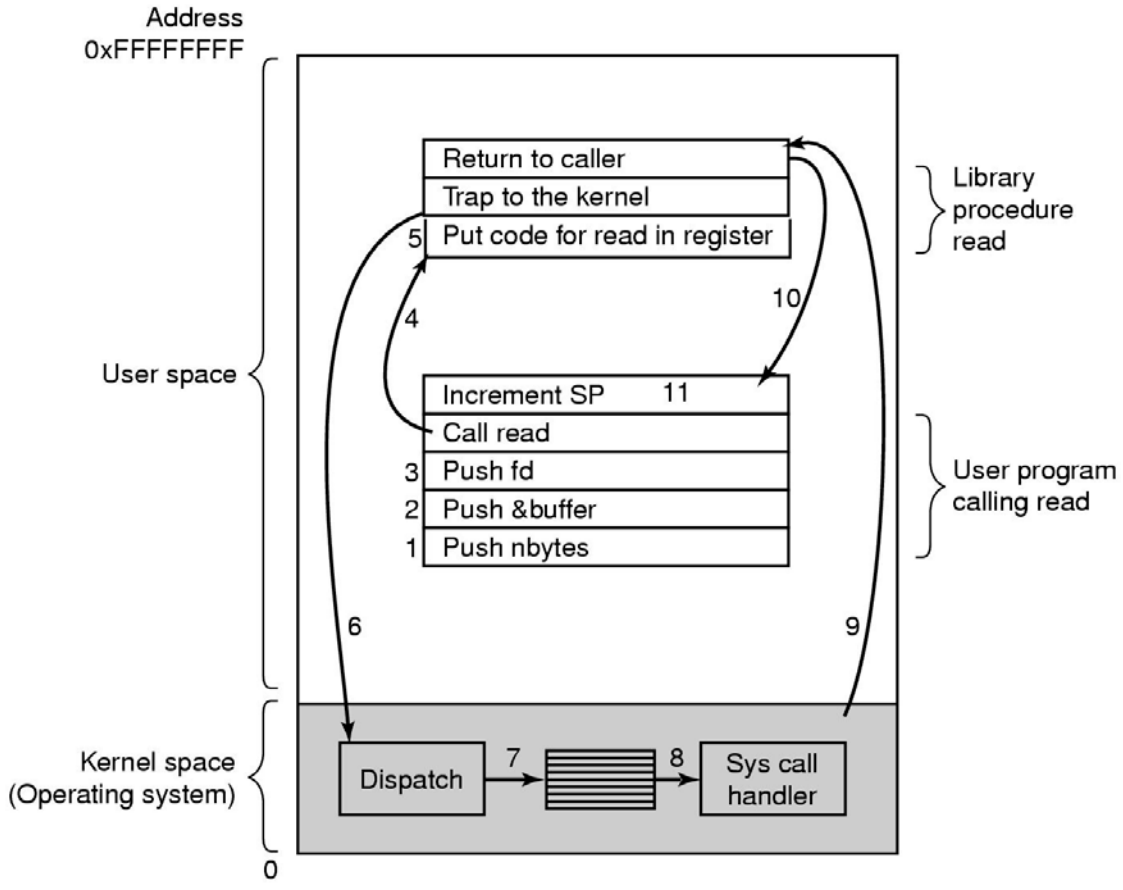


Figura 1-17. Los 11 pasos para hacer la llamada al sistema read(fd, &buffer, nbytes).

Si vemos ahora por ejemplo la implementación del procedimiento de librería `matarProceso`, seguramente quedará todo más claro:

```
int matarProceso (word_t pid) {
  asm { MOV AH,1; MOV BX,pid; INT VINT_SO }  return _AL;
}
```

El paso 5 de la figura corresponde a la instrucción `MOV AH,1` ya que el sistema 'SO' interpreta el código 1 como que el programa de usuario desea llevar a cabo la llamada al sistema *matarProceso*, la cual mata a un proceso. El paso 6 de la figura corresponde a la ejecución de la instrucción `INT VINT_SO`, donde `VINT_SO` es un símbolo definido como `0x60`, que es el número del vector de interrupción utilizado por el sistema 'SO'. Finalmente el paso 10 de la figura se corresponde con la instrucción de retorno `return _AL`.

Otro aspecto a señalar respecto a las llamadas al sistema y procesos de usuario es el fichero "*inic_usr.c*". Este fichero contiene el código de inicialización necesario para generar los programas de usuario, así como el interfaz de llamadas al sistema básico, el cual se haya incluido dentro de él mediante la directiva `#include "..\ll_s_so\bsic-ifz.c"`.

Su contenido es el siguiente:

```
/* ------------------------------------------------------------ */
/*      Funciones de interfaz de llamadas al sistema básicas    */
/* ------------------------------------------------------------ */

asm DGROUP GROUP _TEXT, _DATA
/* Esta directiva es vital para que el compilador genere codigo en
   el que las variables, que usan el segmento DS, tomen un offset
   adecuado, es decir, para que DS y CS sean iguales (mismo segmento).
   Si se omite, se asume que DS va a tomar un valor distinto a CS,
   concretamente CS + tama¤o codigo en paragraphs, o sea, modelo small.
   Todo esto es debido a que compilamos desde linea de comando no
   desde el entorno integrado y sin el codigo inicial C0.obj, el cual
   se encarga entre otras cosas de ello. Los offset de las variables
   empienzan donde acaban los de las funciones. Si se usa TCC 3.0 hay
   que usar la opcion -B para que use TASM en vez de el de inline BASM */

#include "..\tipos.h"

void main (void);             /* declaraciones forward */
void finProceso (void);

/* ----------------------------- start() ---------------------------------
   Esta funcion debe estar situada al principio de cualquier otra funcion
   de todo proceso de usuario, ya que en ella se invoca a la funcion main()
   y despues se efectua la llamada a sistema de finalizacion de proceso */
void start (void) { main(); finProceso(); }
```

#include "..\ll_s_so\bsic-ifz.c"

Este fichero contiene la función `start`, la cual debe ser la primera función en el fichero y dado que hace uso de las funciones `main` y `finProceso`, éstas deben de estar declaradas anteriormente como prototipos de 'C'. El propósito de ***start*** es servir de código inicial de todo proceso de usuario. Este código inicial lo que hace es invocar a la función `main` y tras finalizar ésta, realizar la llamada al sistema `finProceso`, de este modo las aplicaciones de usuario tienen garantizado arrancar por la función `main`, y también que al finalizar ésta, se invoque la llamada al sistema `finProceso`. Por supuesto es condición necesaria que los programas de

usuario tengan al principio de todo, como su primer directiva “include”, precisamente la de este fichero: *#include* "..*ll_s_so**inic-usr.c*" , ya que es el modo más sencillo de garantizar que la primera función que el compilador encuentre sea precisamente la función `start`. Por último conviene aclarar un poco el uso de la directiva `asm DGROUP GROUP _TEXT, _DATA` que aparece en el fichero “*inic-usr.c*”. Esta directiva se usa para informar al compilador que debe usar un solo segmento para código y datos, al igual que se hace en el modelo ‘***tiny***’ de ‘turboC’. Con ese modelo se simplifica el funcionamiento del arranque de las aplicaciones de usuario por parte de ‘SO’, ya que los segmentos CS, DS y SS tienen así el mismo valor inicial que es la dirección de memoria donde se carga del proceso.

Una vez presentada la interfaz de programación con el sistema operativo es conveniente que nos ejercitemos en su utilización poniendo a punto un programa de usuario para el sistema operativo ‘SO’. Como viene siendo habitual, vamos a empezar por el programa de usuario para el sistema ‘SO’, “*Hola mundo*”, cuyo fuente se encuentra en el fichero “*hola.c*” dentro del directorio: “*c:\miso\usrs_prg\hola*”. Una cosa que debe quedar clara es que los programas ejecutables de ‘SO’, en general, no van a funcionar en otros sistemas operativos, ya que las llamadas al sistema en los que se basan hacen uso de la instrucción de *interrupción software* (*trap*) `INT 60h`, y el vector de interrupción `60h`, no tiene porqué soportar las funciones de servicio de llamadas al sistema características de ‘SO’. Seguidamente se muestra el código de este programa:

```
/* ------------------------------------------------------------------ */
/*                              hola.c                                */
/* ------------------------------------------------------------------ */
/*                 programa Hola mundo para el sistema ‘SO’            */
/* ------------------------------------------------------------------ */

#include "..\ll_s_so\inic-usr.c" //Cod.inic.prog.usuario y llam. basicas

void main () {
  printStr("\nSO: Hola mundo.\n") ;
  leerTecla() ;
}
```

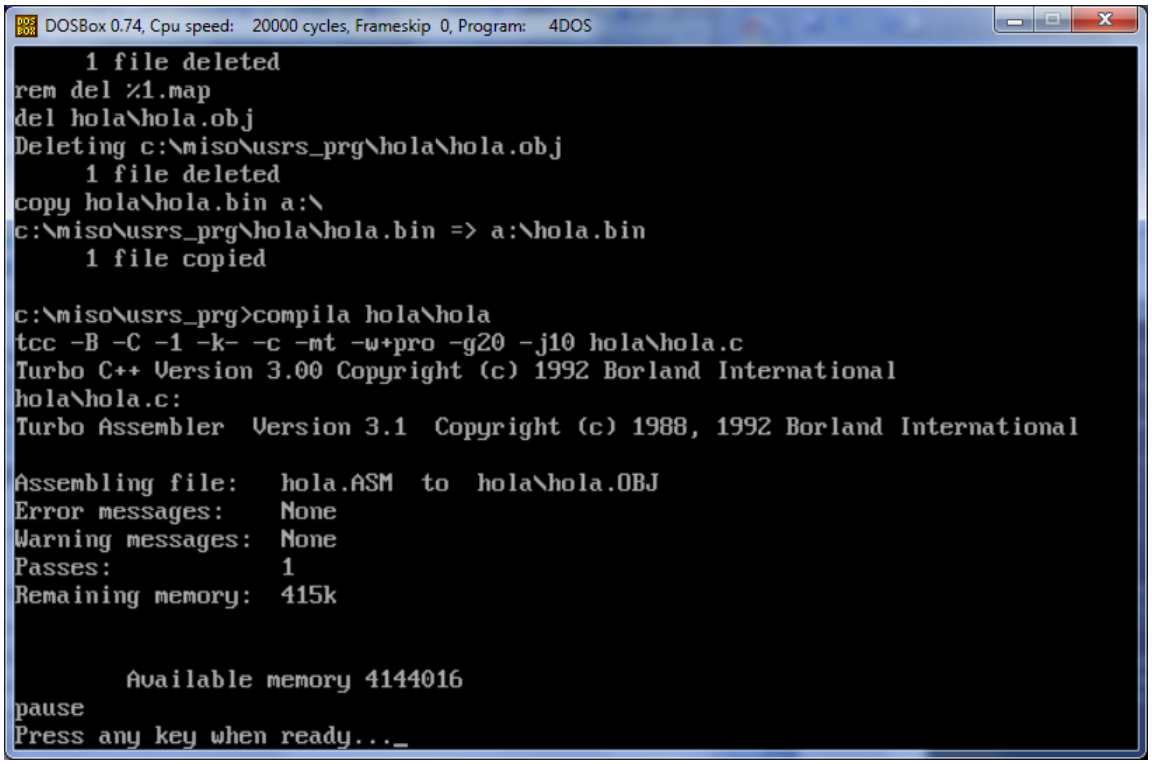
Un pequeño detalle en cuanto a la función `main` es que debe declararse como una función sin parámetros que devuelve un valor de tipo `void`. Después de escribirse la cadena de caracteres `"SO: Hola mundo."` se espera a que se pulse una tecla, con el fin de poder ver tranquilamente lo que se ha escrito. Finalmente el proceso sale de `main` y finaliza.

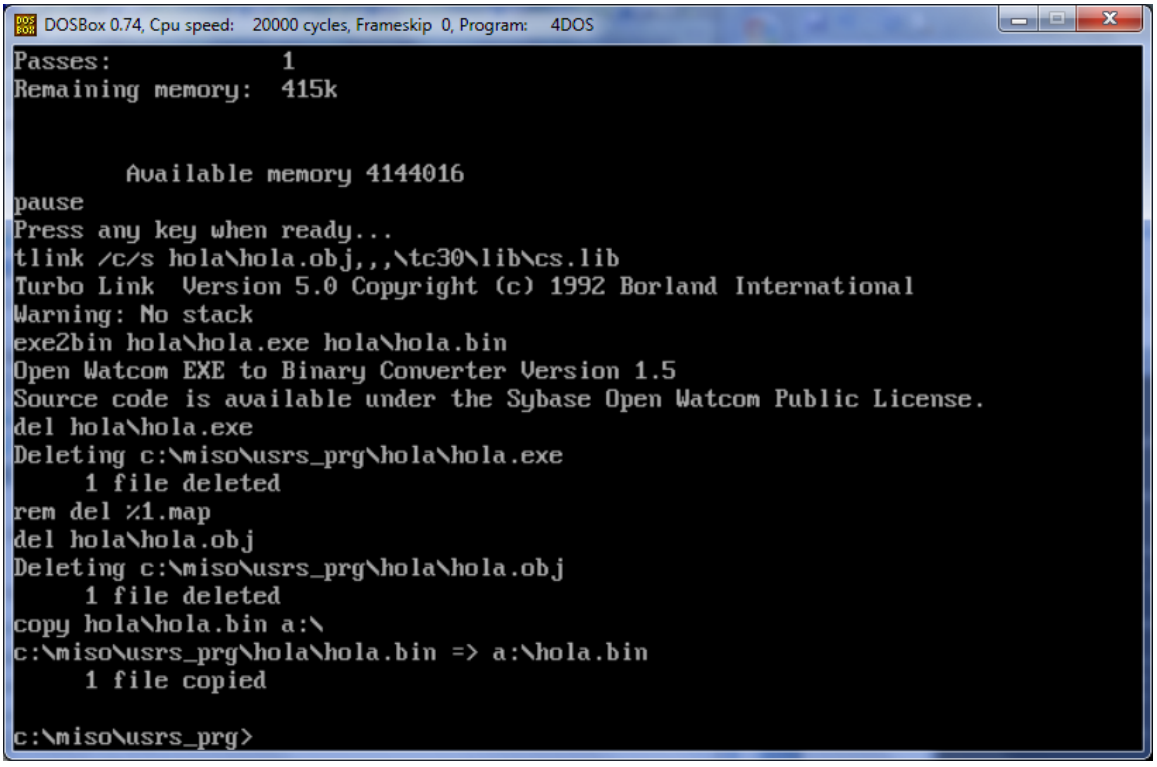
Para compilar y generar el ejecutable del programa "*hola.c*" la forma más sencilla es situarse en el directorio *c:\miso\usrs_prg* y ejecutar el siguiente comando:

```
c:\miso\usrs_prg>compila hola\hola <↵
```

Este comando se ejecuta en dos partes separadas por una pausa que exige la pulsación de una tecla por parte del usuario. La razón de esta pausa es para poder comprobar si hubo errores de compilación, ya que sin ella los mensajes de error desaparecerían por la parte superior de la pantalla.

Tras la pulsación de dicha tecla habremos obtenido las siguientes pantallas:



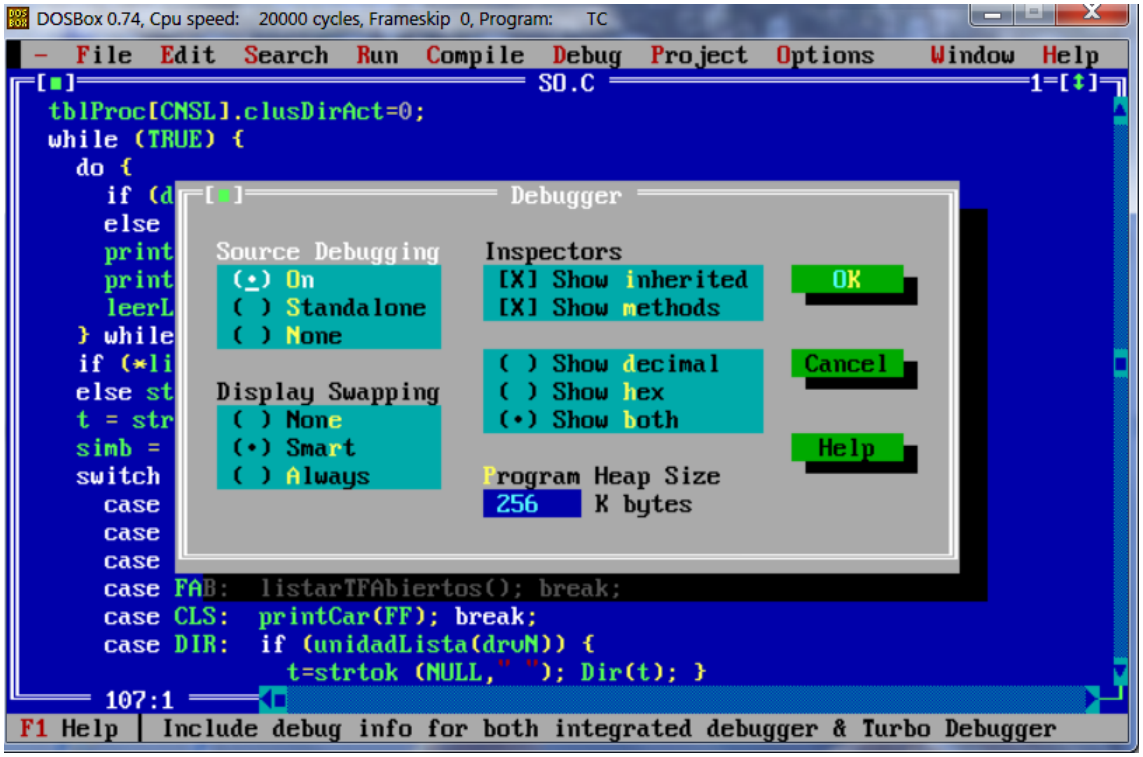


Pantallas con el resultado de la compilación.

