SISTEMAS OPERATIVOS AVANZADOS

Guille Rodríguez González | LS26151

ÍNDICE

REQUISITOS

En esta práctica se nos pide que implementemos un programa que permita leer la información de particiones EXT2 y FAT16. Para ello haremos uso del lenguaje C, no habiendo ninguna restricción a nivel de uso de comandos y funciones en C para la manipulación de ficheros binarios.

El objetivo principal es aprender a manipular los datos según las especificaciones de cada sistema de ficheros a partir de una programación C estándar (no permitiendo el uso de la función ***system***, funciones como ***mount*** y tampoco librerías no estándar que faciliten el acceso a diferentes sistemas de ficheros).

La compilación deberá hacerse mediante el comando ***make***, por lo que deberemos tener un ***makefile*** y la estructuración del programa deberá ser modular, evitando así tener un único fichero de código.

Para la **primera fase**, se debe implementar la operación o parámetro "***/info***". El programa deberá abrir el fichero de sistema de archivos, detectar el tipo de sistema de archivos, extraer la información básica y mostrarla por pantalla.

Solo habrá que extraer la información cuando los sistemas de archivos detectados sean FAT16 o EXT2.

A continuación deberemos explicar como convertir (o crear uno similar) un sistema de ficheros EXT2 a FAT16 (para esta fase podremos hacer uso de herramientas de consola de Linux).

Por último debemos implementar un pequeño control de errores, que evalúe si los parámetros son correctos, si el fichero de sistema de archivos existe y que la operación pasada sea correcta.

Para la **segunda fase**, debemos realizar la búsqueda de un archivo dentro del sistema de ficheros e informar de su tamaño. Si no lo encontramos, debemos indicar que no se encuentra el fichero.

FASE 1 (/info)

DISEÑO

Para esta primera fase, he querido diseñar la práctica de forma sencilla y que se pueda entender a simple vista. He estructurado el código en 3 ficheros.

copycat.c

Fichero que incluye la función main.

func.c

Fichero de código que contiene todas las funciones que utiliza el programa.

func.h

Fichero *header* que contiene las estructuras y las definiciones de las funciones.

Makefile

Fichero que indica como compilar el programa haciendo uso del comando make.

Respecto al diseño de funciones realizado, he creído que la mejor forma sería hacer pequeñas funciones que se encarguen de una única tarea.

execMode

Esta función se encarga de detectar la operación o parámetro con el que se ha llamado al programa.

loadPartitionInfo

La función se encarga de intentar abrir el fichero (en modo lectura) que nos han pasado por parámetro y pasárselo a las funciones de detección de tipo de partición, carga de información básica y mostrar información del sistema de ficheros. Si el tipo de partición no es válido, muestra un mensaje de error. Por último se encarga de cerrar el fichero.

checkPartType

La función se encarga de detectar el tipo de partición que nos han pasado, si EXT2 o FAT16

loadExtInfo

Carga en una variable ***struct*** del tipo Ext la información básica sobre el sistema de archivos que nos han pasado al llamar al programa.

printExt2Info

Muestra por pantalla la información sobre el sistema de archivos EXT2.

loadFatInfo

Carga en una variable struct del tipo Vfat la información básica sobre el sistema de archivos que nos han pasado al llamar al programa.

printFatInfo

Muestra por pantalla la información sobre el sistema de archivos FAT16

Por último existen varias funciones de pintar por pantalla para facilitar la codificación

printMsg, printlnMsg, printErrNoParams, printErrFileOpen, printErrUnknowPartition …

Y una función llamada ***strToLower*** que pasa una string a letras minúsculas, haciendo posible que el usuario ponga la operación tanto en mayúsculas como en minúsculas.

ESTRUCTURAS DE DATOS

Para tener mayor comodidad de organización y poder agrupar la información de un sistema (EXT o FAT) se usan las estructuras que nos permite ANSI C.

Estructura FAT

Para guardar y mostrar la información de la partición FAT se ha usado la siguiente estructura:

```
typedef struct {
        char volume_type[8];
        char volume_name[8];
        char volume_label[12];
        unsigned char fat_tables;
        unsigned short int sectors_size;
        unsigned char sectors_cluster;
        unsigned short int sectors_reserved;
        unsigned short int sectors_maxrootentr;
        unsigned short int sectors_perfat;

} Vfat;
```

El motivo de que todos los valores donde se guardarán números sean ***unsigned*** se debe a que en la documentación indica que al no hacerlo con ***unsigned*** algunos cálculos podrían devolver valores erróneos.

Also take note that all data types are UNSIGNED. Do not do FAT computations with signed integer types, because the computations will be wrong on some FAT volumes

Volume_type (posición 54, 8 bytes)

Este campo guardara el tipo de FAT que nos hemos encontrado. Por defecto en la documentación indica que suele ser FAT12, FAT16, FAT32, pero este campo puede ser cambiado mediante un HEX Editor. Mediante la documentación podemos ver que se usan 8bytes para indicarlo (aunque luego solamente usemos 8 x 5 caracteres).

Volume_name (posición 3 , 8 bytes)

Este campo guardaremos el nombre del volumen, tal y como indica la documentación es un campo de texto y que contiene 8bytes.

Volume_label (posición 43 , 11 bytes)

Este campo guarda la etiqueta que se le ha puesto al volumen.

Fat_tables (posición 16 , 1 byte)

Este campo se ha designado ***unsigned char*** ya que según indica la documentación no hay muchas tablas FAT en una partición FAT. Por ello usaremos pocos bytes (8) para guardar el valor. Podríamos usar un ***short int***, pero este ocupa unos 16 bytes y sería malgastar memoria.

Sectors_size (posición 11, 2 bytes)

Campo que guarda el tamaño de bytes que ocupa cada sector. Mediante la documentación podemos ver que los valores son 512, 1024, 2048 y 4096, por ello usaremos un short int.

Sectors_cluster (posición 13 , 1 byte)

Campo que guarda el número de ***clusters*** por sector, al ser un campo que ocupa 1 byte (como el fat_tables).

Sectors_reserved (posición 14, 2 bytes)

Campo que guarda el número de sectores reservados, para volúmenes FAT12 y FAT16 este valor debe ser siempre 1.

Sectors_maxrootentr (posición 17, 2 bytes)

Campo que guarda el número de máximas entradas (directorios) que se pueden alojar en el directorio raíz.

Sectors_perfat (posición 16 , 1 byte)

Guarda el tamaño de una tabla FAT. Para FAT32 este campo estará a 0. Por lo que si detectamos que es una partición FAT, podemos mirar de esta forma si estamos en una FAT32 u otro tipo de FAT.

