ARQUITECTURA DEL ORDENADOR
MI_FDISK

Cristian Andrione

1 de Julio de 2013

1 Temas Investigados:

1. Organización de la información en el disco duro.
2. Direccionamiento de sectores en el disco: CHS y LBA.
3. Master Boot Record. Partes.
4. Tabla de particiones.
5. Descriptores de Particion. Estructura interna.
6. Sistema de archivos Unix. Dispositivos.
7. Arquitecturas little o big-endian.
8. Llamadas al sistema UNIX.

2 Herramientas Utilizadas:

Comando **dd** : Este comando copia y convierte datos entre archivos a bajo nivel. Algunos ejemplos útiles:

```
dd if=/dev/cdrom of=imagenCD.iso      /*hace una imagen ISO de un CD*/
```

```
dd if=/dev/sda1 of=MBR.img bs=512 count=1      /*guarda el MBR de /dev/sda1 en el archivo MBR.img*/
```

```
dd if=/dev/urandom of=/dev/hda      /*sobreescribe todo el disco con contenido aleatorio.*/
```

Comando **hexdump** o **hd**: volcado ASCII, decimal, hexadecimal y octal de un archivo. Ejemplo:

`hd -c MBR.img` /*muestra en formato canónico hex+ASCII el contenido de MBR.img obtenido con el comando **dd***/

Podemos crear un dispositivo muy elemental (que solo contará con un sector de arranque) utilizando los comando anteriores de la siguiente manera:

```
echo -ne "\x55\xaa"  | dd seek=510 bs=1 of=disco.img
```

qemu: es un emulador y virtualizador generico de CPU. Nos permite, entre infinidad de cosas, crear imágenes de dispositivos: con la siguiente línea creamos la imagen de un disco duro de 32Mb de capacidad.

```
qemu-img create -f raw disco.img 32M
```

`qemu-system-x86_64 -hda disco.img` /*Monta la imagen disco.img en el dispositivo virtual hda e intenta bootear desde allí en una máquina virtual*/

3 Programa:

El programa que se presenta, llamado `mi_fdisk`, tiene por objeto la manipulación de la tabla de particiones de un dispositivo, como podría ser un disco duro. El corazón de dicho programa, escrito en `c`, lo conforman las llamadas al sistema `lseek`, `read` y `write`. Ellas hacen todo el trabajo, lo demás son comprobaciones, manejo de errores e interface de usuario.

El primer sector `(512 bytes)` de un disco duro está reservado, no forma parte de ninguna partición. Este sector almacena información valiosa sobre la estructura interna del dispositivo y lo denominamos `Master Boot Record (MBR)`. Éste se extiende desde el inicio hasta la dirección `0x0200`. Ya dentro del `MBR` focalizamos la mirada en la tabla de particiones; que nos da toda la información necesaria sobre cantidad, tipo, ubicación y tamaño de las particiones presentes.
La tabla de particiones comienza en la dirección `0x01be` y comprende cuatro descriptores de partición cuyos tamaños son `16 bytes` cada uno y son consecutivos, lo que nos facilita escribir el código para movernos por sus partes. Cada descriptor de partición se compone de la siguiente manera: (los números indican el despazamiento en `bytes` a partir del inicio del descriptor):

- `(0)` indica partición activa.
- `(1 a 3)` dirección `CHS` inicio de partición.
- `(4)` tipo de partición.
- `(5 a 7)` dirección `CHS` de fin de partición.
- `(8 a 11)` inicio de partición `LBA`.
- `(12 a 15)` tamaño de partición en sectores.

Este programa no utiliza los datos en formato `CHS` sino los que están en formato `LBA` que reultan más sencillo de usar.
Entonces con la definición de algunas macros hacemos fácil de entender los movimientos:

```
#define SKIP_PARTITION_DESCRIPTOR   0x01be
#define SKIP_BOOT_INDICATOR         0x01be
#define SKIP_PARTITION_TYPE         0x01c2
#define SKIP_PARTITION_START        0x01c6
#define SKIP_PARTITION_SIZE         0x01ca
#define SKIP_VALIDITY_CHECK         0x01fe
#define OFFSET                      0x0010
```

Como ejemplo veamos la función que lee el tipo de partición aceptando como argumento el número de partición:

```
unsigned char partition_number(unsigned n)
{
    if(lseek(fd, SKIP_PARTITION_TYPE + n*OFFSET, SEEK_SET)
                          != SKIP_PARTITION_TYPE + n*OFFSET){
        printf("\nError lseek PARTITION TYPE\n");
        exit(-1);
    }

    if(read(fd, &buf, sizeof(buf)) <= 0){
        printf("\nError read PARTITION TYPE\n");
        exit(-1);
    }

    return pbuf[0];
}
```

`fd` es el descriptor de archivo que nos ha devuelto la llamada `open` y `pbuf` es una variable global definida de la siguiente manera:

```
int fd, buf;
unsigned char *pbuf = (unsigned char *) &buf;
```

Estas declaraciones globales responden a la necesidad conservar sus valores entre llamadas a funciones.