El comando compila es un fichero '*.bat*' típico de MSDOS (procesamiento por lotes) que invoca al compilador de línea de comando '*tcc*' y al montador '*tlink*', con una serie de parámetros ya establecidos y probados. Con estos dos programas se genera un fichero ejecutable '*.exe*' apto en principio para MSDOS, pero no para el sistema 'SO', (por tener una cabecera, etc.), por ello el comando también invoca a la utilidad '*exe2bin*', que se encarga de quitar la

cabecera y convertir el fichero a un formato binario '*.bin*'. El resultado final debe ser la creación del fichero '*hola.bin*' que se encontrará en el directorio USRS_PRG\HOLA dentro de MISO. Este fichero también se copia a la unidad de disquete "A:"

Para probar el programa hay que arrancar 'SO' y ejecutarlo desde él. El sistema 'SO' se puede arrancar desde el perfil visto en la actividad anterior '*DBxBoot-SO*', sin embargo es más sencillo seguir usando el perfil que se está usando y aprovechar la característica de que el sistema 'SO' también puede ser arrancado desde el propio DOS e incluso desde el propio entorno de desarrollo de *TurboC*. Arrancar 'SO' desde este perfil es muy sencillo, simplemente nos situamos en el directorio *c:\miso* y tecleamos *so*, ya que el sistema se encuentra también compilado y en forma de ejecutable para MSDOS (*so.exe*). Otra posibilidad consiste en arrancar 'SO' desde el entorno *TurboC*. Desde este entorno podemos abrir el proyecto 'so.prj' y utilizar las teclas de F9/ctrl F9 para compilarlo/ejecutarlo. Usar esta posibilidad nos permite incluso utilizar el *debugger* integrado para depurar el sistema 'SO'. El compilador de *TurboC* nos permite generar el ejecutable, sin código de depuración o con código de depuración incluido para usarse desde el propio IDE o desde el programa externo *TurboDebugger*.



Pantalla de selección de modo depuración integrado.

El programa "*hola.c*" sólo utiliza 3 de las 6 llamadas al sistema que ofrece 'SO'. Vamos a proporcionar un segundo programa algo más completo. Se trata de un programa, "*errante.c*", que implementa un proceso de usuario errante que da un montón de vueltas por la pantalla antes de morir.

El fichero "*errante.c*" está disponible en el mismo directorio `C:\MISO\USRS_PRG\ERRANTE`, donde el alumno puede editarlo, compilarlo y ejecutarlo de la misma manera que hizo con el programa '*hola.c*'. El alumno deberá cargar en memoria cuatro procesos errantes, poniéndolos todos a funcionar simultáneamente presionando en secuencia *"TAB", "6", "TAB", "7", "TAB", "8", "TAB" y "9"*. Después de esto, podremos ver (al igual que antes) a los cuatro pro-

cesos con sus ventanas desplazándose por la pantalla describiendo un movimiento del tipo espiral cuadrada a diferentes velocidades. Se sugiere al alumno que utilice desde la consola el comando "*ps*" para observar cómo se alternan los procesos errantes en el uso de la CPU al más puro estilo *Round-Robin*.

7 ACTIVIDAD 3: Estructura e implementación del sistema 'SO'

Una vez descrita la máquina virtual que nos ofrece el sistema 'SO', vamos a pasar a comentarlo brevemente como programa. 'SO' es un sistema monolítico parcialmente estructurado. Los ficheros fuente del sistema están disponibles en el directorio C:\MISO. El fichero que contiene el programa principal '*so.c*', se ocupa de la inicialización del sistema (creación procesos consola y servidor), y eventualmente de la interpretación de los comandos de la consola.

En este apartado vamos a comentar brevemente cómo está implementada la gestión de memoria y la gestión de los procesos. En las siguientes actividades nos fijaremos en las rutinas de tratamiento de interrupciones y excepciones, así como en la implementación de las llamadas al sistema.

En cuanto a la gestión de la memoria, hay que decir que el método elegido en cuestión ha sido el de *particiones variables*. El sistema utiliza una lista para gestionar los espacios de memoria libres (huecos). Al principio la lista sólo tiene un elemento, el cual hace referencia a toda la memoria disponible (que el sistema previamente ha determinado). El sistema cuando por ejemplo crea un proceso, necesita ubicarlo en memoria y para ello utiliza unas funciones de gestión de memoria que se encargan de reservar y/o liberar cantidades de memoria arbitrarias (bloques) según se les solicite. También es posible reservar memoria para otros propósitos, como por ejemplo la tabla FAT del sistema de ficheros.

La estructura de datos que permite gestionar la memoria está en el fichero *'memoria.c'* y es la siguiente:

```
/* ------------------------------------------------------------
   Tipos de datos para crear una lista de huecos en la memoria
   disponible para programas. Los "links" se guardan en los
   propios huecos. Especifican el tamaño del hueco y un puntero
   al siguiente. Este puntero es del tipo "_seg" que es especial
   ya que tiene siempre un offset implicito 0 (no ocupa). Se hace
   una unión con un tipo word para operar mejor con el */

typedef struct foo _seg *pHue_t; // ptr. offset siempre 0 (2 bytes)

typedef union {
  pHue_t p;    /* se puede usar como puntero */
  word_t w;    /* sin offset (siempre 0) o como un word */
} uHue_t;
struct foo {  word_t tam;  uHue_t uSig; };
word_t iniHeap, finHeap; // comienzo y fin memoria disponible
PRIVATE uHue_t priHue;      // primer elemento de la lista huecos
PRIVATE word_t memDisponible;   // en paragraphs
```

En el fichero *'memoria.h'* tenemos los prototipos de las funciones de gestión de memoria "`tomaMem()`" y "`sueltaMem()`" ya comentadas, además de otras como "`memBIOS()`" para determinar la memoria disponible que reporta la BIOS, "`inicializarMemoria()`", para inicializar las estructuras de datos empleadas, "`inc`()" para incrementar en una cantidad un puntero largo, y por último "`volcar()` y `mostrarMemoria()`" para mostrar en pantalla el contenido de la memoria. También se declara mediante #define CHUNK 768, la cantidad de *paragraphs* que se le asigna a todo proceso para datos + pila + buffer de teclas + video de ventana. Esta cantidad fija se suma a la que requiera el código del programa según su tamaño.

En cuanto a la gestión de los procesos la implementación es estándar tal y como se explica en la teoría, poniéndose en práctica el algoritmo de planificación *Round-Robin*. La parte de código correspondiente a las operaciones básicas con procesos está en el fichero `'procesos.c'`. Las definiciones básicas están en *"procesos.h"* y son:

```
#ifndef PROCESOS_H
/* ------------------------------------------------------------ */
/*                          procesos.h                          */
/* ------------------------------------------------------------ */

#include "windows.h"
#include "ficheros.h"

#define IDLE -1  /* num. proceso idle */
#define CNSL 1   /* num. proceso consola de 'SO'*/
#define SERV 0   /* num. proceso servidor */
#define PWIN_SO tblProc[CNSL].pWin  /* ptro. a ventana de la consola */
#define SRV_PILA 0x9000    /* origen de la pila para el servidor */
#define TICS_POR_RODAJA 2  /* num. tics por rodaja round robin */

typedef int pid_t;              // identificador de proceso

typedef enum {
  LIBRE, EJECUCION, PREPARADO, BLOQUEADO   /* estados de un proceso */
} estado_t;

typedef enum { // razon por la que un proceso esta bloqueado
  BLK_NO, BLK_TECLADO, BLK_SERV, BLK_PAUSE, BLK_OTROS
} esperaPor_t;

typedef cola_t far * fpCola;

#define MAX_TDF 10
typedef struct {
  estado_t     estado;   // LIBRE, EJECUCION, PREPARADO, BLOQUEADO...
  esperaPor_t  esperaPor;// En caso de bloqueo, razon del mismo
  fpCola       pColaBlk; // En caso de bloqueo, puntero a cola espera
  pid_t        pid,pPid; // Pid del proceso y pid del padre
  nproc_t      sig;      // Siguiente proceso de posible lista
  fpRegsPila_t sp;       // Los registros (status) se guardan en la pila
  word_t       dirMem;
  word_t       tamMem;   // Comienzo y tama¤o del bloque memoria
  pWin_t       pWin;     // Ventana y terminal del proceso
  byte_t       nombre[12];    // nombre fichero ejecutable
  long         tamFichero;    // tama¤o  "        "
  drv_t        drvWork;       // Drive actual (¢ de trabajo)
  word_t       clusWork;      // cluster del dir. de trabajo del proceso
  df_t         tdf[MAX_TDF];  // tabla de descriptores de fichero     @@
```

```
} descrProc_t; // descriptor de proceso

extern descrProc_t tblProc [MAX_PROCS];   /* tabla de procesos */
extern cola_t preparados;
extern cola_t aServir;
extern word_t  numProcesos;     // num. procesos en cada momento
extern nproc_t nprEjec,nprAtnd,nprAtBk;// procesos en ejecucion atendido
(+bak)
extern word_t contTicsRodaja;   // contador de tics de una rodaja

/* --------- funciones ------------*/

pid_t nuevoPid (void);
nproc_t nproc (pid_t pid);
void encolar (fpCola cola, nproc_t npr);
void colar (fpCola cola, nproc_t npr);
nproc_t desencolar (fpCola cola);
void quitarDeCola (fpCola cola, nproc_t npr, nproc_t nprAnt);
nproc_t buscarEnCola (fpCola cola, nproc_t npr, bool *pErr);
void activarProceso (nproc_t pid);
void activarPreparado (void);
void saveStack (void);
void restoreStack (void);
void bloquearProceso (esperaPor_t por, fpCola cola);
nproc_t crearProceso (word_t part, word_t size, char * name);
void inicProcesos (void);
int killProcess (nproc_t npr);
void listarProcesos (void);
void reboot (void);
void finProgDOS (void);
date_t getDate (void);
void pause (void);  // implementada como llamada al sistema

#define PROCESOS_H
#endif
```

El tipo de datos `descrProc_t` corresponde al descriptor de proceso. Los campos que incluye son: el estado de ejecución (*`estado`*), la razón por la que el proceso está bloqueado, si fuera el caso (`esperaPor`), la cola donde se pone el proceso cuando está bloqueado, si fuera el caso (*pColaBlk*), el identificador del proceso (`pid`) y el de su padre (`pPid`), el campo `sig` (para enlazar descriptores y formar listas), un puntero a la pila donde se guardan los registros generales para cambios de contexto (`sp`), la dirección y tamaño de la ubicación en memoria del proceso (`dirMem` *y* `tamMem`), puntero a la ventana (incluye teclado) de trabajo del proceso (*pWin*). El cluster y drive de trabajo del proceso (`drvWork` *y* `clusWork`), la tabla de descriptores de ficheros abiertos por el proceso (`tdf[]`), y por último a título informativo, el nombre del fichero ejecutable.

El tipo `cola_t` se utiliza para representar colas de procesos y en concreto la cola de preparados. Las declaraciones `extern` se encuentran declaradas nuevamente en el fichero "*procesos.c*". La directiva "`extern`" se utiliza para informa al compilador de que no reserve en ese momento el espacio de la variable, únicamente tome nota de su tipo.

En el sistema 'SO' cada proceso tiene su ventana, que es algo así como su pantalla virtual (aunque es posible que varios procesos usen la misma ventana). Por otro lado, como sólo hay un teclado físico es necesario repartirlo entre todos los procesos recurriendo a la multiplexación en el tiempo. Con ese fin se implementa el concepto proceso focal, de manera que los

caracteres procedentes del teclado se asignan en cada momento al proceso focal, que es el que se considera que tiene su ventana con el foco del teclado. Como ya vimos, es posible cambiar la ventana focal con la tecla TAB.

Las funciones internas más relevantes de gestión de procesos implementadas son:

```
void activarProceso (nproc_t npr);
void activarPreparado (void);
void bloquearProceso (esperaPor_t por, fpCola cola);
nproc_t crearProceso (word_t part, word_t size, char *name);
int killProcess (nproc_t npr);
```

La función `activarProceso` reanuda la ejecución del proceso cuyo número se le pasa como parámetro, restaurando su estado (contexto) al que tenía cuando fue interrumpido o cedió el control de la cpu al S.O. En consecuencia, al llamar a esa función si no hay error, no habrá retorno.

La función `activarPreparado` reanuda la ejecución del primer proceso de la cola de preparados, invocando a la función `activarProceso`. Si la cola de preparados estuviese vacía, y puesto que no existe ningún *proceso ocioso*, el sistema activa (permite) las interrupciones (instrucción `STI`) y detiene el procesador (instrucción halt: `HTL`).

La función `bloquearProceso` pone al proceso que se le pasa como parámetro en estado de "bloqueado", especificando la razón de bloqueo que también se le pasa como parámetro. Si el puntero a cola que se le pasa no es NIL pondría en dicha cola al proceso bloqueado. Esta función presupone que el estado (contexto) del proceso se encuentra ya salvado en la pila, de la cual el descriptor del proceso guarda su dirección.

La función `crearProceso` se invoca una vez que se ha conseguido memoria para el proceso y se ha cargado el código del mismo en ella. Básicamente esta función inicializa los campos del descriptor del proceso y pone éste en la cola de preparados, para cuando el planificador lo seleccione, éste pueda empezar su ejecución.

Finalmente, `killProcess` mata el proceso que se le pasa como parámetro. Si fuera el que está en ejecución, al final, pondría en ejecución el primero de la cola de preparados sin retornar al punto donde se hizo esta llamada. Al eliminar el proceso realiza las siguientes acciones: Libera su descriptor; devuelve la memoria al sistema, cierra su ventana terminal, si fuera el último proceso en usar dicha ventana; cierra sus ficheros abiertos; y por último, quita el proceso de la posible cola donde pudiera estar bloqueado (si fuera el caso).

8 ACTIVIDAD 4: Implementación de la interrupción Ctrl-C

En el PC la rutina de interrupción del teclado (`INT 09H` *Keyboard*) correspondiente a la BIOS, se encarga, entre otras cosas, de detectar por software si se está presionando simultáneamente las dos teclas Ctrl y Break, en cuyo caso simula una nueva interrupción que utiliza como vector de interrupción, el **1B**h (`INT 1BH` *Keyboard Break*). Este hecho está documentado en el *techelp*, no obstante, en el caso de la máquina virtual *DosBox*, esto no sucede, es decir, la combinación Ctrl-Break no produce dicha interrupción '**1B**', por esta razón y para con-

seguir un resultado similar, en el sistema 'SO' se ha reemplazado esta combinación por la combinación Ctrl-C, la cual se detecta en el módulo *"teclaint.c"* y se encarga de generar dicha interrupción, por lo que a todos los efectos el resultado será similar.

El objetivo de esta actividad es que el alumno programe la rutina de tratamiento de interrupción del vector '**1B**', para que en ella se "maten" inmediatamente todos los procesos vivos cuya ventana terminal sea la focal en el momento de la activación de la rutina (o sea, al pulsar Ctrl-C).

El programa "*techelp*" nos indica que la dirección del vector que utiliza la interrupción correspondiente al Ctrl-Break es la dirección de memoria 0000:006C (que sale de multiplicar el número del vector '**1B**' por 4, ya que cada vector de interrupción ocupa 4 bytes).

Vamos a centrarnos ahora en qué es lo que debe hacer la rutina de tratamiento de la interrupción '**1B**' para matar todos los procesos de la ventana focal. La respuesta no es complicada, ya que acabamos de ver en la actividad anterior que el sistema operativo 'SO' cuenta ya con la operación que necesitamos (ver fichero: "procesos.c") que es:

```
int killProcess (nproc_t npr);
```

La ventana focal se guarda en todo momento en la variable "`pWinFocal`" declarada en el fichero "*windows.h*".

```
pWin_t pWinFocal;
```

En la tabla de procesos "`tblProc[]`" declarada en "*procesos.h*", tenemos los descriptores de todos los procesos. El campo "`pWin`" del descriptor nos dice cual es la ventana terminal del proceso, y con ello podemos saber si un proceso es focal o no, por ejemplo:

```
If (tblProc[npr].pWin==pWinFocal) …
```

Nos diría si el proceso "`npr`" tiene como ventana la focal, es decir si es focal o no. Por otro lado, el campo "`estado`" del descriptor nos indica el estado del proceso: LIBRE, PREPARADO, etc. Si el estado es LIBRE, sabemos que no es un proceso vivo, y por tanto, al recorrer la tabla de procesos, podríamos descartar estos descriptores, para evitar "matar" un proceso "no vivo". Sin embargo, aunque se intentara matar un proceso "no vivo" (estado == LIBRE), no pasaría nada grave, ya que la función "`killProcess`()" ignora estas peticiones.

Resumiendo: En la rutina de tratamiento de interrupción 'Ctrl-C' hay que recorrer la tabla de procesos, seleccionando los vivos y focales para matarlos, aunque hay que añadir un detalle importante: Si se llega a matar el proceso en ejecución antes de finalizar el recorrido de la tabla, y dado que la función "`killProcess()`" no retorna, podríamos no haber conseguido el objetivo, ya que podría haber más procesos focales en la tabla de procesos.