Estructura EXT

Para guardar y mostrar la información de la partición EXT se ha usado la siguiente estructura.

```
typedef struct {

        char volume_name[17];           // s_volume_name
        char volume_type[5];            // s_magic
        unsigned int inode_size;        // s_inode_size
        unsigned int inode_total;       // s_inodes_count
        unsigned int inode_first;       // s_first_ino
        unsigned int inode_group;       // s_inodes_per_group
        unsigned int inode_free;        // s_free_inodes_count
        unsigned int block_total;       // s_blocks_count
        unsigned int block_reserved;    // s_r_blocks_count
        unsigned int block_free;        // s_free_blocks_count
        unsigned int block_size;        // s_log_block_size
        unsigned int block_first;       // s_first_data_block
        unsigned int block_group;       // s_blocks_per_group
        unsigned int frags_group;       // s_frags_per_group
        time_t last_check;              // s_lastcheck
        time_t last_mount;              // s_mtime
        time_t last_write;              // s_wtime

} Ext;
```

Volume_name (posición 1144 , 16 bytes)

Guarda el nombre que tiene asignado el volumen

Volume_type

Guarda el tipo de volumen que es, (EXT2, EXT3, EXT4), aunque actualmente el programa solo diferencia si es FAT o EXT, ya lo dejamos preparado.

Inode_size (posición 1112 , 4 bytes)

Guarda el tamaño de un *inode*

Inode_total (posición 1024 , 4 bytes)

Guarda el número de *inodes* que tiene la partición

Inode_first (posición 1108 , 4 bytes)

Guarda el índice al primer inodo que se puede usar para guardar ficheros.

Inode_group (posición 1064 , 4 bytes)

Guarda el número de inodos que hay por group.

Inode_free (posición 1068 , 4 bytes)

Guarda el número de inodos que libres en la partición.

Block_total (posición 1032 , 4 bytes)

Guarda el número de blocks totales en la partición

Block_reserved (posición 1036 , 4 bytes)

Guarda el número de blocks reservados en la partición

Block_free (posición 1040, 4 bytes)

Guarda el número de blocks que hay libres en la partición

Block_size (posición 1052 , 4 bytes)

Guarda el tamaño en bytes de un block

Block_group (posición 1056 , 4 bytes)

Guarda el número de bloques hay en cada *group*

Frags_group (posición 1060 , 4 bytes)

Guarda el número de *fragments* que hay en cada *group*

Last_check (posición 1088, 4 bytes)

Tiempo (*timestamp unix*) de la última vez que se hizo una comprobación al sistema de archivos.

Last_mount (posición 1068 , 4 bytes)

Tiempo (*timestamp unix*) de la última vez que se montó el sistema de archivos

Last_write (posición 1072 , 4 bytes)

Tiempo (*timestamp unix*) de la última vez que se escribió en el sistema de archivos.

INODE ENTRY

Para guardar los datos de un *Inode* de forma cómoda, se ha creado una estructura donde guardar completamente el *Inode*.

```
typedef struct {
        unsigned long i_mode;
        unsigned long i_uid;
        unsigned long i_size;
        unsigned long i_atime;
        unsigned long i_ctime;
        unsigned long i_mtime;
        unsigned long i_dtime;
        unsigned long i_gid;
        unsigned long i_links_count;
        unsigned long i_blocks;
        unsigned long i_flags;
        unsigned long i_osd1;
        unsigned long i_block[15];
        unsigned long i_generation;
        unsigned long i_file_acl;
        unsigned long i_dir_acl;
        unsigned long i_faddr;
        unsigned long i_osd2;
} InodeEntry;
```

Existe una función que recibe el offset donde se está ese Inode, se encarga de leerlo y guardarlo.

EXT DIRECTORY

```
typedef struct {
        unsigned int inode;
        unsigned short int rec_len;
        unsigned char name_len;
        unsigned char file_type;
        char file_name[BUFFSIZE];
} ExtDirectory;
```

Tal y como pasa con la estructura para el *Inode*, también existe una estructura encargada de contener toda la información sobre una entrada de directorio. También existe la función que se encarga de cargar la información en la estructura. A continuación podemos ver también la de FAT.

FAT DIRECTORY

```
typedef struct {
        char name[12];
        char ext[4];
        unsigned char attr;
        unsigned short int reserved;
        unsigned short int time;
        unsigned short int date;
        unsigned short int firstblock;
        unsigned long size;

} FatDirectory;
```

CONVERSIÓN DE EXT2 A FAT16

Primero definiremos los campos importantes de cada tipo de partición e intentaremos realizar un símil.

EXT2

Inodo: Es una estructura de datos que contiene las características (permisos, fechas, ubicación) de ficheros, directorios o datos que pueda contener un sistema de ficheros. Viene a ser una "*tabla con los datos informativos + punteros a los datos guardados*".

Block: Un bloque es una agrupación de pequeños sectores. El tamaño de los bloques suele estar determinado cuando realizamos el formato del disco. Normalmente tienen un tamaño de 1KiB, 2KiB, 4KiB y 8KiB.

Block group: Los bloques son guardados en grupos de bloques para reducir la fragmentación y minimizar la cantidad de desplazamientos del cabezal del disco cuando se leen gran cantidad de datos consecutiva. Dos bloques cercanos al inicio del block ***group*** se usan para mapear cuales de los bloques e ***inodes*** están en uso.

Superblock: Contiene toda la información sobre la configuración del sistema de ficheros, indicando el número total de blocks, ***inodes*** que hay en el sistema de archivos, cuantos están libres, cuantos ***inodes*** hay en cada block, cuantos block hay en cada block ***group*** e información de cuando fue montado, comprobado y escrito el sistema de archivos.

FAT

Sector: Es una cantidad de bytes agrupada

Clúster: Numero de sectores por unidad de alojamiento

Lo primero es conseguir el número de bytes que tiene el sistema.

Numero de Block groups x Block size = XX Bytes

A continuación miraremos el tamaño de cada bloque e intentaremos igualar el Sector Size al Block Size. Los tamaños tal y como indica la documentación para cada sistema de archivos, son:

Fat (bytes) ➔ 512, 1024, 2048, 4096
Ext (bytes) ➔ 1024, 2048, 4096, 8192

Para la conversión de EXT a FAT el único caso donde no podemos igualar el Block Size con el Sector Size es cuando Block Size tiene un tamaño de 8Kib.

> Si el Block Size es igual a uno de los valores {1024, 2048, 4096} igualaremos el Sector Size.
> Si el Block Size supera 4096 podremos el valor de Sector Size al máximo (4096).

Ya que si tenemos un tamaño grande de block, lo ideal es mantener el tamaño de sector lo más parecido posible, puesto que la persona que dio formato al sistema debió considerar el motivo por el cual le daba ese tamaño de sector y por lo tanto obtener X rendimiento.

