

El cargador del sistema operativo OSO

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia

27 de abril de 2005

Índice

| | |
|--|----------|
| 1. Introducción | 1 |
| 2. Arranque de un ordenador PC-compatible | 1 |
| 3. El programa cargador | 2 |
| 3.1. El fichero <code>cargador.c</code> | 3 |
| 3.2. La función <code>cargarSO</code> | 5 |
| 3.3. Salto del cargador al sistema operativo | 5 |
| 3.4. El fichero <code>c0t_car.asm</code> | 6 |
| 4. El programa <code>mkboot32</code> | 7 |
| 5. El programa <code>mkboot</code> | 9 |

1. Introducción

En este documento vamos a describir un programa cargador que se podrá almacenar en el sector de arranque de un dispositivo de almacenamiento que posea un sistema de ficheros FAT. Sus funciones serán las de cargar en una posición determinada de memoria el código de un sencillo sistema operativo denominado OSO, establecer un entorno de ejecución adecuado para dicho sistema operativo, cargando en ciertos registros del procesador valores adecuados, y ceder el control de la CPU al sistema operativo cargado.

Para entender mejor como funciona dicho cargador, vamos a ver primero cómo arranca un ordenador PC-compatible.

2. Arranque de un ordenador PC-compatible

Cuando se arranca un ordenador su propio hardware comprueba la presencia de, e inicializa, una serie de dispositivos que son esenciales para el funcionamiento del sistema. Al finalizar todas las comprobaciones el hardware carga en memoria el contenido del primer sector («sector de arranque») de un disco y le cede el control de la CPU. A partir de ese momento lo que haga el ordenador dependerá del programa almacenado en dicho sector de arranque.

En los ordenadores PC compatibles el disco del que se carga el sector de arranque se puede configurar mediante el programa «setup». Aunque se puede especificar un único dispositivo de arranque, lo habitual es indicar una secuencia de dispositivos de tal manera que el hardware cargará en memoria el sector de arranque del primer dispositivo disponible. Generalmente, el primer dispositivo de arranque indicado en dicha secuencia suele ser el disco duro que posee el ordenador (o uno de ellos si posee varios) seguido de otros dispositivos de almacenamiento como disquete, CD-ROM, etc. El orden de estos dispositivos se puede cambiar para así poder arrancar el ordenador directamente desde, por ejemplo, un disquete o un CD-ROM.

Los ordenadores PC-compatibles actuales arrancan en lo que se denomina «modo real» del procesador. En este modo el procesador se comporta como un antiguo procesador 8086 de Intel y ofrece una visión del hardware muy sencilla.

El problema del modo real es que no es el más adecuado para la ejecución de muchos sistemas operativos modernos por lo que es habitual que, durante el arranque del sistema operativo, se pase del modo real al «modo protegido» del procesador. El modo protegido presenta una visión mucho más compleja del hardware pero también ofrece ciertos mecanismos hardware que son esenciales para los sistemas operativos actuales como la memoria virtual, el modo dual de funcionamiento del procesador, etc.

3. El programa cargador

Como hemos comentado antes, durante el arranque, y tras ciertas comprobaciones, el hardware carga en memoria el programa contenido en un sector de arranque. La dirección de memoria en la que se hace la copia es la 0:7C00¹, a la que se salta para ceder la CPU al programa cargador.

El programa cargador se ejecuta en el modo real. En este modo, sólo hay 20 bits para direccionar la memoria, es decir, se puede usar hasta 1 MB de memoria. De ese megabyte, el primer KB contiene la tabla de interrupciones y los siguientes 256 bytes contienen variables utilizadas por la propia BIOS. Por lo tanto, el sistema operativo se podrá cargar a partir de la dirección 1280 o 50:0 de memoria.

Para simplificar el diseño del cargador, vamos a suponer que el sistema operativo a cargar se encuentra en un disquete de 3.5", de 1.44MB de capacidad, con un sistema de ficheros FAT12 que es el sistema de ficheros usado por MS-DOS y Windows para dichos disquetes. También vamos a suponer que el fichero que contiene el núcleo del sistema operativo es el primer fichero creado en el disquete y que tiene un tamaño máximo de 8 KB.

Al ser el primer fichero creado su contenido se encontrará tras el directorio raíz. En un disquete como el descrito esto supone que el fichero se encontrará en sectores consecutivos a partir del sector 33 (si vemos el disquete como un array lineal de sectores numerados desde 0 hasta 2879).