Obsevemos que las macros cuyos nombres comienzan con `SKIP_` hacen el salto a algunas de las partes del primer descriptor de particiones, de acuerdo a su nombre; y el segundo término de la suma `n*OFFSET` selecciona el descriptor de partición. Ejemplo:

Si reemplazamos las macros

```
SKIP_PARTITION_TYPE + n*OFFSET
```

por sus valores obtenemos:

```
0x01c2 + n*0x0010
```

que dependiendo de `n` pueden ser:

```
si n=0:  0x01c2
si n=1:  0x01d2
si n=2:  0x01e2
si n=2:  0x01fe
```

Y estas son las dirrecciones de la zona de cada descriptor de partición donde se guarda un núnero hexadecimal que representa al tipo de partición o sistema de archivos. La función debuelve `pbuf[0]` que es el byte que nos interesa. Debo hacer esto porque `buf` es `int` y para mis propósitos sólo es significativo el primer byte ¿cómo lo obtengo? apuntando a `pbuf` al primer `byte` de `buf` (Aquí debemos saber que `endian` es nuestra máquina). Recordemos que *un arreglo es un puntero* y *un puntero es un arreglo*

Para aclarar un poquito lo anterior analicemos el siguiente código, que no pertenece a nuestro programa pero es un buen ejemplo de lo que queremos explicar:

```
#include <stdio.h>
int main()
{
    int j, i = 0x12345678;
    unsigned char *pi = (unsugned char *) &i;

    for(j = 0; j < 4; j++){
        printf("pi[%d] = 0x%x\n", j, pi[j]);
    }
    return 0;
}
```

La salida de este programa en una máquina `litle endian` es:

```
pi[0] = 0x78
pi[1] = 0x56
pi[2] = 0x34
pi[3] = 0x12
```

Un `byte` está formado por 8 `bits` y puede contener un número hexadecimal entre `0x00` y `0xff`.

Entonces:

```
i =    pi[3]pi[2]pi[1]pi[0]
             ↑
i = 0x 12   34   56   78
```

Vemos que si usamos al puntero como un arreglo podemos acceder a los `bytes` de un `int` en forma independiente.
Tengamos en cuenta que mientras la variable `i` es `int`, a `pi` lo declaramos como

`unsigned char`. Lo cual es coherente ya que

```
sizeof(int) = 4
```
y
```
sizeof(char) = 1
```

Sigamos con otro ejemplo, este sí es de nuestro programa: la siguiente función devuelve `int` y el buffer `buf` es `int` así que no hay que hilar tan fino como antes con los tipos de datos y no merece tanta explicación en este sentido: Leemos con `read`, le decimos que guarde lo leído en `buf` y devolvemos su valor que corresponde con el número de sector dónde comienza la partición.

```
int partition_start(unsigned s)
{

    if(lseek(fd, SKIP_PARTITION_START + s*OFFSET, SEEK_SET)
                          != SKIP_PARTITION_START + s*OFFSET){
        printf("\nError lseek PARTITION START\n");
        exit(-1);
    }
    if(read(fd, &buf, sizeof(buf)) <= 0){
        printf("\nError read PARTITION START\n");
        exit(-1);
    }
    return buf;
}
```

Entonces, como decíamos, con `lseek` nos movemos por la tabla de particiones y con `read y write` vamos leyendo y modificando. Consta también, este programa, de un menú que hace las veces de una sencilla *interface de usuario* y nos guía a través de una serie de opciones similares a las del `fdisk` original.
El desarrollo del programa fue un diálogo constante con `fdisk`. Los resultados obtenidos en cada modificación se iban cotejando con los resultados de sus operaciones homólogas ejecutadas con `fdisk`.
Hubo todo tipo de problemas que llevaron a un repaso, profundización y mejor comprensión de las cuestiones relacionadas con *tipos de datos, punteros y las posibilidades del lenguaje* `c` a bajo nivel.

4 Ejemplo de uso:

Creamos una imagem de disco: `qemu-img create -f raw disco.img 16M`

Cargamos el disp. en nuestro programa: `./mi_fdisk_improved disco.img`

Después modificamos a gusto la tabla de particiones. Con `fdisk` cotejamos resultados: `fdisk disco.img`