Pasemos ahora a otra cuestión. ¿Cómo incorporar la modificación/mejora dentro del código del sistema 'SO'? La respuesta la podemos encontrar fijándonos en la estructura de alguno de los ficheros de 'SO' donde se tratan las rutinas de tratamiento de interrupción, por ejemplo "`teclaint.c`", del que vamos a indicar a continuación su organización. Antes de eso conviene aclarar que aunque en alguna parte del código de 'SO' que veamos haya instrucciones

en ensamblador, no se va a exigir al alumno que programe en ensamblador. Dicho esto a continuación se muestra el código básico empleado en dicha rutina:

```
/* ------------------------------------------------------------------- */
/*           Rutina de tratamiento de la interrupción xxx              */
/* ------------------------------------------------------------------- */

#include “tipos.h”
#include “rticomun.h”

/* Numero de vector utilizado para la interrupción xxx */
#define V_INT_xxx  0x??  // Sustituir las ?? Por su valor

PRIVATE void far * Old_VI_xxx; // para salvar antiguo vector de cBreak

/* ------------------------------------------------------------------
   las rutinas de tratamiento de interrupción (RTI) salvan auto-
   máticamente los registros en la pila del proceso interrumpido
   y además restablecen el registro DS al valor original del S.O.,
   con ello los registros quedan preservados, pero como se va a
   establecer una nueva pila dentro del espacio del S.O., tenemos
   que salvar los punteros de pila (SS:SP) en el descriptor del
   proceso interrumpido. Cuando se desee volver al punto de
   interrupción se debe restaurar antes dicha pila y después
   efectuar el fin del tratamiento de la RTI (pop's e IRET)
   Para establecer la nueva pila dentro de ‘SO’ voy a considerar
   una nueva base con un valor inicial BASE_PILA. Se dejan 0xFFFF -
   BASE_PILA KB para la consola cuya pila empieza en la FFFE.
   ------------------------------------------------------------------*/

void interrupt rti_xxx (void) {
  static int i;  /* ejemplo de variable local static permitido */
  setNewStack(); /* se establece una nueva pila dentro de ‘SO’*/
  /* Ctrl-c debe matar todos los procesos de la ventana focal (de teclado),
     excepto la consola y el servidor */
  /* -----  Aquí vendría el código propiamente dicho de la RTI ------ */
     ...
     etc.
     ...
  /* ------------------- fin del código especifico -------------------*/
  restoreStack();  /* se restaura la pila antes de volver */
}// rti_xxx

void redirigirInt_xxx (void) {
  asm cli
  Old_VI_xxx = ((ptrTVI_t) 0) [V_INT_xxx];
  ((ptrTVI_t) 0) [V_INT_xxxx] = (void far *) rti_xxx ;
  asm sti
}

void restablecerInt_xxx (void) {
  asm cli
  ((ptrTVI_t) 0) [V_INT_xxx] = Old_VI_xxx;
  asm sti
}

/* ------------------------------------------------------------------- */
```

Vamos a explicar un poco este código: Comienza con la directivas #include del fichero “*tipos.h*” y “*rticomun.h*”. En el primero se encuentra en tipo “`ptrTVI_t`” que se usa más

abajo y en el segundo, se encuentran los prototipos de las funciones "`setNewStack()`" y "`restoreStack()`" empleadas para establecer la nueva pila dentro de 'SO' y restaurarla. Seguidamente encontramos la directiva `#define V_INT_xxxx`, que se utiliza simplemente por legibilidad y mantenibilidad. Las "xxx" deben ser sustituidas por un identificador más o menos nemónico de la interrupción con la que tratamos. Después se declara una variable del tipo puntero lejano a función, la cual servirá para guardar la dirección del vector de interrupción que hubiera antes de que 'SO' establezca el suyo propio. Esta variable se usa en las funciones "`redirigirInt_xxx()`" y "`restablecerInt_xxx()`" que se encargan, la primera, de salvar y establecer el nuevo vector de interrupción, y la segunda, de restaurar el vector original. Es conveniente aclarar la razón de la existencia de estas funciones. La función "`redirigirInt_xxx`" se requiere para establecer en el vector adecuado la dirección de la rutina de tratamiento que se desea incorporar a 'SO, sin embargo no está tan claro la necesidad de la funcion "`restablecerInt_xxx`", ya que en principio, se puede pensar que cuando el sistema 'SO' acabe, no hay necesidad de restaurar los vectores modificados durante el arranque. Si bien esto es cierto, no hay que olvidar que por varias razones, el sistema 'SO' también puede funcionar como invitado del sistema MS-DOS, y en ese caso sí es necesario restaurar los vectores a su estado original para no dañar el sistema anfitrión.

Antes de seguir es conveniente recordar un poco el mecanismo de las interrupciones del procesador. Cuando se produce una interrupción, el procesador pasa a ejecutar la instrucción que se encuentra en la dirección apuntada por el vector asociado a la interrupción que se ha producido, apilando previamente la dirección de retorno y los "*flags*" (banderas) de estado del procesador, e inhibiendo además las interrupciones. Una vez dentro de la "RTI", lo primero que se hace es salvar el contenido de los registros en la pila. Aunque el código que se encarga de ello no aparece explícitamente en el fuente "C", el compilador lo genera automáticamente cuando se especifica que la función es de tipo "`void interrupt`", como es el caso de la rutina de tratamiento de interrupción vista arriba: "`rti_xxx`()". Estas funciones, además de tener al comienzo una serie de instrucciones de apilamiento de todos los registros del procesador, también restauran el valor del registro de segmento de datos 'DS' al valor que debe tener para direccionar el área de datos del sistema 'SO'. Esto nos facilita el trabajo ya que la función preserva el estado de los registros y establece el valor del DS para poder trabajar con las variables globales, todo ello de manera implícita. A partir de ese momento habría que añadir el código necesario para lo que se requiera, pero normalmente la RTI lo que suele hacer justo a continuación, es establecer una nueva pila dentro del espacio de 'SO', lo que le permitiría **no depender** del espacio disponible de pila del proceso interrumpido, y por simetría, también justo antes de finalizar, tendría que restaurar la pila dejándola como estaba antes de la interrupción. Este comportamiento, que es el habitual para cualquier RTI, **no debe realizarse en este caso**, ya que la RTI de Ctrl-C se invoca desde dentro de otra RTI, la de teclado, la cual a su vez ya ha realizado esta acción, y volver a realizarla corrompería la pila previamente establecida dentro de SO, por tanto, en el código listado arriba, el cual se puede tomar como ejemplo básico de una RTI, hay que eliminar para este caso concreto las llamadas a las funciones setNewStack() y restoreStack(). Por último un detalle importante a tener en cuenta a la hora de implementar una RTI: "No se deben declarar variables de ámbito local porque se ubican en pila. Si se requisiera alguna debe declararse global ó local con el atributo '**static**', lo cual hace que se ubiquen en el segmento de datos".

Una vez programada la rutina de tratamiento (rti_xxx), necesitamos programar también tanto la redirección del vector de interrupción para que apunte a dicha rutina, como su posterior restauración. Para ello usaremos las funciones ya vistas arriba: redirigirInt y restablecerInt, personalizándolas para nuestro caso, quedando únicamente pendiente la cuestión de dónde

invocarlas. La respuesta es sencilla. Dentro del fichero "so.c" se podrá ver justo al principio, de la función "`main()`", cómo se invocan el resto de funciones de redireccionamiento de los vectores de interrupción, y también de igual modo, se puede ver cuando se trata el comando "EXIT" dentro del "*switch*" general, que también se invocan todas las funciones de restauración de los vectores de interrupción. En estos puntos es donde debemos insertar nuestros cambios o mejoras.

Una vez hechos estos cambios, si compiláramos el proyecto 'SO' sin más medidas, aparecerían errores indicando que no se encuentran los prototipos de las funciones *redirigirInt* y *restaurarInt*. El motivo es muy sencillo. Dichos prototipos se han de incluir en un fichero tipo "*.h*" característico de "C" para poder ser usado donde se requiera. Este fichero lógicamente debería llamarse "*break.h*" y debe ser incluido en el fichero fuente "*so.c*". Podemos tomar como ejemplo otros ficheros "*.h*", como por ejemplo "*timerint.h*".

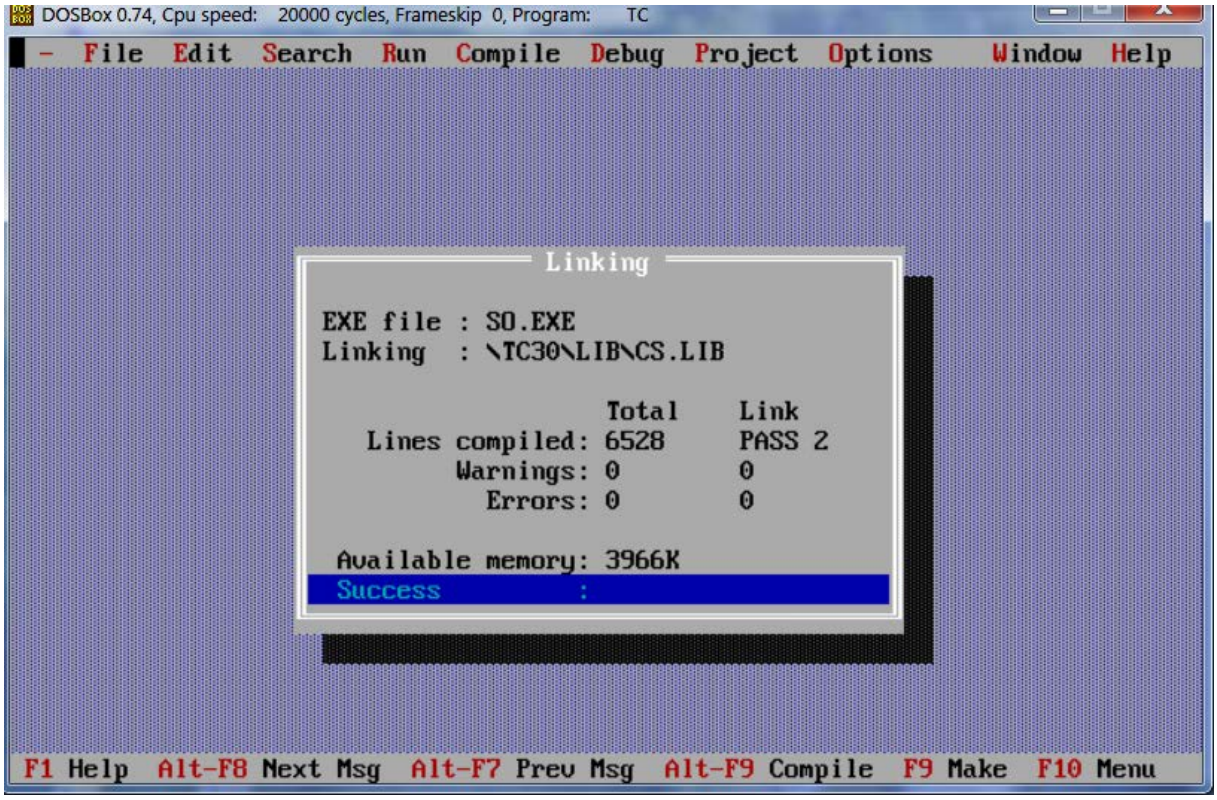
Por último sólo nos falta integrar estos ficheros en el proyecto "*so.prj*". Para ello basta con insertar el fichero *"break.c"* en el fichero del proyecto, abriendo la ventana de proyecto e insertando dicho fichero. Una vez hecho todo esto, ya sería posible compilar nuevamente el proyecto 'SO' mediante la tecla "F9" (ó Ctrl-F9 para compilar y ejecutar).

En la evaluación de este apartado se comprobará que, en el sistema 'SO' modificado por el alumno, la combinación de teclas 'Ctrl-C' mata todos los procesos cuya ventana es focal, tanto si están en ejecución como si están preparados o bloqueados.

Para poder probar que la mejora pedida al alumno funciona hay que conseguir que más de un proceso use una misma ventana. Para este propósito se ha incluido en el comando de ejecución de procesos, un parámetro adicional numérico que indica el 'pid' del proceso de cuya ventana queremos usar como anfitriona, por ejemplo, si tecleamos: ***A:>hola 2***, querríamos ejecutar el programa "hola" pero en la ventana del proceso cuyo 'pid' fuera 2. Al ejecutarse en la ventana de otro proceso, de algún modo, las pulsaciones de tecla se han de repartir entre los procesos que comparten la ventana. En este sistema se ha optado por hacer que *el último proceso que pida una tecla se ponga el primero en la cola de teclas de su ventana terminal,* lo cual no tiene porque ser la mejor política pero puede valer para realizar pruebas.

9 ACTIVIDAD 5: Compilación del sistema 'SO'

Aunque ya se ha visto algo de esto en apartados anteriores vamos a explicar cómo se compila el programa correspondiente al sistema operativo. Utilizaremos el perfil D-Fend "DBOX-SO" desde el cual tenemos acceso a la carpeta "virtualHD" que se monta como unidad de disco duro virtual "C". Los ficheros fuente que componen el programa del sistema están en el directorio '*c:\miso*'. En la fase de desarrollo lo normal es ejecutar el sistema partiendo de MS-DOS y desde el entorno de desarrollo de Turbo-C, todo ello sin necesidad de crear un disquete de arranque en ningún momento. El proceso es muy sencillo y ya hemos visto algo de ello anteriormente. Estando situados en el directorio que contiene los fuentes arrancamos Turbo-C, abrimos el proyecto "*so.prj*" (menú Project->Open Project), si no lo hubiera abierto ya automáticamente el entorno integrado de Turbo-C, y pulsamos "F9":



Una vez compilado podemos ejecutar el sistema 'SO' pulsando la combinación de teclas Ctrl-F9 o bien usar el ratón para abrir el *menú -> Run -> Run*. Esta opción, al ejecutar 'SO' sin salir del entorno TC, no deja mucha memoria disponible y si queremos probar 'SO' con más memoria es preferible salir de TC y ejecutar 'SO' desde la línea de comando:

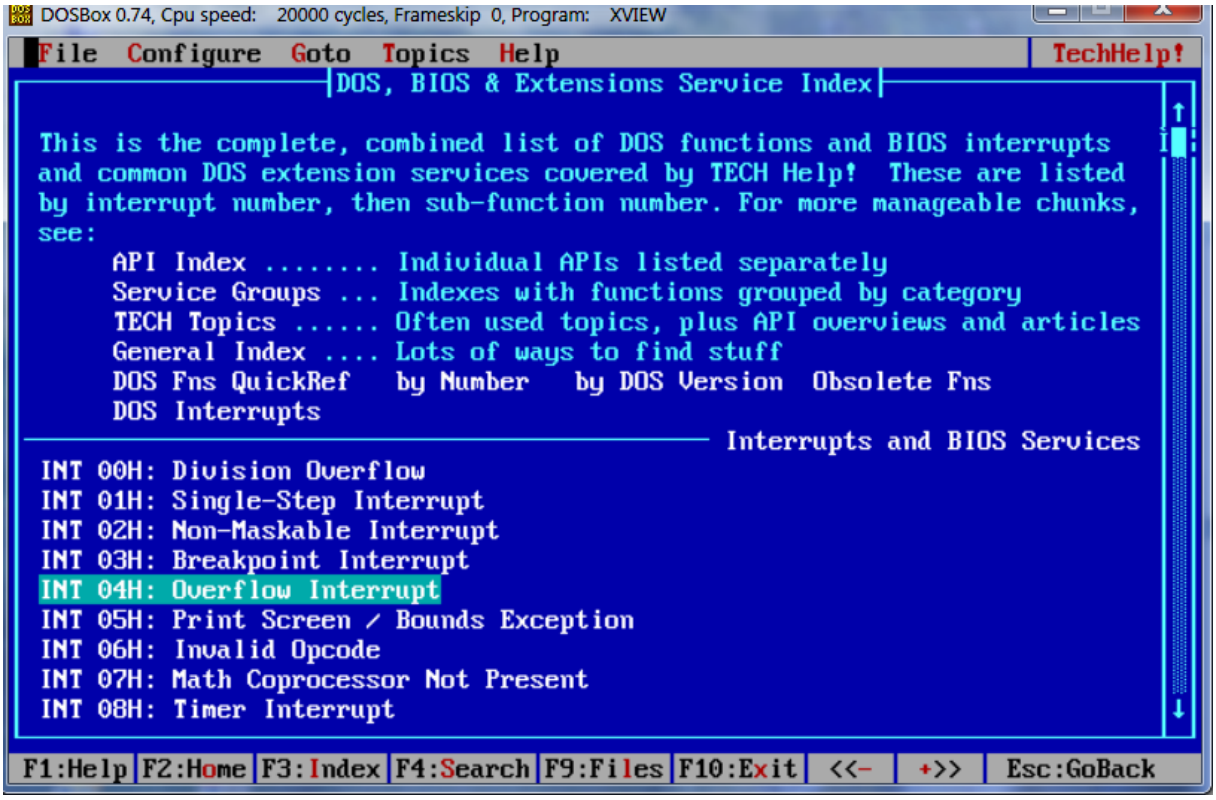
```
C:\MISO>SO <┘
```

(Aunque ejecutemos 'SO' desde MSDOS sólo podremos acceder desde 'SO' a los ficheros que estén en las unidades montadas físicamente en Dos-Box como 0, 1, 2 y/o 3.)

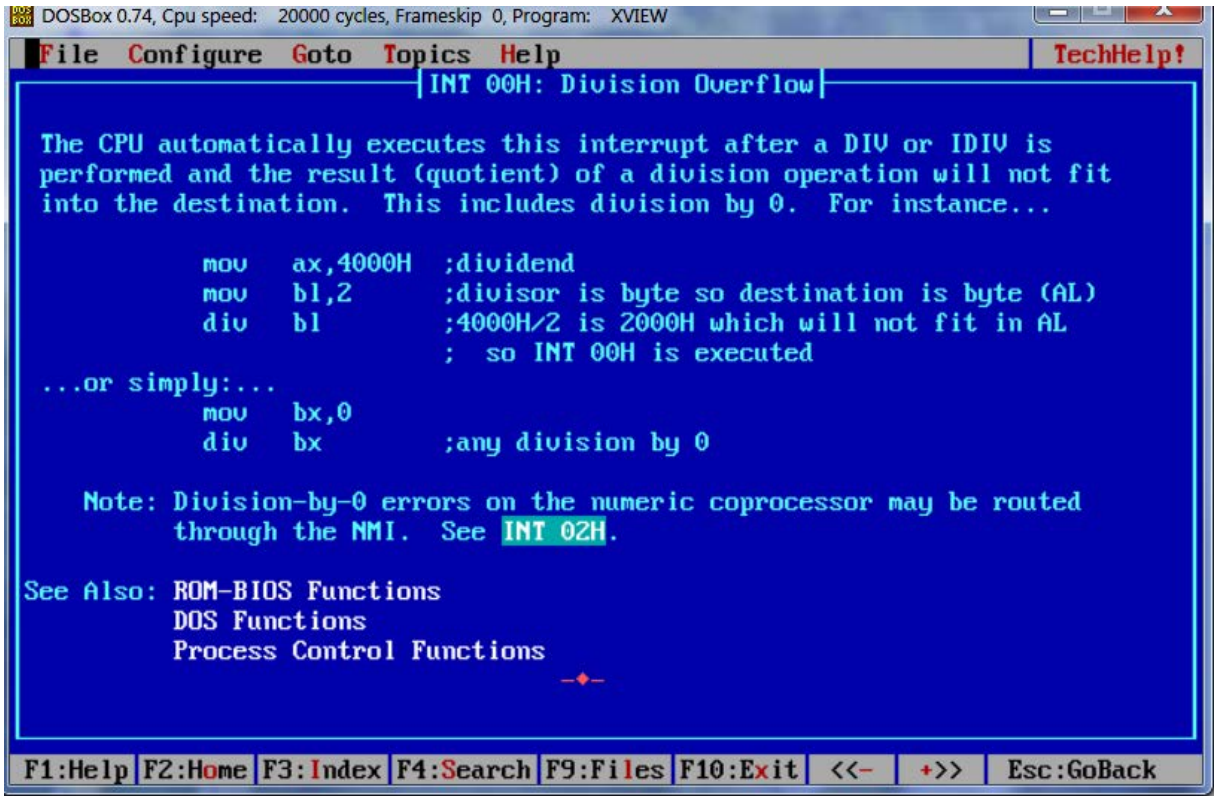
Se recuerda que la velocidad de ejecución del programa puede variar mucho de ejecutarlo dependiendo de la selección que se haya hecho para la CPU *speed*.