La función del cargador en este caso es muy simple: sólo tiene que cargar en memoria 16 sectores a partir del sector 33 del disquete. Observe que no es importante ni el nombre ni el tamaño del fichero que contiene el núcleo del sistema operativo, siempre que dicho fichero sea el primer fichero creado y siempre que no supere los 8 KB de tamaño. Tampoco es importante el hecho de que se carguen en memoria sectores que no contienen código del sistema operativo. De hecho los últimos sectores cargados podrían contener «basura».

El fichero que contiene el núcleo del sistema operativo debe tener formato COM. En este formato de ejecutable los programas se cargan y ejecutan a partir de la dirección X:100 de memoria. En nuestro caso, el sistema operativo se va a cargar y ejecutar a partir de la dirección 50:100. Los 256 bytes anteriores, desde 50:0 hasta 50:FF, se dejarán libres (en MS-DOS estos 256 bytes se utilizan para almacenar distintos datos que no son importantes en nuestro caso).

El utilizar un disquete con un sistema de ficheros FAT12 nos va a facilitar el desarrollo del sistema operativo OSO ya que vamos a poder copiar y borrar ficheros del disquete haciendo uso de los programas que incluye el propio sistema operativo Windows. Sin embargo, esta comodidad también presenta un problema y es que el cargador debe ser pequeño.

¹Recordemos que una dirección X:Y, donde X e Y son números en hexadecimal, se corresponde con la posición de memoria X*16+Y

En un sistema FAT12 o FAT16 el sector de arranque no sólo contiene el código del cargador sino también datos sobre el propio sistema de ficheros (sectores por bloque lógico o «cluster», tamaño del directorio raíz, etc.) y una posible tabla de particiones. Estas dos estructuras de datos reducen el espacio disponible para el cargador en el sector de arranque de 512 a 384 bytes.

Otro aspecto importante a tener en cuenta es que el cargador sólo va a disponer de los servicios ofrecidos por la BIOS para realizar su trabajo, ya que no existe ningún sistema operativo en memoria cuando se ejecuta el cargador.

Vamos a describir ahora los dos ficheros que componen nuestro programa cargador, `cargador.c` y `c0t_car.asm`, y algunos aspectos importantes de los mismos. Debe tener presente que los programas están preparados para ser compilados con la versión 3.0 de Turbo C++ que sólo genera ficheros COM y EXE de MS-DOS. Esta observación es especialmente importante a la hora de analizar los ficheros en ensamblador ya que determinadas instrucciones y directivas son particulares de este compilador.

3.1. El fichero `cargador.c`

A continuación mostramos el código del cargador que hemos implementado y que cumple todo lo que hemos descrito hasta ahora.

```

/* - FICHERO: cargador.c
* - DESCRIPCIÓN: contiene el código de nuestro cargador de sistemas
* operativos.
* - REQUISITOS: disquete de 3.5" y 1.44MB formateado en MS-DOS. El código
* del SO debe ser un fichero .COM y debe ser el primer fichero del
* directorio raíz (sector 33, numerando desde 0). El tamaño máximo de
* ese fichero no debe ser superior a 8 KB.
* - RESULTADO: carga el código del sistema operativo en X:0100h, con todos
* los registros de segmento correctamente inicializados para un fichero
* .COM
*/
5

/* Dirección donde se cargará en memoria el sistema operativo */
#define DIRECCION_SO ((char far *)0x00500100)
15

/* Tamaño máximo del sistema operativo: 8 KB */
#define SECTORES_SO 16

static void escribe_cad(char far * cadena, int longitud)
20
{
    asm {
        mov ah, 03h
        mov bh, 0
        int 10h
        mov ah, 13h
        mov al, 1
        mov bl, 07h
        mov cx, longitud /* Cuidado, bp se utiliza para acceder a longitud, y se
                        * modifica justo a continuación. */
        les bp, dword ptr cadena
        int 10h
    }
    }
30

unsigned int
leersector (unsigned char far *buffer, char pista, char cabeza, char sector)
35
{
    int i;

    for(i = 0; i < 3; i++)
40
    {
        asm {