Una vez hemos obtenido el tamaño de los sectores, los agrupamos en clusters. En un clúster puede haber entre 1 y 128 sectores (siempre en potencias de dos). Por lo que una vez definido cuantos sectores queremos en un clúster (nSecPerCluster), realizaremos una división.

XX Bytes / Sector Size = Número de Sectores

Número de Sectores / nSecPerCluster = Número de clústeres

Si el número de clústers nos da un valor decimal, podemos redondearlo a la alza para evitar perder espacio.

Por último cuando creemos el fichero de sistema de archivos FAT, lo podemos hacer con los siguientes comandos desde un sistema Linux.

Creamos el archivo del tamaño indicado

```
dd if=/dev/zero of=~/myFat16.bin bs=(Sector Size) count=(Número de clústeres)
```

Damos formato al sistema de archivos indicándole mediante parámetros la configuración deseada.

```
mkfs.vfat myFat16.bin–F 16 –f 2 –R 1 –r 512 –s 1
```

Donde los parámetros afectan a:

f: Numero de FAT que queremos tener el sistema de archivos, la documentación indica que este valor debería ser 2 para cualquier volumen FAT, pero que un valor mayor a 1 podría ser correcto, el problema es que bastantes programas y algunos sistemas operativos podrían no funcionar correctamente si el valor es diferente a 2.

F: Tamaño de la FAT, si vamos a crear un sistema de FAT 16, asignamos 16.

R: Numero de sectores reservados, la documentación indica que para FAT16 ha de tener valor 1.

r: Número de entradas en el directorio raíz, para discos duros la documentación indica que ha de ser 512.

s: Indica el número de sectores por clúster que queremos asignar.

PRUEBAS REALIZADAS

Para comprobar el correcto funcionamiento del programa he hecho uso de las aplicaciones que se nos han facilitado en el Estudy y he revisado que la información de los campos que se solicitan en esta práctica sean idénticos.

Compilación mediante comando ***make***, con parámetros ***–Wall –Wextra***

```
ls26151@vela:~/SOA$ make
rm -f func.o copycat.o
gcc -Wall -Wextra -c copycat.c
gcc -Wall -Wextra -c func.c
gcc func.o copycat.o -o copycat
rm -f func.o copycat.o
ls26151@vela:~/SOA$
```

Resultados EXT2

```
ls26151@vela:~/SOA$ ./copycat /info Ext2_1024_Con-info.ex2
---- Fylesystem  Information ----

Fylesystem: EXT2

INODE INFO
        INODE SIZE:     128
        INODE TOTAL:    1024
        INODE FIRST:    11
        INODES GROUP:   1024
        INODES FREE:    968


BLOCK INFO
        BLOCK TOTAL:    4092
        BLOCKS RESVD:   204
        BLOCKS FREE:    966
        BLOCK SIZE:     1024
        BLOCK FIRST:    1
        BLOCK GROUP:    8192
        FRAGS GROUP:    8192


VOLUME INFO
        VOLUM NAME:     TEST1
        LAST CHECK:     Sat Apr  2 14:31:02 2011
        LAST MOUNT:     Sat Apr  2 14:42:36 2011
        LAST WRITE:     Sat Apr  2 14:57:31 2011
```

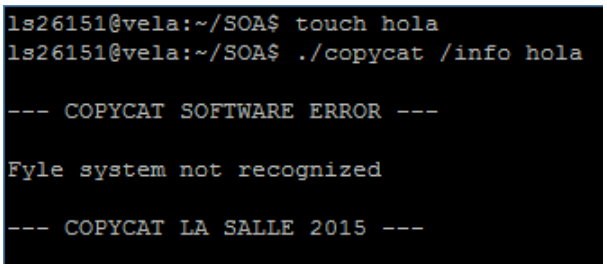
```
ls26151@vela:~/SOA$ ./copycat /info Ext2_1024_Con-Mucha-info.ex2
---- Fylesystem  Information ----

Fylesystem: EXT2

INODE INFO
        INODE SIZE:     128
        INODE TOTAL:    1024
        INODE FIRST:    11
        INODES GROUP:   1024
        INODES FREE:    0


BLOCK INFO
        BLOCK TOTAL:    4092
        BLOCKS RESVD:   204
        BLOCKS FREE:    3868
        BLOCK SIZE:     1024
        BLOCK FIRST:    1
        BLOCK GROUP:    8192
        FRAGS GROUP:    8192


VOLUME INFO
        VOLUM NAME:     TEST1
        LAST CHECK:     Sat Apr  2 14:31:02 2011
        LAST MOUNT:     Sat Apr  2 14:42:36 2011
        LAST WRITE:     Sat Mar 17 11:29:46 2012
```

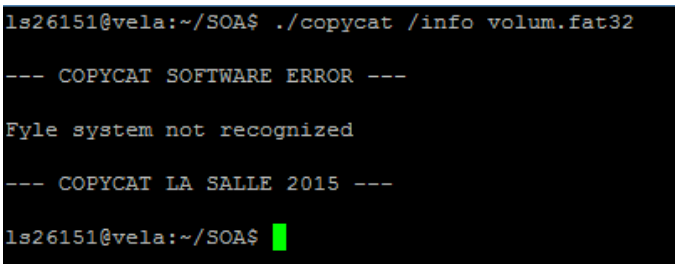
Resultados FAT16

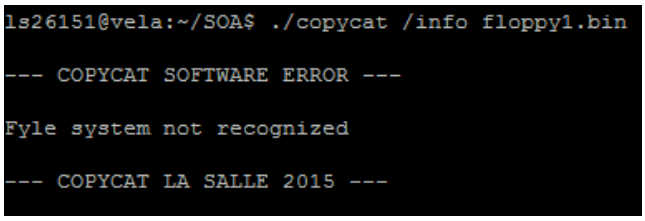
```
 ls26151@vela:~/SOA$ ./copycat /info fat16_1024.bin
---- Fylesystem  Information ----

Fylesystem: FAT16

VOLUME NAME:    mkdosfs
VOLUME LABEL:   TEST2
SECTOR SIZE:    1024
SECTORS per CLUSTER: 1
SECTORS RESERVED: 1
NUM FAT TABLES: 2
MAX ROOT ENTRIES: 512
SECTORS per FAT: 16
```

```
ls26151@vela:~/SOA$ ./copycat /info fat16_2048.bin
---- Fylesystem  Information ----

Fylesystem: FAT16

VOLUME NAME:    mkdosfs
VOLUME LABEL:   TEST2
SECTOR SIZE:    1024
SECTORS per CLUSTER: 2
SECTORS RESERVED: 1
NUM FAT TABLES: 2
MAX ROOT ENTRIES: 512
SECTORS per FAT: 10
```

Resultado de ficheros no validos







PROBLEMAS ENCONTRADOS

En la partición EXT me he encontrado con que al pintar el ***block_size*** daba un valor 2 cuando debería dar 4096 (usando la ext_4k). Para solventarlo, tal y como indica en la documentación se realiza una conversión.

```
ext2.block_size =  1024 << ext2.block_size;
```

También he encontrado problemas al obtener la fecha de ***last_mount*** ya que pensaba que la información se guardaba en el campo ***s_last_mounted***. No entendía porque motivo este campo usaba 64bytes y los ***s_lastcheck*** y ***s_wtime*** usaban tan solo 4bytes, además al guardarlo en un ***time_t*** obviamente daba un error de ***segmentation fault*** *(core dumped).*

Después de revisarlo detenidamente vi que en la documentación había un campo ***s_mtime*** (el cual yo pensaba que se refería a la última vez que se modificó el fichero, como el flag de los ficheros que hay en un sistema Linux).