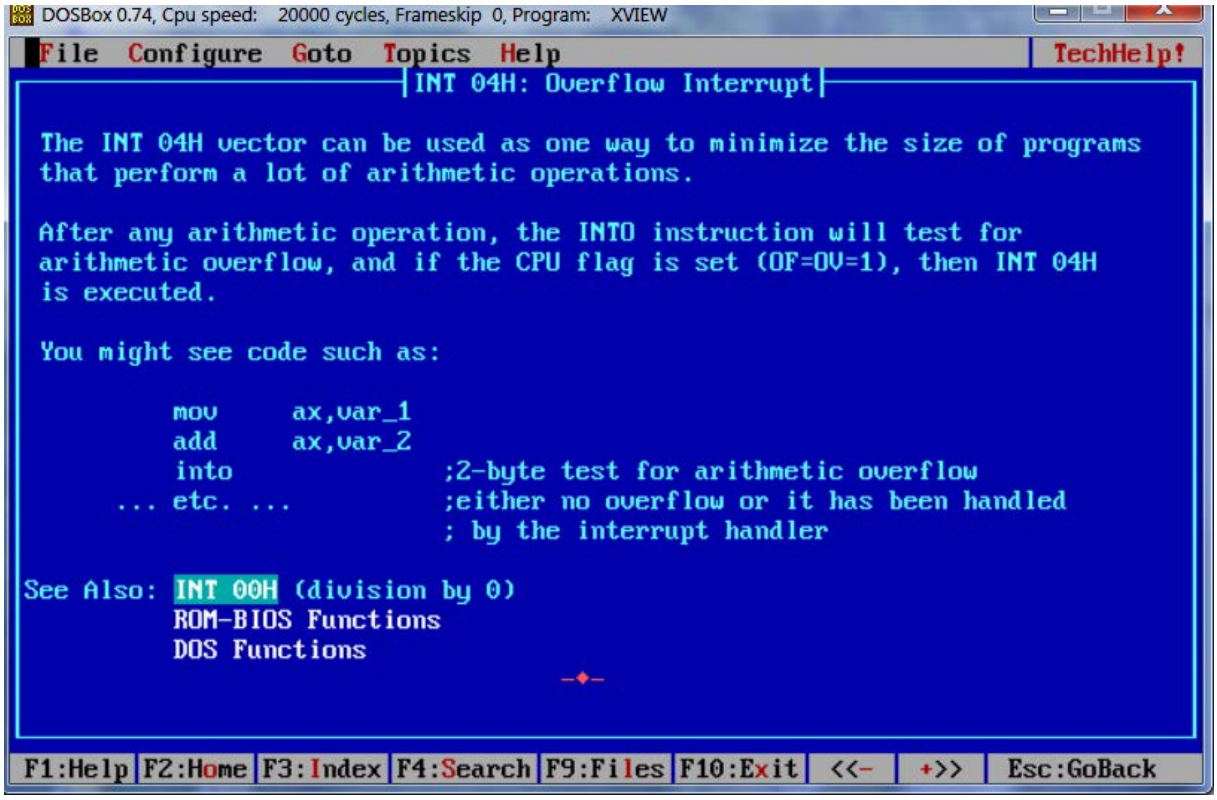
10 ACTIVIDAD 6: Implementación de excepciones (división por 0 y *overflow*)

En este apartado el alumno programará las rutinas de tratamiento de las excepciones de división por 0 y *overflow*. Dado que la programación de esas rutinas es muy parecida a la de las rutinas de tratamiento de interrupción, tenemos ya mucho camino recorrido gracias al apartado anterior. Los números de los vectores de interrupción que utilizan estas excepciones son los indicados en el *techelp*:



Más explícitamente son: el vector `00H` para el caso de la excepción de división por 0, y el vector `04H` para el caso del overflow. Podemos conocer cómo funcionan esas excepciones consultando los enlaces correspondientes:





Según las pantallas anteriores las rutinas de tratamiento de las dos excepciones toman el control automáticamente al ejecutar la instrucción `div` (o `idiv`, división con signo) y la instrucción `into` (interrupción si hay *overflow*) respectivamente bajo ciertas condiciones. En el primer caso la condición que dispara la excepción es que el divisor valga 0, o que el cociente no quepa en el registro destino de ese cociente (`AX` o `AL`). En el segundo caso la excepción se produce al ejecutar `into` estando activado el *flag* de *overflow*.

En cuanto al tratamiento de la excepción, deberá ser el siguiente:

- En la consola de 'SO' debe aparecer un mensaje que avise de que se ha producido una excepción y muestre cuál es su tipo (división por 0 u *overflow*) y cuál es el identificador del proceso (pid) responsable de esa excepción.

- La excepción debe provocar la muerte inmediata del proceso responsable de la excepción excepto si es la consola o el servidor, y

- El siguiente proceso a ejecutar debe ser el que determine la política de planificación existente actualmente en 'SO'.

Para escribir datos en la consola podemos hacer uso de algunas de las siguientes funciones auxiliares internas del sistema 'SO' (declaradas en '*windows.h*'):

```
PrintStr(str) ;                 /* str es una cadena de caracteres */
PrintDec(numero, ancho);        /* numero es un entero sin signo */
PrintHex(numero, ancho);        /* muestra el valor en hexadecimal */
```

En las funciones anteriores que escriben números debe indicarse el número de cifras (`ancho`) con el que va a escribirse el número. Si el número de cifras especificado es insuficiente para el número que se quiere escribir, la función utiliza como `ancho` el número de cifras necesario. Por ese motivo normalmente se indica como `ancho` una única cifra (es decir `ancho` 1). En el caso de escritura hexadecimal, se rellena automáticamente con ceros a la izquierda hasta completar el número de cifras indicadas en `ancho`. Un detalle muy a tener en cuenta es que estas funciones internas muestran el texto en la ventana del proceso que estuviese en ejecución en el momento de la interrupción y si dicho proceso muere, el mensaje no podrá ser visto por la rapidez con la que desaparece la ventana del proceso. En estos casos es mejor que estos mensajes salgan directamente en la pantalla de fondo (no es ninguna ventana). Esto puede hacerse poniendo simplemente la variable global de 'SO' "`videoDirecto`" a TRUE inmediatamente antes de la función `Print…()`, y restaurándola a FALSE justo a continuación. De este modo el sistema 'SO' realiza la salida a dicha pantalla en vez de a la ventana del proceso en ejecución. Si queremos situar el mensaje en alguna posición concreta de la pantalla, utilizaremos las variables globales `curX` (0..79) y `curY` (0..24) antes de dicha salida para situar el cursor. (En general, un buen sitio para los mensajes sería: curX=14 y curY=24)

El esqueleto de las rutinas de excepción es similar al que se uso para Ctrl-C. Es decir habrá que crear los ficheros "*excepcio.h*" y "*excepcio.c*" y añadirlos al proyecto.

Con el fin de comprobar el correcto funcionamiento de las rutinas de tratamiento de excepción de división por cero y *overflow*, proponemos el siguiente programa de usuario: "*menuexc.c*", el cual se encuentra en el directorio *c:\miso\usrs_prg\menuexc*, y que a través de un sencillo menú, nos da la opción de producir cualquiera de las dos excepciones:

```
/* ------------------------------------------------------------------------*/
/*                                menuexc.c                                */
/* ------------------------------------------------------------------------ */
/*       programa de usuario con un menu para provocar excepciones          */
/* ------------------------------------------------------------------------ */

#include "..\ll_s_so\inic-usr.c" //cod.inic.prog.usuario y llam. basicas

void main () {
  char car=0;
  while (car != 'T') {
    printCar(FF);    /* borra el contenido de la ventana */
    printStr("SO: Elija su excepción:\n") ;
    printStr("'T' - Terminar programa\n") ;
    printStr("'D' - División por cero\n") ;
    printStr("'O' - Overflow\n") ;
    car = leerTecla() & 0xDF;   /* convierte a mayúsculas */
    switch (car) {
      case 'D': asm { MOV BX,0; DIV BX }; break;
      case 'O': asm { MOV AX,0x7fff; INC AX; INTO }; break;
      case 'T': break;
      default: printCar(BEL);
    }/* switch */
  }
}
```

11 ACTIVIDAD 7: Implementación de la llamada al sistema '*sleep*'

En los sistemas tipo UNIX existe un servicio o llamada al sistema llamado "sleep" que permite bloquear a un proceso durante un cierto intervalo de tiempo especificado en segundos. En

el sistema 'SO' se ha echado en falta este servicio cuando se programó el programa "errante.c", ya que allí hubo que ajustar la velocidad de los procesos a base de retardos implementados con bucles de espera:

```
void retardo (unsigned n) {   /* espera activa 'n' ticks */
  unsigned long r, far *ptime= (void far *) 0x046C;
  r = *ptime + n;
  while ( *ptime < r ); /* espera activa */
}
```

En un sistema multiprogramado como 'SO' la implementación de los retardos como bucles de espera activa significa un desperdicio de tiempo de CPU que perjudica a todos los procesos. La solución al problema consistirá en conseguir que los procesos esperen de forma pasiva como procesos bloqueados y no como procesos preparados o en ejecución, aprovechándose con ello el procesador.

El objetivo de este tutorial es la implementación de '***sleep***' como una nueva llamada al sistema que permita llevar a cabo retardos en los procesos de forma pasiva. La implementación se basará en las ideas presentadas en la solución del siguiente problema escogido del tema de Gestión de Procesos:

Procesos	02/09/99	Implementar: dormirse (segundos)	9899se3

Lo primero será establecer la interfaz de la llamada al sistema con los programas de usuario (ver los ficheros "*c:\miso\ll_s_so*.c*"). Vamos a convenir que para hacer uso de esta nueva llamada al sistema habrá que añadir en el fichero "*proc-ifz.c*" la siguiente función de biblioteca:

```
/* ----- Interfaz de llamada a sistema sleep ------- */
void sleep (unsigned decimas) {
  asm { MOV AH,9;
        MOV BX,decimas;
        INT VINT_SO }  // VINT_SO : vector llamada al sistema 'SO'
}
```

Lo anterior quiere decir que los programas de usuario podrán solicitar al sistema operativo que los mantenga bloqueados durante un cierto número de décimas de segundo, para lo cual ejecutarán la función anterior `sleep(decimas)`. En cuanto al sistema operativo (el cual toma el control tras la instrucción (*trap*) `INT VINT_SO,` en la función `"rti_SO"` del fichero "*llamadas.c*"), vemos que debe interpretar el código `09h` que le llega en el registro `AH,` como una petición de servicio "`sleep`", debiendo ceder el control a la función encargada de implementar dicho servicio y que vamos a llamar `so_sleep` (por mantener el mismo criterio de denominación de las funciones de servicio). Esta función tomará el valor del parámetro `'decimas'` entregado en el registro `BX,` donde lo dejó la función de interfaz '`sleep`', y se procederá a bloquear al proceso en ejecución y a activar al siguiente preparado.

Para conseguir este primer objetivo el sistema 'SO' nos ofrece en el fichero "*llamadas.c*" la macro ***RGP(N,R)***, la cual nos facilita el acceso al contenido de cualquier registro guardado en la pila de un determinado proceso (recordemos que cuando se realiza una llamada al sistema, la rutina de tratamiento de interrupción (RTI) guarda en la pila todos los registros). Con esta macro podemos acceder a su contenido fácilmente, por ejemplo: `RGP(nprEjec,B.X)` nos daría el valor guardado en pila del registro **BX** del proceso al que se está atendiendo. De este

modo, acceder al parámetro "`decimas`" es sencillo, bastaría poner la macro tal y como se ha mostrado en el ejemplo anterior.

Pero hay que hacerse otra pregunta. ¿Cómo se bloquea el proceso y se activa el siguiente preparado?, afortunadamente la respuesta es sencilla. El sistema 'SO' dispone de la función de uso interno "`bloquearProceso(razon,&cola)`", la cual nos permite en una sola llamada hacer lo siguiente: a) Dejar bloqueado al proceso especificando la causa (si ha sido por espera de teclado, por dormir, etc.), b) Poner en una cola de procesos bloqueados a dicho proceso, y por último, c) Activar el siguiente proceso preparado disponible en la cola de preparados.

Relacionado con la función de arriba, el sistema 'SO' tiene definido un tipo en "*procesos.h*" denominado "`esperaPor_t`", el cual sirve para especificar la razón de bloqueo. Al crear el servicio 'sleep', hay que aumentar la lista de valores de este tipo, añadiendo el valor '`BLK_SLEEP`' para la razón "durmiendo". Hay que preguntarse también, qué cola habría que pasar como parámetro a la función "`bloquearProceso` ()". Es posible no pasar ninguna utilizando NIL, en cuyo caso los procesos dormidos no formaría parte de ninguna cola. No obstante, sería preferible añadir una nueva cola para este propósito, que podríamos declarar en los ficheros "*procesos.h*" y "*procesos.c*", tomando como modelo la forma en la que están declaradas las otras dos colas: "`preparados`" y "`aServir`".

Hasta aquí ya tenemos suficiente información para poder dejar un proceso bloqueado y en una cola de espera, pero tenemos que resolver más problemas. El proceso debe ser despertado transcurrido un lapso de tiempo especificado en el parámetro "`decimas`". Para resolverlo, en primer lugar debemos pensar dónde guardar el tiempo que falta para que un proceso despierte. El sitio más evidente es el descriptor de proceso. Podemos añadir un campo llamado "`lapso`" que mantendría los *ticks* de reloj que faltan para que el proceso despierte. El tipo de datos del descriptor se llama "`descrProc_t`", y está declarado en el fichero "procesos.h". Una vez resuelto esto, se nos plantea la necesidad de convertir las décimas de segundo en *ticks* de reloj, para lo cual necesitamos saber cuánto tiempo es un *tick* de reloj. Vamos o considerar que dicho tiempo es aproximadamente 55 ms, y que al hacer la conversión realizaremos un redondeo, tal que si la fracción de *tick* es menor de 0,5 la despreciamos y si es superior incrementamos en uno el resultado.

Una vez resuelta implementación de la función de servicio "`sleep()`", para completar el servicio, tenemos que acometer el problema de despertar al proceso en el tiempo establecido. Parece evidente lo más sencillo es modificar la rutina de tratamiento de interrupción del reloj, para que lleve la cuenta de los *ticks* que le quedan a un proceso para ser despertado. Esto no es complicado, bastaría con decrementar el campo lapso del proceso y al llegar a 0, despertar al proceso. ¿Cómo despertamos a un proceso dormido?; respuesta: lo quitamos de la cola de dormidos y lo ponemos en la cola de preparados. En el fichero "*procesos.h*" se encuentran disponibles las siguientes funciones relacionadas con colas:

```
void encolar (fpCola cola, nproc_t npr);
void colar (fpCola cola, nproc_t npr);
nproc_t desencolar (fpCola cola);
void quitarDeCola (fpCola cola, nproc_t npr, nproc_t nprAnt);
nproc_t buscarEnCola (fpCola cola, nproc_t npr, bool *pErr);
```

La función `encolar()`, añade un proceso a una cola por el final y la función `colar()` por el principio. La función `desencolar()` quita y devuelve el primer proceso de la cola. La fun-

ción `quitarDeCola()` quita el proceso especificado en `npr` de cualquier posición en la que se encuentre en la cola. Es preciso pasarle también en `nprAnt` cual es el proceso anterior. Se usa normalmente durante el recorrido de una cola. Por último, la función `buscarEnCola()`, busca el proceso `npr` en la cola y devuelve el proceso anterior en la misma. Si no estuviera en la cola devuelve `FALSE` en `*pErr`.

Aunque se han expuesto todas estas funciones de apoyo a la gestión de colas, normalmente no se requiere usar todas ellas para la modificación de la rutina de tratamiento de interrupción de reloj requerida por el servicio "*sleep*". Finalmente quisiera resaltar un error que se comente a menudo y que consiste en que al eliminar un elemento de una cola durante el recorrido de la misma se intente continuar el bucle del recorrido utilizando el campo siguiente del proceso recién eliminando (mediante quitarDeCola), lo cual sería un error porque al quitarlo de la cola, dicho campo se pone a NIL_PROC. (En otras ocasiones, si se elimina el proceso manualmente pero se inserta en la cola de preparados, el campo 'sig' no sería NIL_PROC pero tampoco apuntaría al siguiente de dormidos).

La incorporación de la nueva llamada al sistema *sleep* debe ser consistente con las llamadas al sistema anteriormente existentes. Por ejemplo, podría ser necesario modificar la rutina de tratamiento de la interrupción del Ctrl-C para que funcione consistentemente con los procesos bloqueados a causa de `sleep`, ya que la destrucción de un proceso dormido debe hacerse eliminándolo previamente de la cola de procesos `dormidos`. También habrá que comprobar la función `killProcess`.