```

```

        les    bx, dword ptr buffer
        mov    al, 1          /* 1 sector */
        mov    ch, pista
        mov    cl, sector
        mov    dh, cabeza
        mov    dl, 0         /* Unidad 0 */
        mov    ah, 02       /* De disco a memoria. */
        int    13h
        xor    ax, ax        /* Devolvemos 0 indicando que la operacion tuvo éxito. */
        jnc    noError      /* Si no se produce error salimos del bucle. */
        dec    ax           /* Devolvemos -1 indicando que la operacion falló. */
    }
noError:
}

unsigned int cargarSO (unsigned char far *buffer)
{
    char pista = 0, sector = 16, cabeza = 1;
    char i;

    for(i = 0; i < SECTORES_SO; i++)
    {
        if (leersector(buffer, pista, cabeza, sector))
            return 1;
        escribe_cad(".", 1);
        sector++;
        if (sector > 18)
        {
            sector = 1;
            cabeza++;
            if (cabeza >= 2)
            {
                cabeza = 0;
                pista++;
            }
        }
        buffer += 512;
    }
    return 0;
}

void main(void)
{
    unsigned char far * direccionSO = DIRECCION_SO;

    escribe_cad("Cargando SO", 11);
    if (!cargarSO(direccionSO))
    {
        escribe_cad(" Correcto\n\r", 11);

        /* Inhabilitamos las interrupciones, establecemos un entorno
           adecuado para la ejecución del sistema operativo y saltamos
           a la dirección de inicio del mismo que es 0050:0100 */
        asm {
            cli
            les bx, dword ptr direccionSO
            mov ax, es
            mov ds, ax
            mov ss, ax
            mov sp, 0FB00h
            push ax
            push bx
            retf
        }
    }
}

```

```

        /* Aunque las 3 últimas instrucciones se pueden sustituir por
        db 234
        db 00
        db 01
        db 50
        db 00
        que codifican un jmp 0050:0100, lo anterior es mas claro.
        Además, tcc no ensambla dicha instrucción de salto, da error.
        */
    }
}
while (1)
    escribe_cad(" Fallo", 7);
}

/* $Id: cargador.c 1002 2005-04-26 10:31:51Z dsevilla $ */

```

Como podemos observar, la función `main` básicamente hace dos cosas: llamar a la función `cargarSO`, para cargar el código del sistema operativo en memoria, y saltar a la dirección de comienzo del sistema operativo, tras inicializar adecuadamente distintos registros. Estos dos pasos los vamos a describir en detalle en los siguientes apartados.

La otra función utilizada, `escribe_cad`, es una función auxiliar que se utiliza para mostrar distintas cadenas de caracteres por pantalla. Esta función no es estrictamente necesaria y se podría eliminar si fuera necesario.

3.2. La función `cargarSO`

La función `cargarSO` lee 16 sectores consecutivos de disco y los copia a partir de la dirección de memoria pasada como parámetro (que en nuestro caso es 50:100).

Los 16 sectores a leer se encuentran a partir del sector 33 del disquete, siempre que veamos el disquete como un array lineal de sectores. En realidad, la estructura de un disquete de 1.44MB es la siguiente: 80 pistas, 18 sectores por pista y 2 cabezas. Si numeramos las pistas y las cabezas a partir de 0 y los sectores a partir de 1, entonces podemos decir que el sector 33 se encuentra en el sector 16 de la pista 0 y de la cabeza 1. A partir de esta posición, la función `cargarSO` lee 16 sectores de disco utilizando la función `leersector`, cambiando de sector, cabeza y pista según sea necesario. También incrementa la dirección de memoria en 512 bytes tras leer cada sector ya que se supone que el tamaño de sector de disco es de 512 bytes.

La función auxiliar `leersector` lee un sector de la unidad 0 (la primera unidad de disquetes) haciendo uso del servicio 13h de la BIOS. Esta función recibe como parámetros la dirección de memoria donde copiar el sector leído y los números de pista, cabeza y sector de disco donde se encuentra el mismo.

3.3. Salto del cargador al sistema operativo

Una vez cargado el sistema operativo en memoria es necesario cederle el control de la CPU. Esto se puede ver en el programa `cargador.c` a partir de la línea 100.

El sistema operativo es un programa COM. En estos programas, cuando se ejecutan, todos los registros de segmento (CS, DS, ES y SS) tienen el mismo valor. En nuestro programa se utiliza la instrucción `les` para almacenar en el registro ES el valor 0x50 y en el registro BX el valor 0x100. Estos dos valores se toman de la variable `direccionSO`. El valor del registro ES se copia al resto de registros de segmento mediante el registro AX.