FASE 2 (/find)

DISEÑO

Para poder facilitarme la tarea en las siguientes fases, he pensado que sería mejor crear las funciones de manera que las pueda reutilizar.

getFatStartRootDirectory

Esta función se encarga de obtener la posición donde comienza el ***Root Directory*** en una partición FAT16. Para ello hace uso de los datos cargados en un ***struct*** Vfat.

fatStrLen

Esta función se encarga de obtener cuantos caracteres ocupa realmente el nombre del fichero que hemos leído del sistema FAT, ya que la función ***strlen*** nos indica también como válidos los espacios en blanco, pero en un sistema FAT16, el nombre no puede contener espacios.

findInFat

Esta función es la encargada de buscar el fichero o directorio (a partir de ahora ***entrada***) en el ***Root Directory*** de FAT16 y de devolver la posición del inicio de la entrada para que desde otras funciones podamos ir directamente al fichero. El retorno indica la posición, pero recibe como parámetro por referencia una función donde guardaremos el tipo de entrada encontrada (solo si es un ***fichero o directorio valido!***).

findFileinVolume

Función encargada de realizar las llamadas a las otras funciones para poder mostrar la información cuando nos han indicado el parámetro ***/find***. Viene a ser como la función de la fase 1 ***loadPartitionInfo***, abre el fichero, detecta el tipo de partición y dependiendo de ello llama a unas funciones u otras. Se podría decir que es como un sub main.

strToUpper

Debido a que en las particiones FAT16 los nombres de los archivos y extensiones están en mayúsculas, debemos convertir el nombre del archivo que busca el usuario a mayúsculas para comparar correctamente.

printFatFileInfo

Encargada de imprimir por pantalla la información del tamaño del fichero.

Por la parte de las funciones de EXT se han creado las siguientes.

findInExt

Función que se encarga de iniciar la búsqueda del fichero o directorio pasando el ***Inode*** número 2.

getInodeTableSeek

Función encargada de encontrar la posición (***lseek***) del inicio de la tabla de ***Inode*** dentro del fichero de sistema de archivos.

loadDirectoryEntry

Función encargada de leer una entrada de ***Directory***, devuelve una estructura del tipo ExtDirectory con todos los datos cargados.

loadInodeData

Función encargada de leer la información de un ***Inode***, devuelve una estructura del tipo ***InodeEntry*** con todos los datos cargados.

getInodeSeekPosition

Función que recibe el número de ***Inode*** y nos devuelve la posición de este.

extSearchInSubfolder

Función encarga de realizar la búsqueda del fichero o directorio dentro de la estructura de datos de EXT. **Entra en cada subcarpeta** y busca por todo el árbol de directorios. Se llama a si misma recursivamente cuando encuentra un directorio.

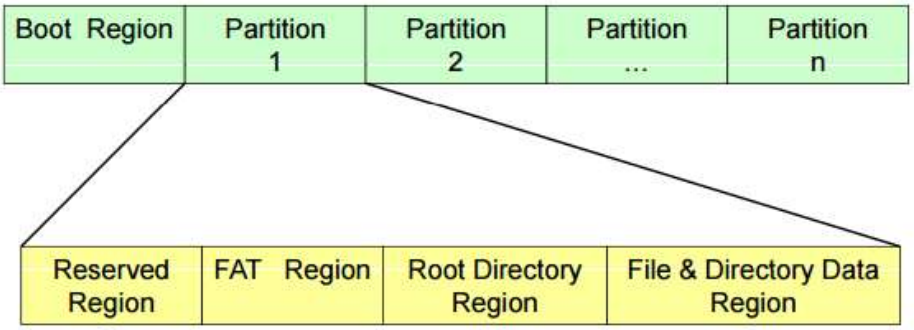
printExtFileInfo

Esta función recibe por parámetro el offset del Directory Entry asociado al archivo, lo lee y muestra su tamaño.

ENCONTRAR UN FICHERO

FAT

Para encontrar un fichero en FAT16, hago uso de algunas de las funciones indicadas arriba. Lo primero que he necesitado es calcular donde se encuentra el ***Root directory***, pero hay que tener en cuenta datos sacados en la fase 1.



Tal y como vemos en la imagen, para llegar a ***Root Directory Region*** debemos pasar la ***Reserved Region*** y la ***FAT Region***.

Para obtener el tamaño de la ***Reserved Region*** podemos sacarlo haciendo el siguiente cálculo.

Tamaño de Reserved Region = nº Sectores Reservados * Tamaño del sector

Para obtener el tamaño de la ***FAT Region*** hemos de tener en cuenta la siguiente formula

Tamaño de FAT Region = nº Tablas FAT * nº Sectores por FAT * Tamaño del sector

Y por último la ***Root Directory Region*** que es la que nos interesa, la calcularemos realizando la suma de las dos anteriores.

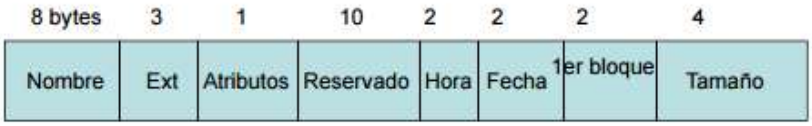
Inicio de Root Directory Region = Tamaño de Reserved Region + Tamaño FAT Region

En el código podremos ver que en la función ***getFatStartRootDirectory*** tenemos el cálculo indicado de la siguiente manera.

```
rootDirectory = ((fat.sectors_reserved * fat.sectors_size) + (fat.fat_tables * fat.sectors_perfat * fat.sectors_size));
```

Esta posición es devuelta por la función y utilizada por la función ***findInFat*** para leer las n entradas (***Vfat. sectors_maxrootentr***) que puede contener una partición FAT16. Recordemos que si es para disco duro son 512, pero para ***floppy*** es otro número de entradas.

Las entradas que hay en la ***Root Directory Region*** ocupan 32bytes, los cuales están organizados de la siguiente manera.



Por ello una vez sabemos el inicio de la Root Directory y sabiendo que cada entrada son 32bytes, tan solo debemos iterar pasando por todas las Root Entry revisando el nombre, la extension y por ultimo el campo atributos que nos indicara el tipo de entrada.