La información dada hasta ahora debe ser suficiente como punto de partida para el alumno. Se recomienda al alumno que tome como modelo la implementación de las demás llamadas al sistema que figuran en el fichero "*llamadas.c*". Ya que el alumno va a tener que modificar varias partes del código de 'SO', deberá identificar los fragmentos de código que añada encerrándolos entre comentarios del tipo:

```
/* inicio modificacion sleep */
...                                (Instrucciones añadidas por el alumno)
/* fin modificacion sleep */
```

Resulta muy útil el uso de '`grep`' para localizar las líneas de los ficheros del código de 'SO' que contienen una determinada palabra. Esta opción se encuentra integrada en el menú de TurboC en: menú –, GREP.

Para comprobar la corrección de la implementación de *sleep* se propone utilizar el siguiente programa de usuario (disponible en *c:\miso\usrs_prg\calibra*):

```
/* ------------------------------------------------------------------------*/
/*                                calibra.c                                */
/* ------------------------------------------------------------------------ */
/*     programa de usuario para calibrar la llamada al sistema sleep        */
/* ------------------------------------------------------------------------ */

#include "..\ll_s_so\inic-usr.c" //cod.inic.prog.usuario y llam. basicas
#include "..\ll_s_so\proc-ifz.c" //llamadas al sistema de procesos

int leerDec (void) {      /* leee un entero sin signo */
  unsigned int acum = 0;
  char car = '0';
```

```
  /*-------------------*/
  while ((car >='0') && (car<= '9')) {
    if (acum>(0xFFFF-(car-'0'))/10) return -1; /* num. demasiado grande */
    acum = 10 * acum + (car - '0');
    car = leerTecla();
    printCar(car);
  }//while
  if (car != CR) return -1;                    /* cifra erronea */
  return acum;
}//leerDec

void main (void) {
  int decimas=-1;
  while (decimas) {
    printCar (FF);
    printStr (" Calibrador del sleep.\n\n Decimas a esperar: ");
    if ((decimas=leerDec()) != 0) {
      if (decimas==-1) {
        printStr ("\n\007 ! Número incorrecto !\n"); }
      else { /* decimas > 0 */
        printStr ("\n Dormido zz...!");
        duerme (decimas);
        printStr ("\n\007 Ya estoy despierto :-)");
      }//else-if
      printStr ("\n Pulsa una tecla."); leerTecla();
    }//if
  }//while
}//main
```

También podrá utilizarse el programa "*erraslp.c*" (donde se han sustituido los retardos mediante bucles de espera por llamadas a `sleep`) para comprobar cómo cambia el funcionamiento respecto de "*errante.c*", el cual utiliza espera activa. (Si se teclea el comando "type leeme.txt" mientras se mueven, por ejemplo, cuatro errantes se observará la diferencia)

12 ACTIVIDAD 8: Implementación de los semáforos

El objetivo de esta actividad es dotar al sistema 'SO' de los objetos abstractos "semáforos". El sistema va a proporcionar a los procesos un cierto número de semáforos con el fin de que los procesos puedan sincronizarse. Las funciones a implementar son: Inicializar, bajar y subir semáforo. Limitaremos la cantidad de semáforos a un máximo de 10. Las funciones de interfaz, al igual que se hizo anteriormente con "*sleep*", se incluirán en el fichero "*proc-ifz.c*".
El procedimiento es similar y basta repasar la actividad anterior para repetirlo sin problemas, aunque hay que precisar cómo serán los prototipos de estas funciones y los valores de los registros a emplear en cada una de ellas. A continuación se muestra dicha información:

```
/* En todas estas funciones el parámetro 'sem' se pasa en el registro BX,
   y el código de operación en AH. En iniSemaforo CX pasa el valor */
int iniSemaforo(unsigned sem, unsigned valor); //AH=0x0B, BX=sem, CX=valor
int bajaSemaforo(unsigned sem);                //AH=0x0C, BX=sem
int subeSemaforo(unsigned sem);                //AH=0x0D, BX=sem
```

Para los detalles sobre la realización de estas funciones de interfaz, úsese como modelo la función de interfaz de '*sleep()*' vista con anterioridad.
Conviene aclarar que el uso de estos registros es arbitrario, pero hay que respetarlo si deseamos utilizar los programas de usuario ya compilados que se entregan con la práctica, ya que estos programas utilizan de este modo los registros. Evidentemente si los recompilamos utilizando otra convención de registros no habría ningún problema.

A continuación vamos a comentar algunos aspectos de la implementación dentro 'SO'. En primer lugar hay que decir que el funcionamiento de los semáforos debe ser coherente con lo visto en la parte de "Teoría" correspondiente al apartado 2.3 de comunicación entre procesos. Por otro lado, y ya referente a los cambios a realizar en el código de 'SO', ya sabemos que por lo general, cada nueva llamada al sistema debe tener su propia función de servicio, la cual ha de añadirse a las ya existentes en el fichero `"llamadas.c"`, del mismo modo en el que se hizo con la llamada "*sleep*" y que por seguir el criterio de nombres ya existente, deberían llamarse: `so_iniSemaforo`, `so_bajaSemaforo` y `so_subeSemaforo`. Recuerdese además el uso de la macro ***RGP(N,R)*** vista en '*sleep'*, para acceder a los registros pasados en la pila cuando se efectuó la llamada.

Parece natural el declarar una tabla en 'SO' para mantener el estado o situación de los semáforos. Lógicamente el número de elementos de la tabla será el máximo número de semáforos soportados; en nuestro caso 10. El número de semáforo que se pasa como parámetro en estas llamadas al sistema será precisamente el índice de esta tabla, por tanto un número de semáforo podrá valer entre 0 y 9. ¿Cómo deben ser los elementos de esta tabla? Como respuesta podemos decir que, en principio, sería suficiente con que tengan un campo numérico para guardar el valor del semáforo y otro del tipo "cola" (`cola_t`) para registrar los procesos bloqueados en el semáforo.

No hay que olvidar inicializar la tabla de semáforos. Bien en algún punto del arranque del sistema 'SO' ó bien en la declaración estática de la misma.

Como especificación del funcionamiento de los semáforos se nos pide lo siguiente: Cuando la operación '`subeSemaforo`' deba desbloquear un proceso y haya varios procesos en la cola del semáforo, se desbloqueará al primer proceso de la cola, es decir al que lleva más tiempo metido en ella (política FIFO). Se recuerda que todas las colas del sistema 'SO' enlazan los descriptores haciendo uso del campo `sig` del descriptor de proceso (que está definido en "*procesos.h*" y que ya existen en 'SO' funciones que facilitan el trabajo con colas).

Debido a que se explica en la parte de teoría cómo debe ser la implementación de los semáforos, y a que ya se han descrito las operaciones básicas de manejo de procesos dentro del sistema 'SO', es el momento de que el alumno ponga en práctica todo lo que sabe.

Para probar la corrección de la implementación de los semáforos se proporciona en el directorio "c*:\miso\usrs_prg\menusem*" el programa de usuario "*menusem.c*" que muestra un menú para realizar cada una de las operaciones propias de los semáforos. Con él podemos crear varios procesos que ejecuten el programa, y conmutar entre ellos con la tecla TAB para que unos procesos se bloqueen con la operación "***bajaSemaforo***", mientras que otros procesos se encarguen de desbloquearlos mediante operaciones "***subeSemaforo***".

Otro programa de usuario que nos permite probar el funcionamiento de los semáforos es el programa "c*:\miso\usrs_prg\cross\cross.c*". Modificación de "*errante.c*", en el que se cumple la siguiente condición:

> ***La cuarta y octava vueltas al circuito corresponden a un tramo de circuito muy estrecho y sólo se permite realizar la vuelta a un único proceso a la vez.***

PRÁCTICA 4

Implementación del Paso de Mensajes

ÍNDICE

1 OBJETIVOS

Los objetivos de esta práctica son:

- Que el alumno profundice sobre el mecanismo de las llamadas al sistema con parámetros que representan direcciones de memoria.
- Que el alumno sea capaz de implementar en un sistema operativo, un tipo de paso de mensajes sencillo a través de buzones (*enviaMsjBuzon*, *recibeMsjBuzon*), con una capacidad fija que viene establecida por una constante interna del sistema operativo.

2 INTRODUCCIÓN

En esta práctica se pretende profundizar en la implementación de uno de los tipos de llamadas al sistema más complejos, como son las correspondientes al paso de mensajes. El tratar este tipo de llamadas al sistema tiene la ventaja de permitirnos ilustrar la comunicación de datos desde el espacio de direcciones de un proceso de usuario hasta el espacio de direcciones de otro usuario distinto pasando por el espacio de direcciones propio del sistema operativo común a ambos, algo que resulta muy interesante para un primer curso de sistemas operativos.

El modelo de comunicación que se va a implementarse será el de la comunicación asíncrona, indirecta, y simétrica, con buzones de capacidad limitada y mensajes de tamaño fijo de 16 bytes transmitidos mediante copia (ver paso de mensajes en el apartado 2.3 del tema de procesos).

3 TRABAJO A REALIZAR

Vamos a presuponer que el alumno conoce ya el funcionamiento del sistema 'SO', tanto a nivel de usuario, introduciendo comandos a través de la consola o haciendo llamadas al sistema desde un programa, como a nivel de la estructura del programa 'SO' en correspondencia con la de un sistema monolítico. En relación con dicha estructura, se supone el conocimiento de la implementación concreta de los procesos (descriptores, tabla de procesos, cola de preparados, el planificador y el despachador) y de la gestión de memoria.

En pocas palabras el trabajo a realizar consiste en la implementación de los manejadores de las llamadas al sistema cuyas rutinas de interfaz vamos a denominar *enviaMsjBuzon* y *recibeMsjBuzon*, las cuales permitirán a los procesos comunicarse a través de un esquema de paso de mensajes basado en buzones (comunicación asínicrona, indirecta y simétrica). Los buzones son de capacidad limitada, debiendo admitirse también el caso particular de que sean de capacidad nula (comunicación síncrona, *rendez-vous*).

4 ESPECIFICACIÓN DEL PASO DE MENSAJES

Nuestro objetivo es añadir un mecanismo de paso de mensajes al sistema 'SO', ampliando su conjunto de llamadas al sistema con las siguientes llamadas, de las cuales indicamos los prototipos de las correspondientes funciones de interfaz:

```
/* ------------------------------------------------------------------- */
/*                              buzones.h                              */
/* ------------------------------------------------------------------- */
/* Especificación de las funciones de interfaz  para el paso de mensajes */
/* ------------------------------------------------------------------- */

#define CAPACIDAD 2  // Capacidad de los buzones, puede ser 0, 1, 2, ...
#define MBOX_MAX 10  // máximo 10 buzones

typedef unsigned char mboxDesc_t;  // descriptor de buzón

int enviaMsjBuzon(mboxDesc_t mbox, void far *msj);  // cod. op. AH = 0Ch
int recibeMsjBuzon(mboxDesc_t mbox,void far *msj);  // cód. op. AH = 0Dh
```

La idea es que el sistema operativo va a ofrecer a los programas de usuario hasta 10 buzones identificados cada uno de ellos por un número del 0 al 9 que denominaremos '*descriptor del buzón*'. Los mensajes se envían/reciben a/de los buzones por lo que la comunicación es indirecta. La comunicación es asíncrona porque el uso del buzón permite que el proceso receptor no tenga que estar listo para recibir, ya que la entrega se deposita en el buzón. Sólo en el caso de que el buzón sea de capacidad nula, el mensaje se entrega directamente al proceso, en este caso la comunicación es síncrona, ya que se requiere que el proceso esté listo para recibir. Un mensaje queda identificado por su dirección de comienzo (que podemos ver como un puntero lejano), entendiéndose que el mensaje está físicamente compuesto por el valor de los 16 bytes consecutivos que se encuentran a partir de dicha dirección de memoria. Por tanto los mensajes son de tamaño fijo igual a 16 bytes. A continuación se muestran varios ejemplos del envío de diferentes tipos de mensajes:

```
mboxDesc_t buzon = 5;

char str[16] = "123456789012345";    // los strings terminan con '\0'
int tabla[8] = { 60, 100, 200, 400, 800, 1500, 3000, 5000 };
struct {
  char car1, car2 ;
  int i;
  long long1, long2, long3 ;
} registro = { 'A', 'B', -77, 100000, 1000000, 10000000 } ;
...

enviaMsjBuzon (buzon, str);       // el mensaje es una cadena de caracteres
enviaMsjBuzon (5, tabla);         // el mensaje es una tabla de 8 enteros
enviaMsjBuzon (buzon, &registro); // el mensaje es un registro
```

Para recibir un mensaje, el proceso receptor debe indicar el buzón del cual quiere recibir el mensaje, así como la dirección de destino de los 16 bytes correspondientes al mensaje. La interpretación de la estructura del mensaje es competencia del proceso receptor, y lo normal

es que el proceso emisor y el receptor se hayan puesto de acuerdo de alguna manera en el formato de los mensajes. Lo lógico es que los mensajes se depositen en estructuras de datos análogas a las utilizadas por el emisor. Por ejemplo el proceso receptor de los tres mensajes enviados anteriormente podría proceder de la siguiente manera:

```
mboxDesc_t mailbox = 5;

char cadena[16];
int bufer[8];
struct {
  char c1, c2;
  int e;
  long l1, l2, l3;
} reg;
...

recibeMsjBuzon (5, cadena) ;
recibeMsjBuzon (mailbox, bufer) ;
recibeMsjBuzon (5, &reg) ;
```

Tras la ejecución de esas instrucciones por parte de los procesos emisor y receptor se cumplirían todas las igualdades siguientes:

```
cadena[0]=='1', cadena[1]=='2', ..., cadena[14]=='5', cadena[15]=='\0'
bufer[0]==60, bufer[1]==100, ..., bufer[7]==5000
reg.c1=='A', reg.e==-77, ..., reg.l3==10000000
```

Además de esas condiciones, el paso de mensajes impone la sincronización de los procesos que se comunican, y en esta sincronización interviene decisivamente la capacidad de los buzones (constante `CAPACIDAD`).

Una consecuencia de que *enviaMsjBuzon* y *recibeMsjBuzon* sean llamadas al sistema es que el sistema operativo asegura que se ejecuten como acciones atómicas e indivisibles. Dicho de otro modo, no puede haber dos procesos que estén ejecutando simultáneamente dichas funciones. Si dos procesos intentan ejecutar a la vez estas funciones, el sistema asegura que primero un proceso ejecutará su función y sólo después, el otro proceso podrá ejecutar la suya. Por tanto, el emisor ejecuta primero *enviaMsjBuzon* y el receptor ejecuta después *recibeMsjBuzon*, o sucede al contrario.

Cuando un proceso llama a *recibeMsjBuzon*, si hay algún mensaje en el buzón, el proceso recibe el mensaje (transfiriéndolo desde del buzón a la dirección de memoria especificada en la llamada) y sale de la llamada al sistema continuando su ejecución. Si por el contrario, no hay mensajes en el buzón, el proceso se bloquea y se queda al final de la cola de procesos que esperan recibir un mensaje de dicho buzón. El proceso debe permanecer bloqueado hasta que llegue un mensaje al buzón y le toque el turno de la cola de dicho buzón, es decir, que sea el primero de dicha cola. Cuando esto suceda, el mensaje debe ser transferido, al igual que en el caso anterior, desde el buzón a la dirección de memoria del proceso receptor y éste debe reanudar su ejecución (pasar a preparado). De lo dicho hasta ahora es fácil deducir que en la implementación de buzones debe haber un buffer para los mensajes (de tamaño máximo "`CAPACIDAD`") y una cola (tipo "`cola_t`") para los procesos en espera de mensaje.

Análogamente, cuando un proceso llama a *enviaMsjBuzon*, si hay espacio en el buffer del buzón, el mensaje se transfiere desde la dirección de memoria especificada en la llamada a dicho buffer, y el proceso sale de la llamada y continúa su ejecución. Si por el contrario, el buffer del buzón está lleno, el proceso se bloquea y se pone al final de la cola de procesos que esperan enviar un mensaje al buzón. El proceso debe permanecer bloqueado hasta que un proceso al llamar a *recibeMsjBuzon*, habilite espacio en el buffer del buzón para que el mensaje pendiente de entrega de este proceso pueda depositarse en el buzón y con ello reanudar su ejecución.

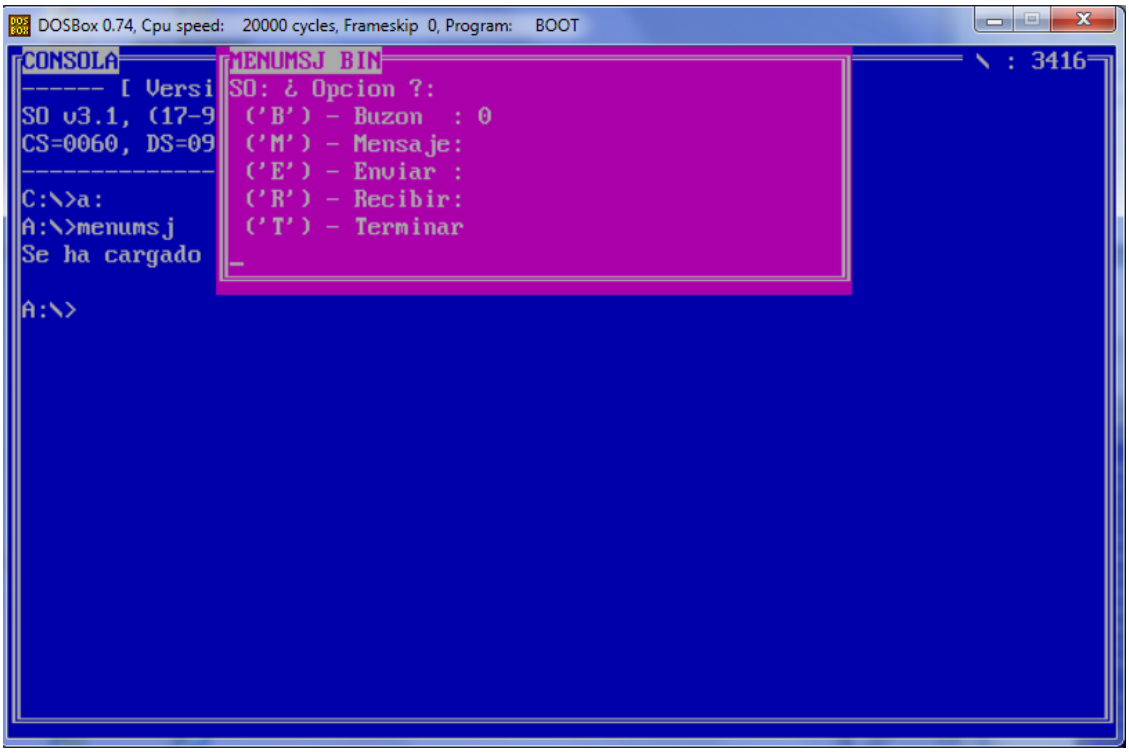
Como especificación del servicio de paso de mensajes hay que decir que los mensajes deben recibirse y enviarse por estricto orden cronológico. Es decir, si por ejemplo un proceso 'A' envía un mensaje a un buzón y posteriormente lo hacen los procesos 'B' , 'C' y 'D', cuando cualquier otro proceso intente recibir de dicho buzón, debe recibir el mensaje del proceso 'A', y si posteriormente este proceso o cualquier otro intente recibir más mensajes, recibirá respectivamente y por este orden, los mensajes de los procesos 'B', 'C' y 'D'. Este concepto es análogo al caso recíproco de intentos de recepción de los procesos 'A', 'B', 'C' y 'D' y de envíos posteriores de cualquier otro proceso.