Además de los registros de segmento hay que inicializar el puntero de pila, SP, del programa. En nuestro caso asignamos a dicho registro el valor 0xFB00 lo que significa que la cima de la pila se va a encontrar inicialmente en la dirección 0x10000 de memoria. El sistema operativo va a ocupar, pues, 64256 bytes de memoria.

El último paso es saltar a la dirección 50:100 de memoria, donde comienza el código del sistema operativo. Este salto se puede hacer de varias maneras. En nuestro caso hemos optado por apilar el valor del segmento (`push ax`) y el del desplazamiento dentro del segmento (`push bx`), y efectuar un «retorno lejano» (`retf`, *return far*) para desapilar los valores de CS e IP y así poder saltar a la dirección deseada.

3.4. El fichero `c0t_car.asm`

Ya que nuestro cargador es básicamente un programa en C que utiliza funciones y, por tanto, la pila, necesitamos crear un entorno adecuado para su ejecución. Además, debemos asegurarnos de que no utiliza ningún servicio externo (como los que habitualmente proporciona un sistema operativo) y de que utiliza direcciones de memoria acordes con la dirección de memoria en la que se cargará.

Para solucionar todos estos aspectos es necesario crear un fichero `c0t` adecuado para el compilador y que hemos llamado `c0t_car.asm`. El contenido de este fichero se muestra a continuación:

```

model tiny
.code
  EXTRN _main:NEAR
.startup
  org 7C3Eh
; Establecemos un entorno adecuado para la ejecución del cargador
  mov ax, cs
  mov ds, ax
  mov es, ax
  cli
  mov ss, ax
  mov sp, 8000h
  sti
; Llamamos a la función main del programa en C
  call _main
END
; $Id: c0t_car.asm 987 2005-04-20 13:52:07Z piernas $

```

Lo primero que podemos observar es que no se utiliza ningún servicio externo solicitado mediante la instrucción `int` (que produce una interrupción software).

Lo siguiente que podemos destacar es la línea `org 7C3Eh`. Esta línea le dice al programa ensamblador que tome como dirección de inicio del programa `7C3Eh`, en lugar de la dirección `0x100` que se toma por defecto para los programas COM.

La dirección `0x7C3E` se corresponde con la dirección que ocupará en memoria el programa cargador una vez que se cargue en memoria el sector de arranque. Como hemos indicado al principio de esta sección, el sector de arranque se carga en la dirección de memoria `0:7C00`. Dentro del sector de arranque de un sistema de ficheros FAT12 el programa cargador ocupa el rango de bytes `0x3E-0x1BD`, por lo que en memoria ocupará la posición `0x7C00+0x3E=0x7C3E`.

Aunque el programa cargador ocupa el rango de bytes `0x3E-0x1BD`, en realidad la primera instrucción que se ejecuta es la almacenada en los tres primeros bytes del sector de arranque. Estos bytes codifican un salto relativo a la posición `0x3E`, de ahí que nosotros consideremos, a efectos prácticos, que el código del cargador se encuentra en dicho rango de bytes.

Lo último que se hace en el fichero `c0t_car.asm`, antes de saltar a la función `main` del programa en C, es preparar el entorno adecuado para la ejecución de cargador, asignando a ciertos registros del procesador valores adecuados.

Ya que el programa cargador es un ejecutable COM, lo que hay que hacer es asignar a los cuatro registros de segmento el mismo valor, asignando a los registros DS, ES y SS el valor de CS. También se asigna al puntero de pila un valor adecuado (`mov sp, 8000h`). En nuestro caso, hemos reservado para la pila 512 bytes tras la

zona de memoria ocupada por el sector de arranque. Este tamaño es suficiente tanto para ejecutar el programa cargador como para ejecutar los servicios de la BIOS que va a necesitar dicho programa.