Bit	Mask	Attribute
0	0x01	Read-only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Unused
7	0x80	Unused

Pero el campo de atributos no podemos leerlo directamente, debemos calcular la mascara de bits y una vez la tengamos, podremos saber el tipo de entrada.

Para leer los atributos, posicionamos lseek y leemos 1 byte (***unsigned char***) en la variable atributes, por ultimo calculamos la bitmask y comparamos si coincide con el atributo que deben tener las entradas de fichero o subdirectorio

Archivo
(atributes & 0x20) == 0x20

Directorio
(atributes & 0x10) == 0x10

De esta forma aunque tengamos el mismo nombre de entrada que el nombre que nos ha pasado el usuario, si los atributos no son de tipo fichero o de tipo directorio, no mostraremos datos.

Para esta fase, como tan solo debemos obtener el tamaño del fichero, podriamos obviar todas las entradas y mostrar tan solo las entradas con el bit 5 (fichero) (si ademas el nombre y extension de la entrada coincide el nombre de fichero + extension introducido por el usuario).

Una vez hemos encontrado el fichero correcto, cargamos un numero en la variable filetype y devolvemos la posicion de inicio de esa entrada.

Por último procedemos a llamar a la funcion printFatFileInfo indicando como argumentos el ***file descriptor*** del fichero y la posicion que le pasaremos a ***lseek***.

Leemos los últimos 4 bytes de la entrada cargándolos en un ***unsigned int*** y a continuación mostramos el resultado.

```
ls26151@vela:~/SOA$ ./copycat /find Fat16_2048_Con-info.bin alloca.h

        File found! Size: 1289 bytes
```

Si el nombre (incluyendo extensión si tiene) y el atributo es válido, daremos por correcto el fichero encontrado y devolveremos la posición de ***lseek*** y el tipo de entrada (variable ***filetype***).

EXT

Para encontrar un fichero en EXT2 lo primero que he tenido que hallar es la ***Inodes Table***. Como indica en la documentación de EXT2, el ***Superblock*** siempre comienza desde la posición 1024 y ocupa 1024 bytes independientemente de lo que ocupe el ***Block Size***. Sin embargo para encontrar los siguientes bloques sí que hemos de tenerlo en cuenta.

Para obtenerlo podemos ver en la imagen que deberíamos movernos a la posición 1024 que es donde inicia el ***Superblock***, a esta posición, le sumamos el tamaño del ***Superblock*** (1024). En este punto tenemos el inicio del ***Group Descriptor***.

Pero para calcular el Group Descriptor nos encontramos con un problema, y es que dependiendo del Block Size, estará en el primer bloque o no. Si el Block Size es de 1KB encontraremos que el Group Descriptor está justo después del Superblock. Sin embargo si el Block Size es mayor a 1KB detrás del superblock tendremos padding hasta llegar al tamaño del Block Size. El campo Block First que encontramos en el superblock, nos indica si hemos de obtener la Group Descriptor Table de una forma u otra.

Por ello para calcularlo se ha usado el siguiente código.

```
if ( ext.block_first == 0 ) {
        GroupDescriptorTablePosition = ext.block_size;
} else {
        GroupDescriptorTablePosition = EXT_SUPERBLOCK_START_POS + EXT_SUPERBLOCK_SIZE;  (1024+1024)
}
```

Una vez ya tenemos la posición de *Group Descriptor*, podemos encontrar la ***Inote Table*** position leyendo el campo ***bg_inode_table*** de la *Group Descriptor*. Este campo nos indica el número de *Block* donde se encuentra alojada la *Inode Table*.

```
Inode Table = bg_inode_table * ext.block_size
```

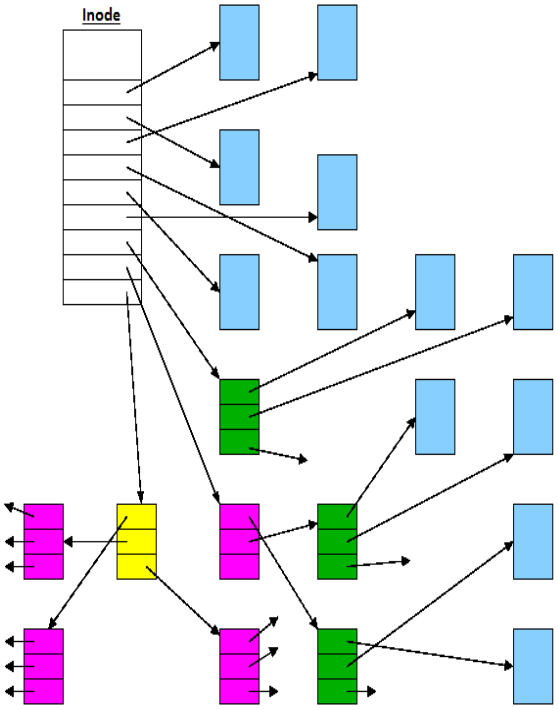
Cada Inode ocupa 128Bytes, y los primeros 11 Inodes suelen ser reservados. En el Inode número 2 nos encontramos la Root Directory.

```
Root Directory = Inode Table + Inode Size (128)
```

Offset (bytes)	Size (bytes)	Description
0	2	i_mode
2	2	i_uid
4	4	i_size
8	4	i_atime
12	4	i_ctime
16	4	i_mtime
20	4	i_dtime

Offset (bytes)	Size (bytes)	Description
24	2	i_gid
26	2	i_links_count
28	4	i_blocks
32	4	i_flags
36	4	i_osd1
40	15 x 4	i_block
100	4	i_generation
104	4	i_file_acl
108	4	i_dir_acl
112	4	i_faddr
116	12	i_osd2

En el Inode encontraremos los 12 punteros directos, 1 indirecto, 1 indirecto doble y 1 indirecto triple que nos indican el número de bloque de datos donde se encuentra la información. De nuevo para saber la posición de este bloque de datos dentro del fichero, debemos multiplicar el número por el ***Block Size***.



Comprobando bloque a bloque de los 12 punteros directos la información que vayamos encontrando, si encontramos que uno de los punteros directos (o indirectos) tiene un valor de 0, significa que ese puntero no apunta a ningún bloque y por lo tanto ya no tenemos que seguir buscando.

En el caso del puntero indirecto, el numero guardado nos indica este número nos indica el bloque donde hay más punteros a bloques de información, de manera que ese bloque es un bloque entero de punteros a bloques.

El indirecto doble, viene a ser lo mismo solo que se añade un bloque más de punteros.

En el caso del indirecto triple, es otro nivel más que en el indirecto doble. Con una imagen se puede entender mejor.