De la exposición anterior se deduce inmediatamente una estructura de datos válida para representar los buzones en el núcleo de 'SO', utilizando en cada buzón una tabla circular (campo `bufer`) para representar la cola de mensajes contenidos en el buzón. Nótese que la cola del buzón puede contener en un momento dado procesos que están esperando dejar un mensaje en el buzón y en otro momento dado, procesos que están esperando recibir un mensaje del buzón. Nunca puede darse el caso de que haya procesos esperando dejar un mensaje y en el mismo momento, que haya procesos esperando recibir un mensaje del mismo buzón, por ello no es necesario implementar dos colas en el buzón (una para procesos receptores y otra para procesos destinatarios).

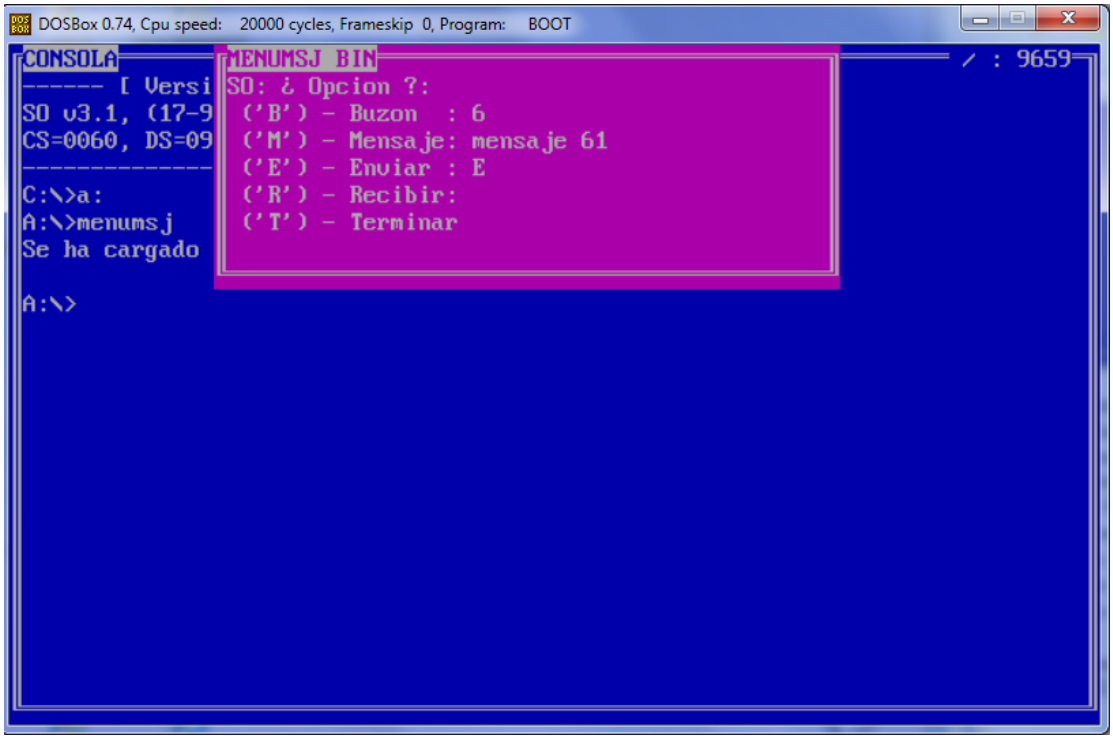
```
#define CAPACIDAD  2      // Capacidad de los buzones,
#define MBOX_MAX  10      // 10 buzones (mailbox)
#define LONG_MSJ  16      // longitud de los mensajes

struct {
  cola_t cola;
  unsigned char buffer[Capacidad][LONG_MSJ];
  unsigned in,out;   // buffer[in/out] primera entrada libre/ocupada
  unsigned numMensajes;                      // si numMensajes > 0
} buzon [MBOX_MAX];
```

5 EJERCITÁNDONOS CON EL PASO DE MENSAJES

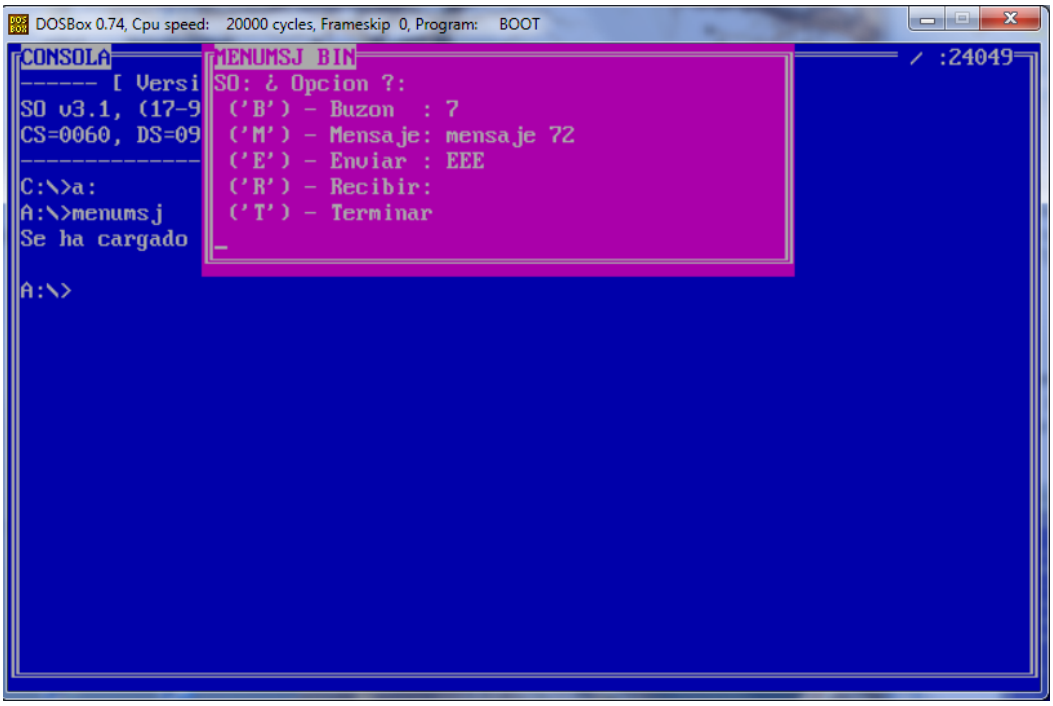
Para que el alumno capte de la forma más clara posible cómo debe funcionar el paso de mensajes lo mejor es que se ejercite con él dentro de 'SO'. El perfil *D-Fend* "***DBxBoot-SO***" ya conocido, permite arrancar desde disquete una versión de 'SO' que tiene implementadas todas las modificaciones solicitadas en las prácticas 3 y 4 (break, excepciones, sleep, semáforos y buzones). La capacidad de los buzones ha sido establecida en 'SO' mediante la declaración de la constante `CAPACIDAD` con el valor 2. Seguidamente arrancaremos desde dicho sistema el programa de usuario "*menumsj.bin*" situado en la unidad '**A:**', el cual proporciona un menú para que realicemos todas las operaciones relacionadas con el paso de mensajes:



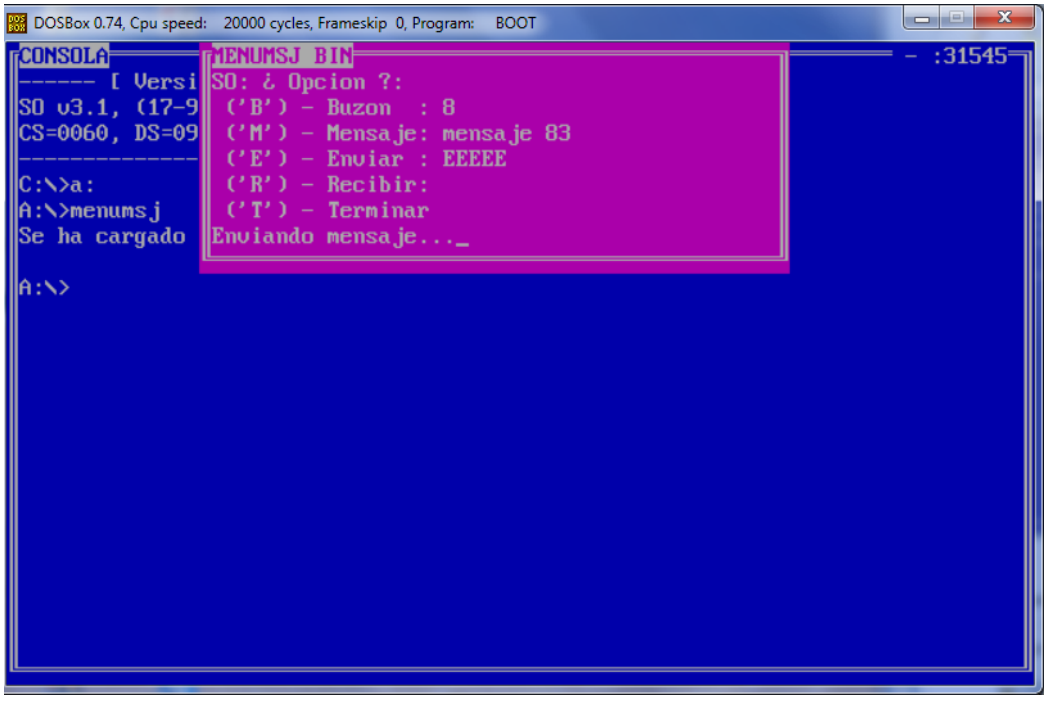
El buzón al que se envía o del que se recibe un mensaje figura tras los '**:**' de la opción '**B**', la cual nos permite seleccionar cualquier otro buzón, mediante un número entre el 0 y el 9. El mensaje que se envía es la última cadena de caracteres (de longitud máxima 15) que se haya establecido con la opción '**M**', el cual se muestra también tras los '**:**' en esta opción. Después de recibir un mensaje con la opción '**R**', el mensaje recibido también se muestra en el mismo sitio, pero aparecerá una 'R' tras los '**:**' de esta opción. Del mismo modo, cuando se envía un mensaje, aparecerá una 'E' tras los ':' de la opción '**E**'. Vamos a empezar enviando al buzón 6 el mensaje "mensaje 61" (pulsar: B, 6, M, "mensaje 61" <┘, E):



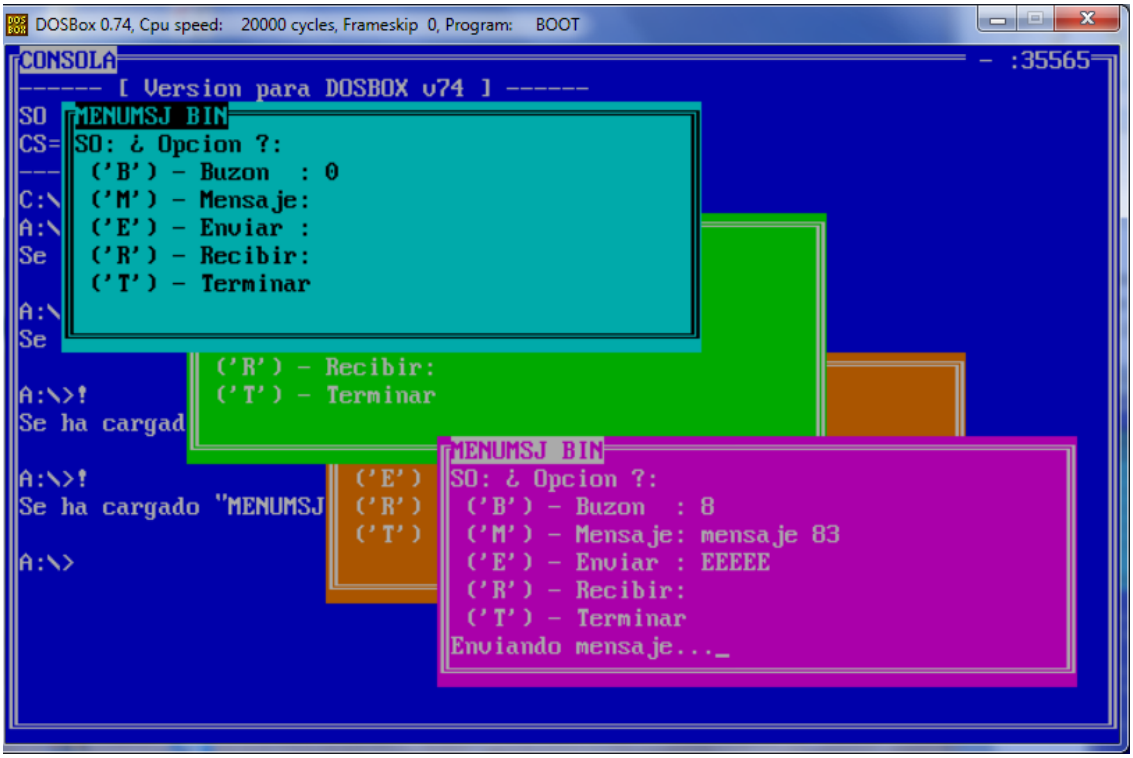
Vemos que el proceso 1 realiza el envío sin problemas, y el mensaje queda en el buzón 6. Ahora enviamos al buzón 7 los mensajes `"mensaje 71"` y `"mensaje 72"`, operaciones que también se realizan sin problemas, quedando los dos mensajes en el buzón 7 (pulsar: B, 7, M, “mensaje 71” <┘ , E, M, “mensaje 72” <┘ , E):



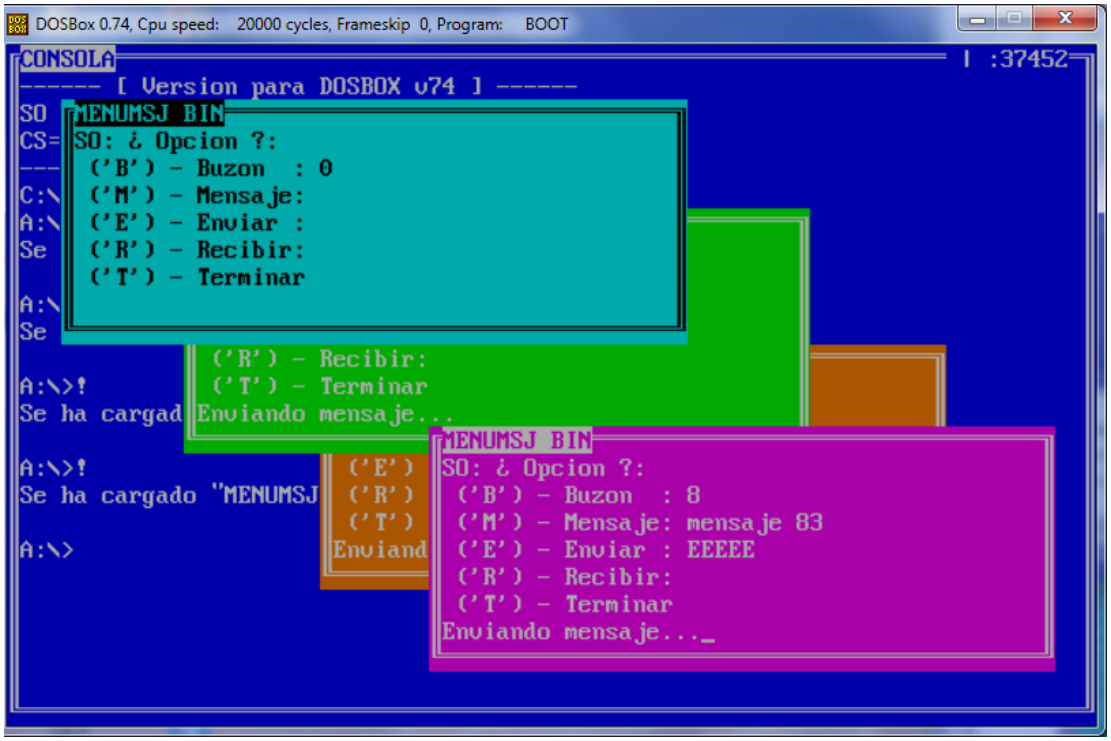
Seleccionamos ahora el buzón 8 y le enviamos los mensajes `"mensaje 81"`, `"mensaje 82"` y `"mensaje 83"`. Observamos que ahora el proceso queda bloqueado al intentar enviar el último mensaje ya que el buzón 8 está lleno a causa del envío de los dos primeros mensajes.