4. El programa `mkboot32`

La función del programa `mkboot32.c` es la de copiar el contenido del fichero ejecutable que se le pasa como parámetro en el sector de arranque de un disquete. Este programa utiliza las funciones del API Win32 y se puede compilar con cualquier compilador que soporte dicha API (como Visual C/C++). El código fuente de este programa es el que se muestra a continuación:

```

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

struct boot_sector {
    char salto[3]; /* 00h */
    char identificacion[8]; /* 03h */
    short bytes_por_sector; /* 0Bh */
    char sectores_por_cluster; /* 0Dh */
    short sectores_reservados; /* 0Eh */
    char copias_fat; /* 10h */
    short entradas_raiz; /* 11h */
    short total_sectores; /* 13h */
    char formato_disco; /* 15h */
    short sectores_por_fat; /* 16h */
    short sectores_por_pista; /* 18h */
    short cabezas; /* 1Ah */
    short sectores_ocultos; /* 1Ch */
    short pad;
    int total_sectores_long; /* 20h */
    char unidad_hd; /* 24h */
    char reservado; /* 25h */
    char marca; /* 26h */
    int numero_serie; /* 27h */
    char etiqueta[11]; /* 2Bh */
    char reservado_dos[8]; /* 36h */
    char cargador[0x1BE-0x3E]; /* 3Eh */
    char particiones[512-0x1BE]; /* 1BEh */
};

int leerbootsector(LPVOID buffer) {
    HANDLE hDisquete;
    DWORD leidos;

    hDisquete = CreateFile("\\\\.\\A:", GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        0, NULL);
    if (hDisquete == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "ERROR: imposible acceder al disquete\n");
        return 1;
    }

    if (!ReadFile(hDisquete, buffer, 512, &leidos, NULL) || leidos != 512) {
        fprintf(stderr, "ERROR: fallo la lectura del sector de arranque\n");
        CloseHandle(hDisquete);
        return 1;
    }

    CloseHandle(hDisquete);
    return 0;
}

```

```

int escribirbootsector(LPVOID buffer) {
    HANDLE hDisquete;
    DWORD escritos;
    55

    hDisquete = CreateFile("\\\\.\\A:", GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL);
    if (hDisquete == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "ERROR: imposible acceder al disquete\n");
        60
        return 1;
    }

    if (!WriteFile(hDisquete, buffer, 512, &escritos, NULL) || escritos != 512) {
        fprintf(stderr, "ERROR: fallo la lectura del sector de arranque\n");
        65
        CloseHandle(hDisquete);
        return 1;
    }

    CloseHandle(hDisquete);
    70
    return 0;
}

void main(int argc, char *argv[]) {
    75
    struct boot_sector sector;
    FILE * fichero;
    int total;
    char etiqueta[12];

    if (argc != 2) {
        80
        fprintf(stderr, "Uso: %s cargador.com\n", argv[0]);
        exit(1);
    }

    if (leerbootsector((char *)&sector)) {
        85
        fprintf(stderr, "ERROR: leyendo el sector de arranque\n");
        exit(1);
    }

    printf("Instruccion de salto: %x%x%x\n", sector.salto[0] & 0xFF,
        sector.salto[1] & 0xFF, sector.salto[2] & 0xFF);
    printf("Bytes por sector: %d\n", sector.bytes_por_sector);
    printf("Sectorres por cluster: %d\n", sector.sectores_por_cluster);
    printf("Sectorres por pista: %d\n", sector.sectores_por_pista);
    printf("Cabezas: %d\n", sector.cabezas);
    95
    strcpy(etiqueta, sector.etiqueta, 11);
    etiqueta[11] = '\0';
    printf("Etiqueta: %s\n", etiqueta);
    printf("Sectorres reservados: %d\n", sector.sectores_reservados);
    printf("Copias de la FAT: %d\n", sector.copias_fat);
    100
    printf("Maximo de entradas en directorio raiz: %d\n", sector.entradas_raiz);
    printf("Total sectorres: %d\n", sector.total_sectorres);
    printf("Formato del disco: %x\n", sector.formato_disco & 0xFF);
    printf("Sectorres por FAT: %d\n", sector.sectores_por_fat);
    printf("Sectorres ocultos: %d\n", sector.sectores_ocultos);
    105
    printf("Total sectorres 2: %ld\n", sector.total_sectorres_long);

    fichero = fopen(argv[1], "rb");
    if (!fichero) {
        110
        fprintf(stderr, "ERROR: abriendo el fichero imagen\n");
        exit(1);
    }

    printf("Copiando imagen%s al sector de arranque... \n", argv[1]);
    if (fseek(fichero, 0x7B3E, SEEK_SET)) {
        115
        fprintf(stderr, "ERROR: posicionamiento en el fichero\n");
        exit(1);
    }
}