En la imagen se pueden ver los bloques de información (azules), el color verde indica los punteros indirectos simples, el color lila indica los punteros indirectos dobles y el color amarillo indica el puntero indirecto triple.

Ahora tan solo debemos leer las entradas existentes en la ***Root Directory*** y buscar dentro de las carpetas y subcarpetas, hasta encontrar una ***Directory Entry*** donde el nombre coincida.

El problema está en que no es tan sencillo, puesto que las ***Directory Entry*** a diferencia de FAT, tienen longitud variable.

Como hemos visto en la estructura, existe un campo ***rec_len*** que nos indica la longitud de la entrada de directorio. Por lo tanto sabiendo la longitud de cada entrada podemos ir moviéndonos entre ellas y revisar el nombre de la entrada.

Al igual que en FAT, estas entradas pueden ser o no válidas. Por lo que al leer la entrada comprobamos si es una entrada valida, de ser así comprobamos el nombre (campo ***file_name*** de la estructura de directorios).

Si el nombre coincide, es momento de comprobar si es un fichero o es un directorio. Para este fin, haremos uso del campo ***file_type*** de la estructura de datos de directorios.

Si el valor guardado es un 1, entonces lo que hemos encontrado es un fichero. Si el valor guardado es un 2, entonces lo que hemos encontrado es un directorio. Cualquier otro tipo de entrada no nos interesa.

Si el nombre no coincide, solo comprobaremos si la entrada es un directorio, ya que si es un directorio puede contener archivos dentro. Cualquier otro tipo de entrada no nos interesa, ya que puede que sea un fichero, pero el nombre no coincide por lo tanto no es el que estamos buscando.

Si encontramos que se trata de un directorio, entramos dentro. ¿Pero cómo? En este caso llamaremos a la función ***extSearchInSubfolder*** (precisamente en la que estamos) haciendo que se realicen llamadas recursivas, facilitando así la programación y búsqueda del archivo.
Si hemos llegado a buscar por todo el árbol de directorios (sabemos que esto pasa cuando ya no hay más ***Directory Entry***) y no hemos encontrado el fichero, devolveremos un mensaje de error ***File not Found***.

Si lo hemos encontrado, entonces devolveremos la posición (offset) de la ***Directory Entry***.

A continuación volveremos a leer esa ***Directory Entry***, si es un fichero miraremos el campo ***inode*** que nos indica el número de Inode donde se encuentra ese archivo. Nos moveremos al ***Inode*** (desde la ***Inodes Table***, sumaremos 128 n-1 veces donde ***n*** seria el número del ***Inode***, obteniendo así la posición de inicio de esa entrada de ***Inode***.

Leeremos el ***Inode***, y en su campo ***i_size***, tendremos guardada la longitud. Tan solo queda mostrarla por pantalla.

```
ls26151@vela:~/SOA$ ./copycat /find Ext2_1024_Con-Mucha-info.ex2 999

        File found in inode [1023]!
        Size 0 Bytes
```

En la partición Punteros indirectos se añadió texto al archivo 999 (montando la unidad).

```
ls26151@vela:~/SOA$ ./copycat /find Ext2_1024_PuntersIndirectes.ex2 999

        File found in inode [1012]!
        Size 29 Bytes
```

Como última cosa a añadir se ha realizado la búsqueda en todo el árbol de directorios, no solo en la ***Root Directory*** de EXT.

PRUEBAS REALIZADAS

Para realizar la comprobación se han seguido las mismas pruebas que hay en el enunciado de la práctica. A continuación se muestran las pruebas para los sistemas de archivos FAT16.

Buscamos un fichero que no exista

```
ls26151@vela:~/SOA$ ./copycat /find Fat16_2048_Con-info.bin pepe.dat

        Error: File not found.
```

Buscando en un sistema de archivos diferente a FAT16 y EXT2

```
ls26151@vela:~/SOA$ ./copycat /find vol32.bin pepe.dat

        Error: The volume is neither FAT16 or EXT2
```

Probando a pasar número incorrecto de parámetros

```
ls26151@vela:~/SOA$ ./copycat /find

Error: Incorrect parameters number
         > copycat.exe /find volume file
```

Por ejemplo en esta prueba, test2 es una entrada existente, pero es el ***volume_label*** y el atributo no indica que es un fichero o subdirectorio, por lo tanto al realizar la búsqueda, no debería encontrarlo aunque la entrada exista.

```
ls26151@vela:~/SOA$ ./copycat /find Fat16_2048_Con-info.bin test2

        Error: File not found.

ls26151@vela:~/SOA$ ./copycat /find Fat16_2048_Con-info.bin alloca.h

        File found! Size: 1289 bytes
```

Si buscamos un fichero existente y que es un archivo (***alloca.h***) deberá mostrar el tamaño del fichero. Sin embargo si buscamos un directorio, el programa deberá indicar que es un directorio.

```
ls26151@vela:~/SOA$ ./copycat /find Fat16_2048_Con-info.bin hdparm

        Is a directory!.
```

Para el sistema EXT se han realizado las pruebas siguientes.

Búsqueda de ficheros ocultos (no debe encontrarlo, aunque este exista, pues por ello se llama oculto). Cuando detectamos que el nombre comienza por un punto, directamente devolvemos un error de que el fichero no se encuentra.

```
ls26151@vela:~/SOA$ ./copycat /find Ext2_1024_PuntersIndirectes.ex2 .ficheroOculto

        Error: File not found.
```

Si buscamos una carpeta existente

```
ls26151@vela:~/SOA$ ./copycat /find Ext2_1024_PuntersIndirectes.ex2 carpeta

        Is a directory!.
```

Si buscamos un fichero existente (y no oculto)

```
ls26151@vela:~/SOA$ ./copycat /find Ext2_1024_PuntersIndirectes.ex2 999

        File found in inode [1012]!
        Size 29 Bytes
```

Si intentamos buscar un directorio tipo “ . ” o “ .. ”

```
ls26151@vela:~/SOA$ ./copycat /find Ext2_1024_PuntersIndirectes.ex2 .

        No se realizara busqueda de un nombre reservado [ . ]

        Error: File not found.
```

PROBLEMAS ENCONTRADOS

En esta fase, para el apartado de la FAT he tenido problemas al obtener el ***Root Directory Region*** ya que no había entendido correctamente el cálculo a realizar que comenta la documentación de FAT, pero una vez solucionado pude ver el listado de todas las entradas en la ***Root Directory Region***. Tan solo quedaba pues calcular la ***bitmask*** y filtrar los datos que mostraríamos.

Para la fase de EXT han sido mucho mayores los problemas encontrados, pero todos ellos han sido por la dificultad que he tenido hasta entender como estaba formada la estructura. Una vez entendida la implementación ha sido sencilla (que no corta).