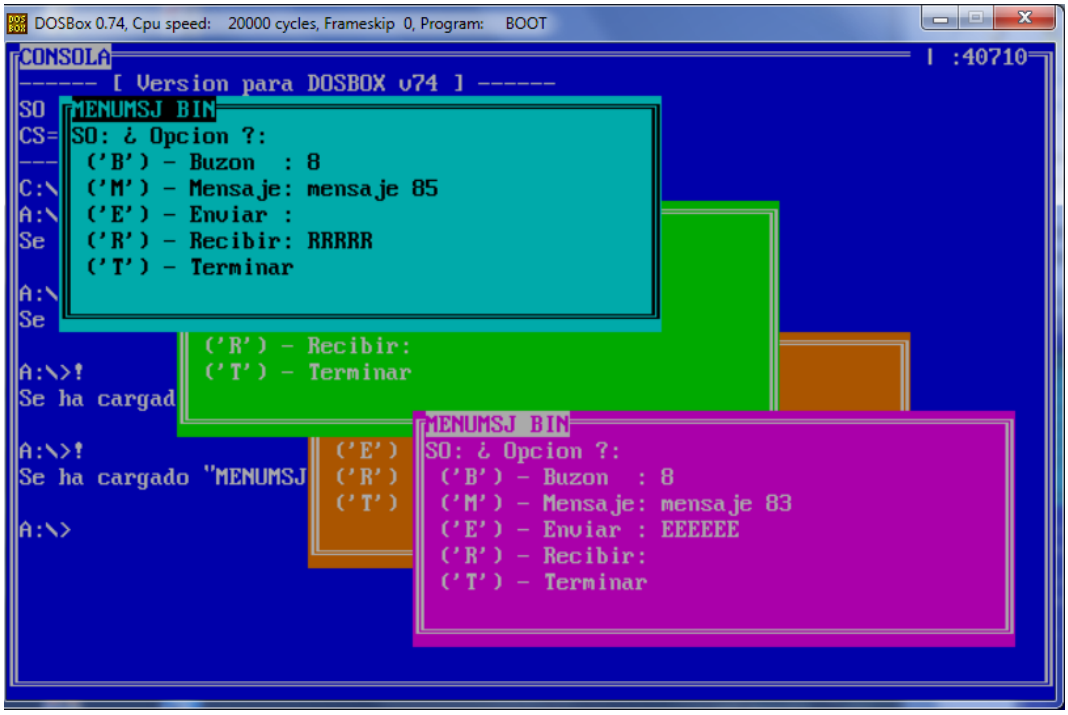
La única manera posible de desbloquear al proceso violeta (pid=2) es que otro proceso reciba un mensaje del buzón 8. Vamos a crear tres procesos más que ejecuten también el programa *"menumsj.bin"*.



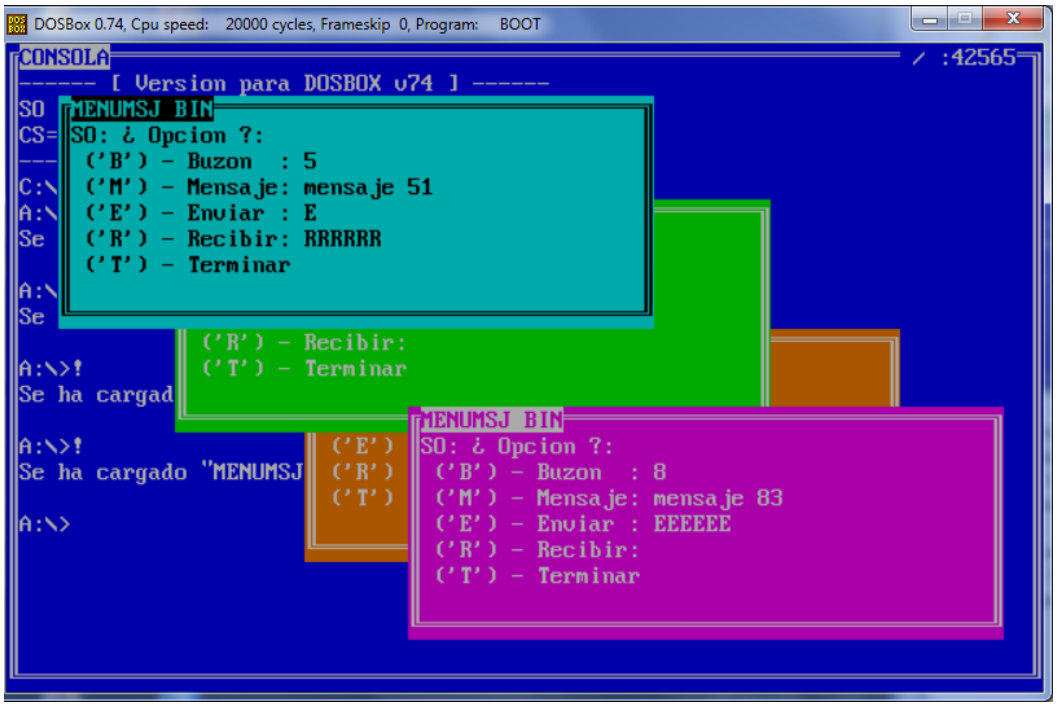
Ahora vamos a hacer que el proceso marrón (pid=3) envíe al buzón 8 el mensaje `"mensaje 84"`, y que el proceso verde (pid = 4) envíe al buzón 8 el mensaje `"mensaje 85"`. Evidentemente esos dos procesos quedarán también bloqueados.



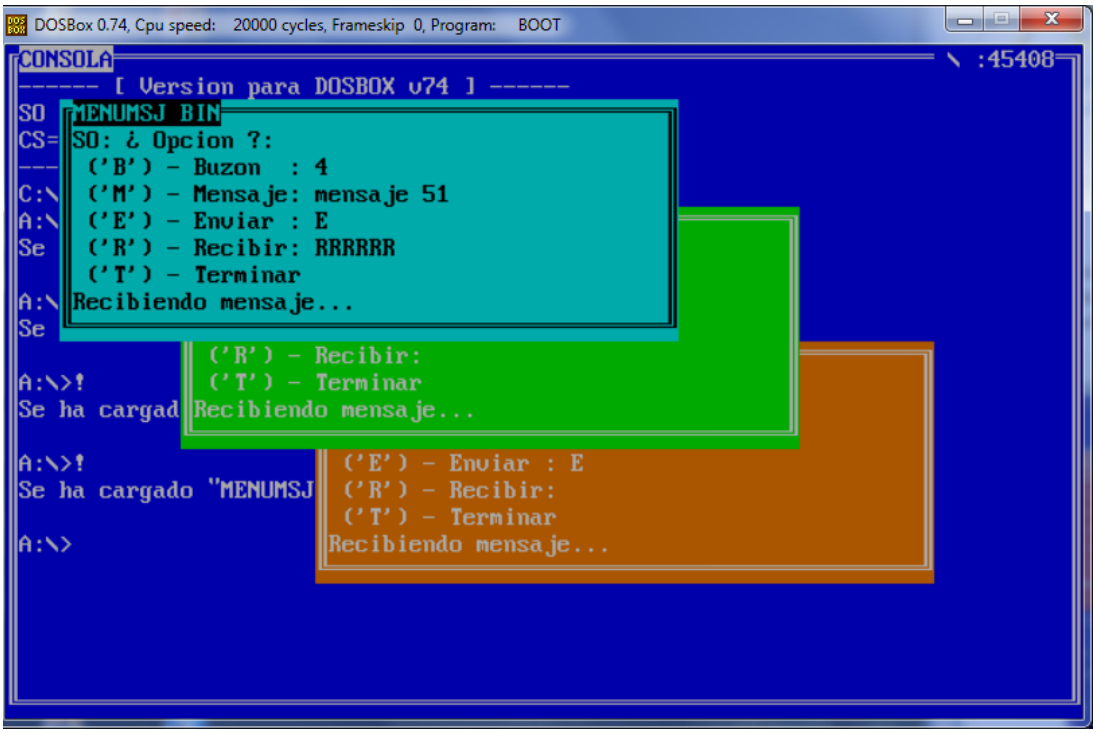
Ahora vamos a hacer que el proceso azul (pid=5) reciba del buzón 8 cinco mensajes. Observaremos que los mensajes se reciben exactamente en el orden en el que se han enviado (`"mensaje 81"`, `"mensaje 82"`, `"mensaje 83"`, `"mensaje 84"` y `"mensaje 85"`) y que en las tres primeras operaciones de recepción desbloquean sucesivamente a los procesos violeta, marrón y verde, siguiendo estrictamente la secuencia temporal en la que se bloquearon.



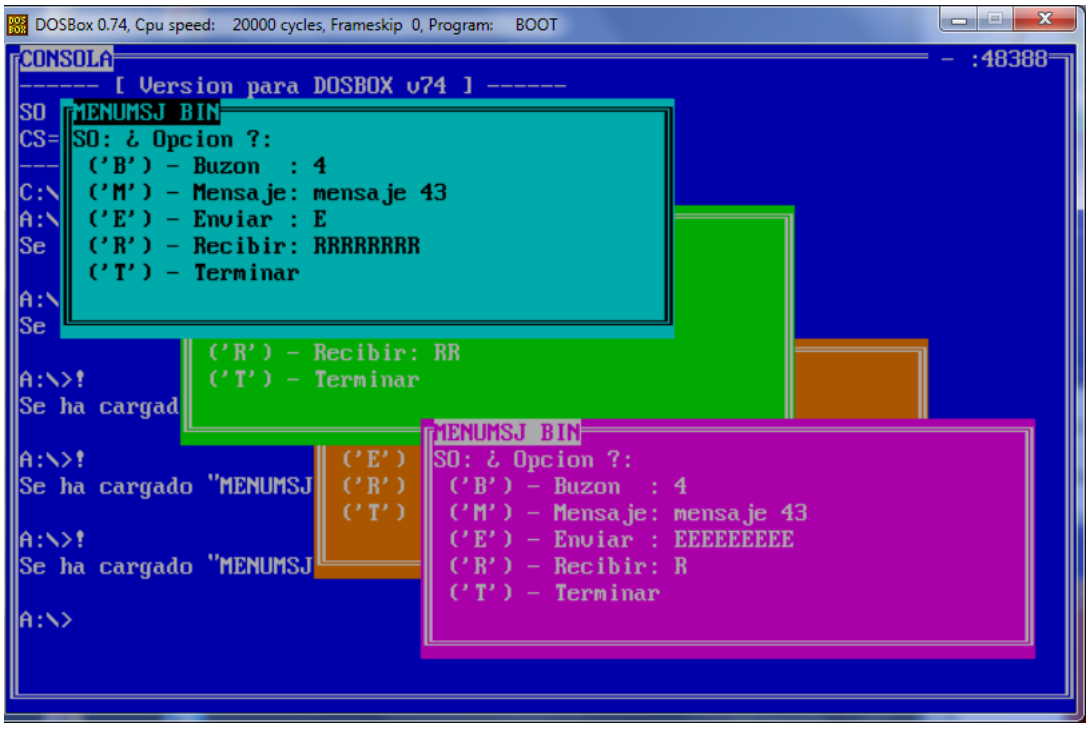
Ahora vamos a hacer que el proceso azul se envíe a través del buzón 5 el mensaje `"mensaje 51"` a sí mismo. Tras el envío debemos utilizar la opción `'m'` para introducir una cadena de caracteres vacía, borrando así el mensaje visualizado. Al recibir del buzón 5 debemos obtener el mismo mensaje que enviamos (`"mensaje 51"`).



Vamos a ver qué sucede ahora cuando varios procesos se quedan bloqueados intentando recibir de un buzón que está vacío. Para ello vamos a hacer que los procesos marrón, verde y azul intenten recibir (en ese orden) del buzón 4.



Si ahora el proceso violeta envía al buzón 4 los mensajes `"mensaje 41"`, `"mensaje 42"` y `"mensaje 43"` lo que debe suceder es que los procesos marrón, verde y azul se desbloquearán por ese orden al recibir los mensajes respectivos.



Lo visto hasta aquí debe ser suficiente para que el alumno se haya familiarizado suficientemente con las operaciones de envío y recepción de mensajes utilizando buzones de capacidad limitada, por lo que vamos a dejar el nivel del usuario, para pasar a ver a continuación algunos aspectos sobre el desarrollo de los buzones para el sistema 'SO'.

6 RUTINAS DE INTERFAZ DEL PASO DE MENSAJES

La implementación de las rutinas de interfaz que facilitan al usuario la realización de las llamadas al sistema correspondientes al paso de mensajes desde un programa en C son las siguientes:

```
/* ------------------------------------------------------------
       Funciones de llamadas al sistema para mensajes
   ------------------------------------------------------------ */
int enviaMsjBuzon (unsigned nBuz, void far *msj) {
  asm { MOV CX,nBuz
        LES BX,msj          // ES = SEGMENT msj, BX = OFFSET  msj
        MOV AH,0xE
        INT VINT_SO }  return _AX;
}//enviaMsjBuzon

int recibeMsjBuzon (unsigned nBuz, void far *msj) {
  asm { MOV CX,nBuz
        LES BX,msj          // ES = SEGMENT msj, BX = OFFSET  msj
        MOV AH,0xF
        INT VINT_SO }  return _AX;
}//recibeMsjBuzon
```

Estas funciones podrían definirse en un fichero independiente, pero por razones prácticas, es mejor incluirlas en el fichero de rutinas de interfaz ya existente "*c:\miso\ll_s_so\proc-ifz.c*".

Como puede apreciarse, en ambas funciones de interfaz el descriptor del buzón (`nBuz`) se pasa a través del registro **CX**, mientras que el segmento de la dirección del mensaje (`msj`) se pasa a través de **ES**, y su desplazamiento a través de **BX**. Como siempre el código de la operación se pasa a través del registro **AH**, siendo su valor **`0x0E`** para el caso de *enviaMsjBuzon*, y **`0x0F`** para el caso de *recibeMsjBuzon*.

Como ejemplo de programa de usuario que utiliza estas llamadas al sistema tenemos el programa "*c:\miso\usrs_prg\menumsj\menumsj.c*", el cual hemos utilizado en el apartado 6 para ejercitarnos en el uso del paso de mensajes.

7 IMPLEMENTACIÓN DE LAS FUNCIONES DE SERVICIO

Por seguir con la normativa sobre los nombres de las funciones de servicio es apropiado que los nombres de los manejadores de las llamadas al sistema correspondientes a *enviaMsjBuzon* y *recibeMsjBuzon* sean **`so_enviaMsjBuzon()`** y **`so_recibeMsjBuzon()`** respectivamente. Estas funciones, al igual que las demás, deben ubicarse en el fichero *"c:\miso\llamadas.c"*.

El significado del descriptor de un buzón (`nBuz`) utilizado en las funciones `enviaMsjBuzon` y `recibeMsjBuzon` debe ser el índice del buzón en la tabla `buzon`, es decir si `nBuz` es un descriptor de buzón, el acceso al mismo sería: `buzon[nBuz]`.

Tal y como ya se comentó para la llamada "*sleep*", es posible acceder fácilmente a cualquiera de los registros pasados en la pila a través de la macro:

```
#define RGP(N,R)(tblProc[N].sp->R)
```

Que se encuentra declarada en el fichero "*llamadas.c*".

Por ejemplo, desde la función de servicio `so_enviaMsjBuzon()`, se puede acceder al parámetro '`nBuz`' pasado en el registro **CX** del siguiente modo:

```
unsigned miBuz = RGP (nprAtnd,C.X)
```

El valor del parámetro '`msj`' que es un puntero, se envía a través de los registros **ES** y **BX**, y para formar un puntero mediante dos "*words*" disponemos de la macro PTR(S,O). En este caso por ejemplo podríamos usarlo del siguiente modo:

```
char far *miMsj = PTR(RGP(nprAtnd,es),RGP(nprAtnd,B.X))
```

Si deseamos conocer los nombres de todos los registros que se pueden usar con el parámetro '**R**' de la macro, podemos ver los campos de la estructura de datos "`regsPila_t`" declarada en el fichero "*c:\miso\tipos.h*".

Para realizar las copias de los mensajes desde una zona de memoria cualquiera a otra, necesitamos emplear punteros largos y sería deseable disponer de alguna función que realizara dicho trabajo. Afortunadamente disponemos de ella. La función '`_fmemcpy()`' de *turboC* que utiliza punteros largos y cuyo prototipo podemos incluir con `#include <mem.h>`. Por ejemplo, para copiar el mensaje cuya dirección está en '`pMsj`', al buffer '`i`' del buzon '`n`', haríamos lo siguiente:

```
_fmemcpy(&buzon[n].buf[i], pMsg, LONG_MSJ);     //LONG_MSJ=16
```

Un aspecto muy interesante es el siguiente ¿Dónde está (dónde se guarda) el mensaje de un proceso que se ha bloqueado al intentar enviar un mensaje? La respuesta es que el mensaje está en el espacio de direcciones del proceso emisor, y que la dirección de comienzo de ese mensaje está en los registros **ES:BX** de la trama de registros guardados en pila pertenecientes al contexto del proceso emisor. El sistema operativo no tiene ningún problema en acceder a esos campos para posteriormente recoger el mensaje y completar ese envío. La forma de hacerlo recomendada es mediante el uso de la macro que ya hemos visto: RGP (N,R). En este caso pondríamos en el parámetro '**N**' el numero de proceso que obtengamos de la cola del buzón, y como siempre en el '**R**' el registro, bien '**es**' o bien '**B.X'**.

Y análogamente, el problema anterior se vuelve a plantear y se resuelve de forma similar, para el caso en el que lo que queremos es obtener la dirección de la variable destino donde dejar un mensaje dirigido a un proceso receptor cuando éste está bloqueado.

Por último y para acabar, se recuerda que no hay que olvidar la inicialización de la tabla de buzones del sistema 'SO', estática o dinámicamente.

PRÁCTICA 5

Compactación de memoria

ÍNDICE

1 OBJETIVOS

Los objetivos de esta práctica son:

- Que el alumno profundice sobre el conocimiento de la gestión de memoria de un pequeño sistema operativo.
- Que el alumno sea capaz de implementar en un pequeño sistema operativo, un procedimiento para compactar los huecos de memoria que se van produciendo durante el ciclo de creación y destrucción de procesos, ó asignación y liberación del recurso memoria.

2 INTRODUCCIÓN

En esta práctica se pretende implementar un comando de la consola del sistema operativo de prácticas que compacte los huecos de memoria.