```

```

total = fread(sector.cargador, 1, 0x1BE-0x3E, fichero);
if (ferror(fichero)) {
    fprintf(stderr, "ERROR: leyendo el codigo del cargador\n");
    exit(1);
}
if (fread(sector.cargador, 1, 1, fichero)) {
    fprintf(stderr, "ATENCION: el cargador es demasiado grande (>%d)\n", 0x1BE-0x3E);
    exit(1);
}
printf("Copiados %d bytes\n", total);

if (fclose(fichero)) {
    fprintf(stderr, "ERROR: cerrando fichero\n");
    exit(1);
}
if (escribirbootsector((char *)&sector)) {
    fprintf(stderr, "ERROR: escribiendo el sector de arranque\n");
    exit(1);
}
}

/* $Id: mkboot32.c 987 2005-04-20 13:52:07Z piernas $ */

```

El programa supone que el disquete tiene un sistema de ficheros FAT12, por lo que el sector de arranque tiene la estructura indicada por `struct boot_sector`.

Básicamente, lo que hace el programa es lo siguiente:

- Lee el sector de arranque del disquete que hay en la unidad mediante la función `leerbootsector`.
- A título informativo, muestra el contenido de distintos campos del sector de arranque. Esto es útil para comprender mejor el funcionamiento de un sistema de ficheros FAT.
- A continuación copia hasta 384 bytes del fichero ejecutable especificado como parámetro. Si el programa cargador es mayor que 384 bytes, muestra un mensaje de error y el programa termina.
- Si todo ha ido bien, escribe el nuevo sector de arranque en el disquete con la función `escribirbootsector` y termina.

La única peculiaridad del programa que merece la pena destacar es que el programa supone que el código del cargador se encuentra en la posición `0x7B3E` del fichero pasado como parámetro, y no en la posición `0` del mismo. Esto se debe a que el compilador Turbo C de Borland crea una cabecera llena de `0` en un fichero ejecutable para el que se ha usado la directiva `org`, como es nuestro caso con el programa cargador (vea la sección 3.4). Esta cabecera tiene un tamaño igual a «`X-0x100`», siendo «`X`» el valor especificado en la directiva `org`.

5. El programa `mkboot`

El programa `mkboot.c` es funcionalmente equivalente al programa anterior `mkboot32.c` y tiene una estructura muy similar. Si analizamos el código fuente, las dos principales diferencias que podremos observar entre estos dos programas son:

- `mkboot.c` es un programa para MS-DOS que hace uso de los servicios de la BIOS para leer/escribir sectores de disco mientras que `mkboot32.c` hace uso de funciones del API Win32 para hacer lo mismo.
- `mkboot.c` se compila también con la versión 3.0 del compilador Turbo C de Borland, al igual que el programa cargador, mientras que `mkboot32.c` necesita un compilador, como Visual C/C++, que construya ejecutables para Windows.

A continuación se muestra el código fuente de mkboot.c:

```

#include <stdio.h>

struct boot_sector
{
    char    salto[3];                /* 00h */
    char    identificacion[8];      /* 03h */
    int     bytes_por_sector;      /* 0Bh */
    char    sectores_por_cluster;  /* 0Dh */
    int     sectores_reservados;    /* 0Eh */
    char    copias_fat;            /* 10h */
    int     entradas_raiz;         /* 11h */
    int     total_sectores;        /* 13h */
    char    formato_disco;         /* 15h */
    int     sectores_por_fat;      /* 16h */
    int     sectores_por_pista;    /* 18h */
    int     cabezas;              /* 1Ah */
    int     sectores_ocultos;      /* 1Ch */
    int     pad;
    long    total_sectores_long;   /* 20h */
    char    unidad_hd;            /* 24h */
    char    reservado;            /* 25h */
    char    marca;                /* 26h */
    long    numero_serie;         /* 27h */
    char    etiqueta[11];         /* 2Bh */
    char    reservado_dos[8];      /* 36h */
    char    cargador[0x1BE-0x3E]; /* 3Eh */
    char    particiones[512-0x1BE]; /* 1BEh */
};

/*****
 * Funcion: leerbootsector
 * Transfiere el sector de arranque de un disquete a una
 * zona de memoria.
 *****/
unsigned int leerbootsector (unsigned char far *buffer)
{
    int i;

    for(i = 0; i < 3; i++)
    {
        asm {
            les    bx, dword ptr buffer
            mov    al, 1 /* 1 sector */
            mov    ch, 0 /* Pista 0 */
            mov    cl, 1 /* Sector 1 */
            mov    dh, 0 /* Cabeza 0 */
            mov    dl, 0 /* Unidad 0 */
            mov    ah, 02 /* De disco a memoria. */
            int    13h
            xor    ax, ax /* devolvemos 0 indicando que la operacion tuvo exito. */
            jnc    noError /* Si no se produce error salimos del bucle. */
            dec    ax /* devolvemos -1 indicando que la operacion fallo. */
        }
    }
noError:
}

/*****
 * Funcion: escribirbootsector()
 * Escribe un area de memoria en el sector de arranque
 * de un disquete.
 *****/