FASE 3 (/cat)

Para esta fase debemos mostrar la información de un fichero existente dentro del sistema de archivos EXT. Para ello haremos uso de la función ***extSearchInSubfolder*** para obtener la ***Directory Entry*** y por tanto el ***Inode*** con todos los punteros a bloques. La llamada se realizará usando el parámetro /cat.

Para ello realizaremos la misma tarea que cuando hacíamos la búsqueda de ficheros, salvo que ahora sabemos que no nos vamos a encontrar un directorio ya que ya tenemos el archivo. Como un Inode solo es para un archivo, tendremos que leer los punteros directos, movernos al bloque de información.

Como conocemos el tamaño del fichero, podemos saber cuándo parar de leer, si usamos un contador de bytes, usaremos la condición en los bucles de

```
do { . . . } while ( readedBytes < inode.i_size );
```

Esta función se aprovechará más adelante para la fase 4 que será la copia de fichero.

Como apartado opcional también se ha realizado el mostrar el contenido de un fichero en el **sistema FAT**. Para ello he tenido que realizar los siguientes pasos.

Gracias a la fase 2 la parte de encontrar el fichero ya estaba resuelta, además la función de encontrarlo, devolvía la posición ***lseek*** donde comenzaba su entrada de directorio. Tal y como hice para EXT, creé una función llamada ***loadFatDirectoryEntry*** que cargaba toda la información de esa entrada de directorio en un struct.

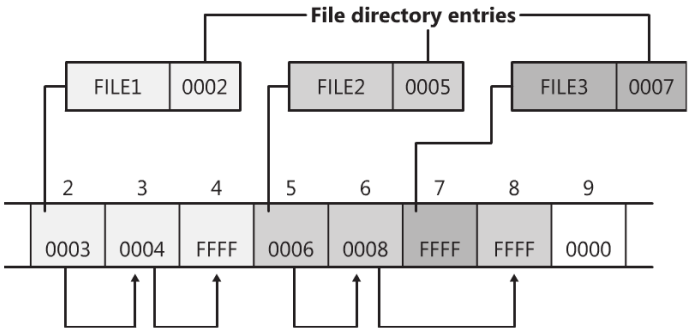
Gracias a la información cargada puedo saber cuál es el cluster inicial de un fichero, tan solo queda ir a la fat y leer las entradas de ese fichero.

Para ello he implementado la función ***mapFatClustersInArray*** que rellena un array con todos los clusters asociados al fichero, de esta forma evito estar consultando a cada momento la tabla de FAT, facilitándome así la programación para el mostrado del fichero.

Para saber la longitud del array he realizado una división entre file_size / clusterSize (ojo! cluster, no sector) redondeando al alza en caso de encontrar decimales.

```
unsigned int blocksNeeded = fatdir.size / clusterSize;
if ( (fatdir.size % clusterSize) > 0 ){
        blocksNeeded++;
}
// Asignamos +1 para el bloque FFFF
blocksNeeded++;
int blocks[blocksNeeded];
```

El funcionamiento de la fat se puede ver en la siguiente imagen



Podemos ver que FILE1 comienza en el cluster 0002. Si vamos al índice 0002 de la tabla fat, veremos que nos indica 0003, esto significa que el siguiente bloque de información está en el cluster 0003. Si vamos al índice 0003 de la tabla FAT veremos que nos indica 0004, en el cluster 0004 encontraremos el siguiente bloque de información. Por último si vamos al índice 0004 de la tabla FAT, veremos que el siguiente bloque nos indica FFFF, esto quiere decir que no hay más bloques de información.

Como podemos ver en FILE3, este comienza en el cluster 0007 y si nos vamos al índice 0007 de la tabla FAT podemos ver que tenemos FFFF, esto significa que ese fichero ocupa 1 o menos clusters y por lo tanto no hay más entradas en la tabla FAT.

Ahora que ya tenemos en un array mapeados los números de cluster donde está la información, queda la tarea de posicionarse en ellos y leer la información guardada.

Para ello he creado la función ***getFatFirstDataSector*** que devuelve el número de sector donde comienza la información. Para encontrar pues el primer sector del cluster, tal y como se indica en la documentación debemos realizar la operación siguiente. Suponiendo que el start cluster fuera el 0005.

```
firstSectorOfCluster = ( ( 0005 -2 ) * fat.sectors_cluster ) + getFatFirstDataSector(fat);
```

Donde sectors_cluster es el número de sectores asociados a un cluster y el -2 tal y como dice la documentación.

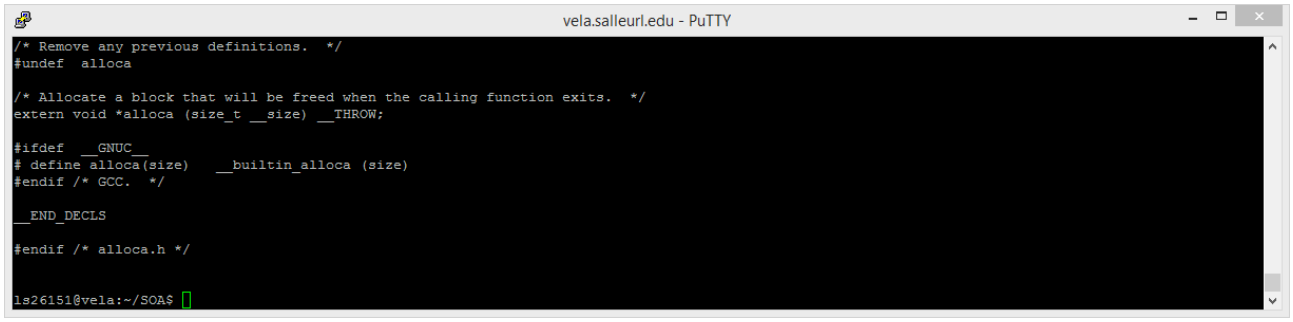
*Given any valid data cluster number N, the sector number of the first sector of that cluster (again relative to sector 0 of the FAT volume) is computed as follows: FirstSectorofCluster = ((N – 2) * BPB_SecPerClus) + FirstDataSector;*

PRUEBAS REALIZADAS

Mostrar el contenido del fichero PROVA3.TXT del sistema EXT2 Con-info.ex2

Mostrar el contenido del fichero tralala (dentro de subcarpeta) del sistema EXT2 Punters_indirectes.ex2

Mostrar el contenido del fichero alloca.h del sistema FAT16 Fat16_2k



Mostrar el contenido del fichero C99.txt del sistema FAT16 Fat16_1K

PROBLEMAS ENCONTRADOS

Por la parte de EXT realmente no ha habido muchos problemas, únicamente un contador mal reseteado que hacía que el programa no encontrará ficheros / directorios en los punteros indirectos de los inode.

Por la parte de FAT, he tenido problemas para el cálculo de la posición ***lseek*** de los clusters del fichero, llegando a obtener posiciones fuera del tamaño del fichero y por lo tanto no mostrando ninguna información de este.