Durante la vida del sistema el recurso memoria es asignado en bloques de tamaño variable, que normalmente se utilizan para ubicar el código y los datos de los procesos. Cuando estos procesos mueren el sistema libera la memoria ocupada por ellos, produciéndose un hueco, o espacio de memoria contigua, de tamaño igual al bloque de memoria que se le asignó cuando fue creado. Estos huecos acaban eventualmente estando dispersos por toda la memoria, produciendo lo que se conoce como fragmentación externa de memoria. El objetivo de este comando es fusionar todos estos huecos dispersos en uno sólo, eliminando así dicha fragmentación.

3 TRABAJO A REALIZAR

Vamos a presuponer que el alumno conoce ya el funcionamiento del sistema 'SO', tanto a nivel de usuario, introduciendo comandos a través de la consola o haciendo llamadas al sistema desde un programa, como a nivel de la estructura del programa 'SO', puesto que se da por hecho que ha realizado todas las prácticas anteriores.

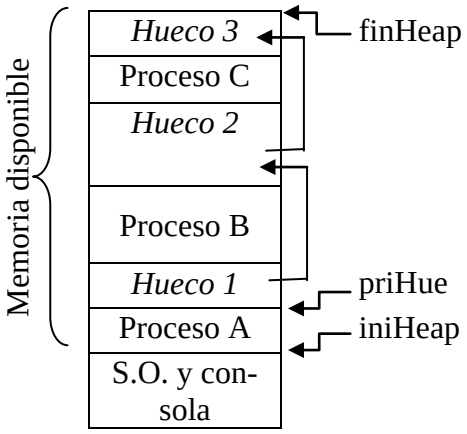
Tal y como ya se comentado en la introducción, el trabajo a realizar consiste en la implementación de un nuevo comando, al que llamaremos "*compac*" que se encargará de fusionar todos los huecos de memoria en uno sólo, eliminando completamente la fragmentación. Para conseguirlo, el comando tendrá que mover los objetos ubicados en la memoria asegurándose de que el cambio sea totalmente transparente para los procesos. Ello es posible gracias a que los procesos en 'SO' admiten reubicación dinámica, aunque eso sí, asumiendo algunas pequeñas restricciones. Actualmente todos los procesos de usuario disponibles como ejemplo las cumplen. En general la principal restricción consiste en que los procesos no deben emplear punteros largos para acceder a zonas de memoria que no le corresponden y que pudieran formar parte de algún objeto ubicado en memoria con posibilidad de reubicación. Esta restricción podría incluso superarse si se asumieran algunas reglas de comportamiento a seguir por dichos procesos y el propio 'SO'.

4 IMPLEMENTACIÓN DEL COMANDO 'COMPAC'

Nuestro objetivo es añadir este comando a la consola de 'SO', cuyo código se encuentra fundamentalmente dentro del fichero "so.c" perteneciente a los fuentes de 'SO'. En dicho fichero se declara el tipo "*simb_t*" que es un enumerado con todos los posibles comandos. Igualmente, se declara la tabla "*cmds[*]" que contiene los literales de estos mismos comandos, existiendo una correlación directa fácilmente visible entre ambas declaraciones. El añadir a estas declaraciones el nuevo comando es algo sencillo y un primer paso. También es posible y apropiado, añadir dicho comando a los mensajes informativos que muestra el comando ya existente "ayuda" en este momento, aunque esto por supuesto no es estrictamente necesario, ya que no influye directamente en los objetivos del nuevo comando.

Con los cambios indicados en el punto anterior ya podemos incorporar una nueva etiqueta "*case*" al "*switch*" principal de la función "*main()*". Este "*switch*" se encarga de distribuir la ejecución de cada uno de los comandos de 'SO'. La modificación podría consistir simplemente en invocar una nueva función, llamada por ejemplo "compactaMem()", que sería la que tendría que llevar a cabo todo el trabajo. El código de esta función debería estar incluido por coherencia con el resto en el fichero, en el módulo "*memoria.c*", y su prototipo en "*memoria.h*". De este modo podría ser utilizado desde "so.c". Hasta aquí con todo esto, habríamos realizado todos los cambios necesarios en el modulo "so.c", aunque hay que señalar que esta parte es la más sencilla del trabajo a realizar.

En el módulo "*memoria.c*" se encuentra el código fuente que contempla todos los aspectos relativos y relevantes acerca de la gestión de memoria en 'SO', aunque para entender y llegar a implementar el trabajo pedido tendremos que estudiar también los módulos: "*ficheros.c*", "*procesos.c*" y "*windows.c*", a los que se hará referencia más adelante, indicando aquellos aspectos que hay que conocer para poder llevar a cabo la implementación del comando "*compac*". En la figura siguiente se muestra un ejemplo de mapa de memoria.



En él se muestran tres procesos cargados en memoria y también tres huecos. En la zona inferior, en las direcciones más bajas de memoria se encuentra 'SO', donde a su vez se halla en él, en su espacio de memoria, el proceso consola. Justo donde acaba la última dirección de memoria del 'SO' empieza la memoria de uso dinámico para asignación. El valor de comienzo de esta zona lo indica la variable "*iniHeap*". La variable "*finHeap*" indica el final de la memo-

ria disponible. La operación "*finHeap*"-"*iniHeap*" nos da el valor de la variable "*memDisponible*". Todas estas variables la establece 'SO' durante el proceso de inicialización. El tipo de datos de estas variables es "*word_t*" (unsigned int) y por tanto lo que representan son direcciones de segmento en el caso "*iniHeap* y *finHeap*" y *paragraphs* (16 bytes) en el caso de "*memDisponible*".

En el módulo memoria están implementadas las funciones "T*omaMem()*" y "S*ueltaMem()*" que sirven respectivamente para solicitar memoria y devolverla, y aunque son muy importantes, en principio no es necesario conocerlas en profundidad para realizar el trabajo pedido. La función "*mostrarMemoria*", que se encarga de mostrar en pantalla la lista de huecos de memoria a petición del comando "*mem*", es interesante estudiarla, ya que nos permite ver como se recorre la lista de huecos, no obstante tampoco esto es estrictamente necesario para cumplir nuestros objetivos, aunque eso sí, usaremos el comando "*mem*" durante el desarrollo de la práctica para ver los resultados de nuestro trabajo.

La lista de huecos está implementada usando los propios huecos para guardar los enlaces de la lista. El primer hueco lo indica la variable "*priHue*" y los siguientes se obtienen accediendo a la zona del hueco mediante punteros y utilizando el campo "*uSig*" para acceder al siguiente hueco y el campo "*tam*" para conocer el tamaño del hueco en "*paragraphs*" (unidades de 16 bytes). Los tipos de datos empleados se muestran a continuación:

```
/* ------------------------------------------------------------------------
   Tipos de datos para crear una lista de huecos en la memoria disponible
   para programas. Los "links" se guardan en los propios huecos. Especifi-
   can el tamaño del   hueco y un puntero al siguiente. Este puntero es del
   tipo "_seg" que es especial ya que tiene siempre un offset implícito '0'
   que no ocupa espacio. Hago una unión con el tipo "word_" para operar me-
   jor con él.
   ------------------------------------------------------------------------ */

typedef struct foo _seg *pHue_t; // offset siempre = 0 (ocupa un word)
typedef union {
  pHue_t p;    // dato que se puede ver como puntero sin offset
  word_t w;    // ó como un word
} uHue_t;
struct foo {  word_t tam;  uHue_t uSig; };
/* ------------------------------------------------------------------------*/
```

El empleo de la estructura "*foo*" es por motivos sintácticos. Nos permite la declaración de campos del tipo "*pHue_t*" dentro de la unión "*uHue_t*". Lo que nos interesa es saber que tenemos un tipo de datos "*uHue_t*" que es una unión (es decir los dos campos dentro de ella, "p" y "w" permiten el acceso a una sola cosa, pero de dos formas distintas, como puntero y como entero sin signo), y otro tipo "*pHue_t*" puntero a "*struct foo*", que usaremos para acceder al interior de los huecos, para obtener el siguiente de la lista y el tamaño del mismo. Por ejemplo la variable "*priHue*" que es del tipo "*uHue_t*" podemos usarla escribiendo "*priHue.p*" para acceder a la dirección del primer hueco. Para acceder al tamaño del primer hueco pondríamos: "*priHue.p->tam*", ó bien para acceder al siguiente hueco al primero: "*priHue.p->uSig*", etc. Si por otra parte queremos obtener el valor del segmento de la dirección contenida en dicho puntero para copiarlo a una variable tipo "*word_t*" para operar con él usaríamos entonces "*priHue.w*". Para familiarizarse con estas operaciones es aconsejable estudiar un poco el código de este módulo.

A continuación vamos a explicar a grandes rasgos una posible forma de llevar a cabo la compactación.

Para compactar es necesario conocer las direcciones y tamaños de todos los objetos ubicados en la memoria de asignación dinámica. Estos objetos actualmente son sólo las FAT de los drives, los procesos y las ventanas. Para obtener estos datos hay que examinar el contenido de la tabla de información de las unidades de disco (***infDrv[]***) y la tabla de procesos (***tblProc[]***), para ir rellenando, en una nueva tabla de objetos, las direcciones y tamaños de los objetos ubicados en memoria que hemos obtenido de dicho examen. Estos objetos son, en el caso de las unidades de disco, la '**FAT**' de cada unidad montada, en el caso de los procesos, las áreas de código y datos del mismo, y en el caso de las ventanas, el objeto '**win_t**' que guarda toda la información sobre una ventana terminal. La introducción de estos datos en la nueva tabla debe ser de tal modo que la tabla debe quedar al final ordenada por direcciones, preferiblemente de menor a mayor.

Una vez que tengamos registrados en la nueva tabla todas las direcciones y tamaños de todos objetos que el sistema 'SO' tiene ubicados en memoria dinámica, y además tengamos los elementos de esta tabla ordenados por su dirección de memoria, podremos recorrer esta tabla siguiendo su ordenación, e iremos comprobando por cada elemento de la tabla, si su dirección es menor que una previamente establecida a la que llamaremos "**base**", cuyo valor inicial es la del primer hueco de la lista. Si la dirección del objeto fuera menor que la de la "base" no habría que reubicar el objeto y pasaríamos al siguiente objeto de la tabla, si por el contrario fuera superior, habría que reubicar el objeto, es decir habría que mover el objeto (FAT, proceso o ventana) desde su posición de memoria actual, a la dirección de "base", y en este caso actualizaríamos al nuevo valor "base = base + tamaño del objeto reubicado". Cuando hayamos llegado al último objeto de la tabla tendríamos en "base" la dirección del único hueco resultante de este proceso, siendo su tamaño igual a la diferencia entre *finHeap* y *base* (tam = finHeap - base).

El procedimiento visto hasta ahora reubicaría los objetos, pero habría que hacer más cosas para que el sistema siga funcionando correctamente. Habría que modificar también todas las direcciones o punteros que el sistema 'SO' mantiene sobre aquellos objetos que se han reubicado. De estos punteros sólo haría falta cambiar la parte de "segmento" de la dirección, ya que la reubicación se ha efectuado siempre con alineamiento a "*paragraph*".

Seguidamente vamos a comentar en función del objeto reubicado, qué punteros hay que modificar y dónde se encuentran éstos dentro de 'SO'.

Si el objeto reubicado es una 'FAT' basta con modificar el puntero a la misma que se encuentra en la tabla de información de unidades de disco (*infDrv[]*). Esta tabla está definida en "*ficheros.h*", y en ella tenemos el campo "**pFat**" que es un puntero largo a byte (*fptrb_t*) que apunta a la FAT de la unidad (si es que está montada, sino valdría NIL). Si durante la reubicación hemos tomado nota del nuevo segmento de reubicación y de qué FAT se trata, podríamos hacer la modificación utilizando la macro *SEG(p)* que nos permite acceder a la parte "segmento" de un puntero. Por ejemplo, supongamos que "*newSeg*" es el nuevo segmento de dirección de memoria del objeto FAT reubicado correspondiente a la unidad "*drv*", entonces podríamos escribir:

```
SEG(infDrv[drv].pFat) = newSeg;
```

Alternativamente, también podríamos usar la macro *PTR(S,O)*, que nos construye un puntero a partir de un segmento (O) y un desplazamiento (offset, O), y podríamos escribir:

```
infDrv[drv].pFat = PTR(newSeg,0);
```

Ya que las direcciones de todos los objetos ubicados en memoria dinámica tienen siempre el offset a 0 (ya que las funciones de tomar memoria y soltar memoria devuelve un valor del tipo word que resulta ser el segmento de la zona de memoria que se asigna o libera).

Pasemos a analizar el caso de que el objeto reubicado fuera un 'proceso'. En este caso habría que hacer los siguientes cambios:

a) Modificar la dirección de memoria (segmento) especificada en el descriptor del proceso que indica dónde está cargado el proceso en memoria (***dirMem***).

b) Modificar el segmento del puntero (***sp***) que guarda la dirección de la cima de la pila (puntero de pila o *stack pointer*) donde se guardan los registros del contexto del proceso.

c) Modificar los registros **CS, DS** y **ES**, que se hallan en la trama de pila del contexto del proceso con el nuevo valor de segmento reubicado (*dirMem*). Para acceder a estos valores podemos utilizar el valor de '***sp***' del siguiente modo: `tblProc[np].sp->cs=dirMem;` donde '***np***' es el numero de proceso reubicado y '***dirMem***' la nueva dirección de segmento. Los otros dos registros, DS y ES, dado el modelo de memoria que utilizan los procesos de usuario, podemos asumir perfectamente que tendrán el mismo valor que el registro CS, por ello la modificación es similar a la ya vista.

En este momento conviene hacer la siguiente reflexión: El descriptor de proceso tiene también un campo puntero llamado '***pWin***' que contiene la dirección de la ventana terminal del proceso, la cual es uno de los objetos de posible reubicación. En el momento en el que se están haciendo los cambios relacionados con la reubicación del proceso, hemos modificado los punteros que hacían referencia a la nueva posición del proceso, retocando ciertos campos del descriptor del proceso, sin embargo este campo '**pWin**' hace referencia a la ventana del proceso, la cual puede haber cambiado de posición o no. Este cambio se tratará cuando se efectúe el proceso de retoque de los punteros mantenidos por 'SO' referentes a las ventanas (lista de ventanas), lo cual se comentará en el punto siguiente.

Vamos a tratar por último el caso en que el objeto reubicado sea una ventana terminal. Este caso es algo más complejo ya que las ventanas forman parte de una lista que esta ordenada por la posición que ocupa la ventana en la pantalla (primer plano, segundo plano, etc.) y hay que modificar los punteros de dicha lista referentes a las ventanas que se hayan reubicado. En primer lugar vamos a ver algunas de las estructuras de datos usadas por 'SO' para el manejo de ventanas, las cuales se encuentran definidas en el fichero "*windows.h*":

```
typedef struct wfoo win_t;
typedef win_t _seg *pWin_t;

#define SIZE_KEYBUF 127  // no pasar de 255
struct wfoo {
  char nombre[12];       // nombre de la ventana
  // --- Información de ventana ---
  pantalla_t  plano;     // guarda la información visual (una pantalla)
  int         dx, dy;    // desplazamiento del plano respecto a la pantalla
  pos_t       eSI, eID;  // coordenadas del marco de ventana
  byte_t      atr;       // atributo normal de la ventana
```

```
  pos_t       cursor;     // posición del cursor
  pWin_t pWDw, pWUp;      // ventana de debajo y de encima
  // --- Informacion de teclado ---
  byte_t keyBuf [SIZE_KEYBUF];
  byte_t nKeys;           // cantidad de teclas en el buffer
  byte_t ent,sal;         // posiciones de tecla entrante y saliente
  cola_t cola;            // cola de procesos esperando recibir tecla
};
extern pWin_t pWinTop;
extern pWin_t pWinFocal;  // ventana con el foco del teclado
```

Las ventanas de los procesos están organizadas formando una lista que está ordenada con arreglo a la posición que ocupan a la hora de ser mostradas en pantalla. Si suponemos una tercera dimensión, la profundidad, las ventanas se muestran y en pantalla ocultándose unas a otras. Podemos hablar de las ventanas que están encima y de las que están debajo, o también la ventana de primer plano, de la ventana del fondo, del fondo de pantalla, etc. Pues bien las ventanas forman una lista donde la primera es la que se presenta en primer plano y la última la que se encuentra más al fondo o en último plano.

Del listado de código de arriba, nos interesa sobre todo la variable "***pWinTop***" que es un puntero a la primera ventana de la lista (ventana de primer plano) y los campos '***pWDw***' y '***pWUp***' de la estructura '***win_t***', que son los punteros que tiene toda ventana apuntando a la ventana que se encuentra debajo y arriba de ella respectivamente. Son estos punteros los que hay que modificar para que se ajusten a las nuevas posiciones de memoria de las ventanas cuando éstas se reubican.

No hay que olvidar que además de los cambios en esta lista, también habría que cambiar el campo '***pWin***' del descriptor del proceso, tal y como ya se ha comentado con anterioridad.

Es conveniente por último hacer dos consideraciones. 1) El procedimiento de compactación debe efectuarse con las interrupciones inhibidas, ya que si durante la reubicación se intentase efectuar un cambio de proceso el resultado podría sería desastroso, y 2) A la hora de mover los objetos en memoria hay que tener en cuenta el sentido de la copia por posibles solapamientos. TC aporta una función que tiene esto en cuenta "***movemem()***" pero sólo admite como parámetros punteros cortos, y necesitamos punteros largos y por otro lado, las funciones de copia que admiten punteros largos como parámetros (***_fmemcpy()***..), dejan indefinido el resultado si hay solapamiento, por lo que tampoco pueden usarse, por lo tanto a falta de una solución mejor, habrá que implementar una función específica de copia para conseguir nuestros propósitos. Para facilitar esta tarea al alumno, se muestra a continuación una posible implementación de esta función:

```
/* Mueve 'sz' paragraphs desde el segmento 'o' al segmento 'd'. lo
   hace en unidades dword (4 bytes) para mejorar la velocidad. lo movido
   debe ser menor de 64KB y 'd' debe ser menor que 'o' si se solapan */
void mueveMem (word_t d, word_t o, word_t sz) {
  word_t i;
  sz <<= 4; // multiplico por 16 para pasar a bytes
  for (i=0; i < sz; i+=4)
    *((fptrd_t)PTR(d,i)) = *((fptrd_t)PTR(o,i));
}//mueveMem
```