```

```

unsigned int escribirbootsector (unsigned char far *buffer)                                65
{
    int i;

    for(i = 0; i < 3; i++)                                                                70
    {
        asm {
            les    bx, dword ptr buffer
            mov    al , 1
            mov    ch , 0
            mov    cl , 1                                                                75
            mov    dh , 0
            mov    dl , 0
            mov    ah , 03    /* De memoria a disco. */
            int    13h
            xor    ax , ax    /* devolvemos 0 indicando que la operacion tuvo exito. */
            jnc    noError    /* Si no se produce error salimos del bucle. */
            dec    ax    /* devolvemos -1 indicando que la operaci3n fallo. */
        }
    }
    noError:
}                                                                                          85

int main(int argc, char *argv[])
{
    struct boot_sector sector;
    FILE * fichero;
    int total;

    if (argc != 2)
    {
        fprintf(stderr, "Uso: %s cargador.com\n", argv[0]);
        return 1;
    }
    if (leerbootsector((char *)&sector))
    {
        fprintf(stderr, "ERROR: leyendo el sector de arranque\n");
        return 1;
    }
    printf("Instruccion de salto: %x%x%x\n", sector.salto[0] & 0xFF, sector.salto[1] & 0xFF, sector.salto[2] & 0xFF);
    printf("Bytes por sector: %d\n", sector.bytes_por_sector);
    printf("Sectores por cluster: %d\n", sector.sectores_por_cluster);
    printf("Sectores por pista: %d\n", sector.sectores_por_pista);
    printf("Cabezas: %d\n", sector.cabezas);
    sector.etiqueta[10]=' \0';
    printf("Etiqueta: %s\n", sector.etiqueta);
    printf("Sectores reservados: %d\n", sector.sectores_reservados);
    printf("Copias de la FAT: %d\n", sector.copias_fat);
    printf("Maximo de entradas en directorio raiz: %d\n", sector.entradas_raiz);
    printf("Total sectores: %d\n", sector.total_sectores);
    printf("Formato del disco: %x\n", sector.formato_disco & 0xFF);
    printf("Sectores por FAT: %d\n", sector.sectores_por_fat);
    printf("Sectores ocultos: %d\n", sector.sectores_ocultos);
    printf("Total sectores 2: %ld\n", sector.total_sectores_long);
    fichero = fopen(argv[1], "rb");
    if (!fichero)
    {
        fprintf(stderr, "ERROR: abriendo el fichero imagen\n");
        return 1;
    }

    printf("Copiando imagen%s al sector de arranque... \n", argv[1]);
    /* Aunque solo manejamos disquetes, tenemos en cuenta la
}                                                                                          130

```

```

    * posible tabla de particiones que empieza en 0x1BE.
    */
    if (fseek(fichero, 0x7B3E, SEEK_SET))
    {
        fprintf(stderr, "ERROR: posicionamiento en el fichero\n");
        return 1;
    }
    total = fread(sector.cargador, 1, 0x1BE-0x3E, fichero);
    if (ferror(fichero))
    {
        fprintf(stderr, "ERROR: leyendo el codigo del cargador\n");
        return 1;
    }
    printf("Copiados %d bytes\n", total);
    if (fread(sector.cargador, 1, 1, fichero))
    {
        fprintf(stderr, "ATENCION: el cargador es demasiado grande (>%d)\n", 0x1BE-0x3E);
        return 1;
    }
    if (fclose(fichero))
    {
        fprintf(stderr, "ERROR: cerrando fichero\n");
        return 1;
    }
    if (escribirbootsector((char *)&sector))
    {
        fprintf(stderr, "ERROR: escribiendo el sector de arranque\n");
        return 1;
    }
}

```

135
140
145
150
155
160

/* \$Id: mkboot.c 1004 2005-04-27 12:27:07Z piernas \$ */