FASE 4 (/copy)

En esta fase se trata de realizar la copia de un archivo de Ext en la partición de Fat. Para ello he comenzado primero a buscar si hay una root directory entry libre en la partición de Fat, ya que sabiendo que como máximo pueden ser 512 entradas, el tiempo de búsqueda será como máximo siempre el mismo, evitando tener que buscar el archivo en la partición de Ext (pudiendo estar dentro de varias subcarpetas con muchos ficheros…).

Para ello se hace uso de una función llamada searchFreeRootDirInFat, la cual se posiciona en la root directory entry y lee todas las entradas hasta encontrar una que contenga un 0xE5 o un 0x00 en el primer carácter del nombre. Esto tal y como indica en la documentación, quiere decir que esa entrada esta “eliminada” y que aunque haya información, es como si no fuera valida.

Si obtenemos una posición libre, entonces buscaremos el fichero que nos han pasado por parámetro en la partición de Ext, haciendo uso de la función de la fase 2, findInExt. Obtenemos el seek de la directory entry, la cual podemos leer, obtener el inodo y del inodo obtenemos el size del archivo.

Ahora que sabemos que tenemos entrada libre en la root directory, y que el archivo que buscamos es un archivo valido, debemos calcular cuántos clusters necesitamos para poder guardarlo en nuestra partición. Dividimos pues el tamaño del archivo entre el tamaño del cluster redondeando hacia arriba si diera decimales, de ello se encarga la función ***calculateNecesaryClusters***.

Creamos un array del tamaño de los clusters necesarios y se lo pasamos a la función mapFatClustersFree que nos devolverá el array lleno (si hay espacio) de los números de cluster libres donde podemos guardar el archivo. Si el return de esta función es -1 significa que no ha encontrado suficientes clusters libres, y el programa mostrará un error indicándolo.

En el caso de que haya encontrado suficientes clusters libres, procedemos a convertir la entrada de root directory de Ext en la de Fat, encargándose de ello convertFromExtToFatDirEntry devolviéndonos directamente la entrada de FatDirectory preparada con los datos para directamente guardarla en el fichero de sistema de archivos.

Si los clusters necesarios son 0, entonces machacamos el campo firstblock de la entrada de directorio de fat con el código 0xFFFF el cual tal y como indica en la documentación quiere decir que ya no hay más bloques de información.

Con los datos de la root entry pasados ya en Fat, buscamos si existe un archivo con el mismo nombre en la FAT y de ser así, indicamos al usuario que no se copiara el fichero puesto que ya hay uno existente.

En el caso de que no encontremos un fichero con el nombre del fichero que hemos convertido a Fat, procedemos a guardar la entrada en la root directory con la función ***fatSaveEntryInRooTDirectory*** y a continuación usamos el mismo código usado en la fase 3 para hacer el cat del fichero en Ext para ir leyendo bloques e ir copiándolo en los clusters de Fat.

Para escribir los datos, se ha creado una función llamada writeFat donde se le pasa un array con la información que se ha copiado de Ext (tamaño máximo blocksize) y se le pasa esta función writeFat junto con el file descriptor de fat, la estructura de información de fat, el número de cluster donde se ha de guardar y los datos que se le han leído. La función se encarga de posicionarse en el cluster y escribir la información.

Por último tan solo debemos actualizar la tabla Fat con los datos de clusters libres que hemos obtenido anteriormente. Marcando el último con el código 0xFFFF tal y como se indica en la documentación.

PRUEBAS REALIZADAS

Para realizar pruebas he copiado los archivos ***tralala***, ***PROVA1.TXT*** , ***PROVA2.TXT***, ***PROVA3.TXT*** y ***999*** en la partición de Fat. Posteriormente se ha redirigido el cat de ambas particiones a diferentes ficheros y comprobado que el tamaño y el contenido de ambos es el mismo.

```
ls26151@vela:~/SOA$ ./copycat /copy Ext2_1024_Con-info.ex2 Fat16_1024_Binary.bin PROVA2.TXT
Guardando nombre PROVA2

File saved!
ls26151@vela:~/SOA$ ./copycat /cat Fat16_1024_Binary.bin PROVA2.TXT > fatPROVA2
ls26151@vela:~/SOA$ ./copycat /cat Ext2_1024_Con-info.ex2 PROVA2.TXT > extPROVA2
ls26151@vela:~/SOA$ ls -l
total 17176
-rw-r--r-- 1 ls26151 LS   13756 2015-06-03 22:16 backup.tar.gz
-rwxr-xr-x 1 ls26151 LS   32256 2015-06-03 23:46 copycat
-rw-r--r-- 1 ls26151 LS    1011 2015-06-02 12:21 copycat.c
-rw-r--r-- 1 ls26151 LS 4192256 2015-06-02 13:15 Ext2_1024_Con-info.ex2
-rw-r--r-- 1 ls26151 LS 4194304 2015-06-02 13:55 Ext2_1024_PuntersIndirectes.ex2
-rw-r--r-- 1 ls26151 LS  350711 2015-06-03 23:58 extPROVA2
-rw-r--r-- 1 ls26151 LS 8288256 2015-06-03 23:57 Fat16_1024_Binary.bin
-rw-r--r-- 1 ls26151 LS  350711 2015-06-03 23:57 fatPROVA2
-rw-r--r-- 1 ls26151 LS   62351 2015-06-03 23:46 func.c
-rw-r--r-- 1 ls26151 LS    5617 2015-06-02 18:10 func.h
-rw-r--r-- 1 ls26151 LS     240 2015-04-25 19:18 makefile
-rwxrwxrwx 1 ls26151 LS   10245 2015-04-25 20:30 TesterFAT16.exe
-rw-r--r-- 1 ls26151 LS      23 2015-06-02 19:29 testmontado
```

PROBLEMAS ENCONTRADOS

En esta fase el mayor problema ha sido como juntar la lectura y escritura de los ficheros, y sobretodo el convertir la información de la entrada de root directory de Ext a la root directory de Fat. Gracias a las funciones “helper” que he ido realizando a lo largo de la práctica, la parte restante ha sido bastante sencilla y rápida de realizar.

CONCLUSIONES Y COMENTARIOS

Gracias a esta práctica he podido entender que un sistema de archivos no es nada extraño, simplemente que es como un fichero con cierta estructura (cada cual diferente).

Sobre todo es interesante el ver como se han implementado diferentes formas de guardar información y que cuando estamos usando un sistema operativo este se encarga de la tarea de hablar con el sistema de archivos y darle los datos a leer/guardar.

VALORACIONES METODOLOGÍA

Por un lado la metodología me parece correcta puesto que nos muestra una breve explicación que nos ayuda a empezar y luego tenemos que profundizar usando la documentación.