



# INFORME

# TP1: FAT

## Sistemas Operativos y Redes 2

Docentes:

- Alexis Tcach
- Alan Echabari

Integrantes del grupo:

- Luis Curto
- Sebastian Pintos
- Matias Morales
- Vanesa Vera

# TP 1 - Sistemas Operativos y Redes 2 - UNGS

## FAT File System

Vanesa Vera - Matias Morales - Sebastian Pintos - Luis Curto

23 de Abril de 2023

Realizamos una exploración en profundidad del sistema de archivos FAT 12, siguiendo lo propuesto por la catedra. Para esto estaremos trabajando con el archivo de imagen provisto llamado test.img. Lo estaremos leyendo a bajo nivel, o sea directo del iso, pero lo montaremos para entender y comprobar lo que vamos haciendo. Detallamos lo resuelto en cada punto y con capturas de pantallas damos pruebas de lo realizado. El código se incorpora al texto de manera de poder dar fe y explicar el mismo.

### 1. ¿ Para qué se ha puesto umask=000 ?

El comando **mount test.img /mnt -o loop,umask=000** monta el archivo de imagen de disco llamado **test.img** en el directorio de punto de montaje **/mnt**. ( Fig. 1 )  
La opción **umask=000** establece los permisos de archivo predeterminados en el punto de montaje **/mnt** como lectura, escritura y ejecución para el propietario, el grupo y otros usuarios. En este caso le estamos dando todos los permisos al poner 000 es decir **drwxrwxrwx**. El comando **loop** nos permitirá trabajar con el disco montado como un block device.

```
alumno@alumno-virtualbox:~$ ls
bin dev home lib32 libx32 media opt root sbin srv snap var
boot etc lib lib64 lost+found misc proc run snap sys usr
alumno@alumno-virtualbox:~$ cd mnt
alumno@alumno-virtualbox:/mnt$ ls
hola.txt prueba.txt
```

mount test.img /mnt -o loop,umask=000

(Fig. 1)

A continuación mostramos como se ven los discos antes y después de montarlos, para esto usamos el comando **sudo fdisk -l**. ( Fig. 2 y Fig. 3 )

```
alumno@alumno-virtualbox:~$ sudo fdisk -l
Disk /dev/loop0: 238,6 MiB, 250175488 bytes, 488624 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes

Disk /dev/loop2: 116,8 MiB, 122458112 bytes, 239176 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes

Disk /dev/loop3: 116,78 MiB, 122433536 bytes, 239128 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes

Disk /dev/loop4: 63,32 MiB, 66392064 bytes, 129672 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes

Disk /dev/loop5: 291,87 MiB, 306024448 bytes, 597704 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes
```

fdisk antes del montaje

(Fig. 2)

```
alumno@alumno-virtualbox:~/Desktop/entregable$ sudo fdisk -l
Disk /dev/loop0: 238,6 MiB, 250175488 bytes, 488624 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes

Disk /dev/loop1: 1 MiB, 1048576 bytes, 2048 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes
Tipo de etiqueta de disco: dos
Identificador del disco: 0x00000000

Dispositivo Inicio Comienzo Final Sectores Tamaño Id Tipo
/dev/loop1p1 * 1 2047 2047 1023,5K 1 FAT12

Disk /dev/loop2: 116,8 MiB, 122458112 bytes, 239176 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes

Disk /dev/loop3: 116,78 MiB, 122433536 bytes, 239128 sectores
Unidades: sectores de 1 * 512 = 512 bytes
Tamaño de sector (lógico/físico): 512 bytes / 512 bytes
Tamaño de E/S (mínimo/óptimo): 512 bytes / 512 bytes
```

fdisk después del montaje

(Fig. 3)

Podemos ver como después del montaje se agrega el dispositivo **loop1**, que corresponde con el montaje de **test.img**. Cabe destacar que el loop depende de diferentes factores por lo que para saber cual es el correspondiente podemos hacer el procedimiento anterior.

### 2. a) Mostrando el MBR con el Hex Editor : Muestre los primeros bytes y la tabla de particiones. ¿Cuántas particiones hay ? Muestre claramente en qué lugar puede observarlo.

El MBR (Master Boot Record) es una pequeña área de almacenamiento ubicada en el primer sector de un disco duro, que contiene información esencial para arrancar el sistema operativo. Tiene un tamaño de 512 bytes donde del 0 al 445 se encuentra el boot code, del 446 al 509 están las tablas de particiones y por último del 510 a 511 tenemos el Signaturevalue.

Para ver el MBR en ghex debemos seguir los siguientes pasos:



En el paso 2 (Fig 10) podemos ver que los datos solicitados son mostrados por pantalla al correr nuestro programa en c. Se pueden visualizar las 4 particiones de nuestra tabla y los datos de cada una. En nuestro caso solo la primera partición tiene datos porque las demás están vacías.

c) Muestre en el Hex Editor si la primera partición es booteable o no. ¿Lo es?

Para saber si una partición es booteable o no, tenemos el primer byte de la entrada. (Fig. 12) Este byte nombrado en el ítem a, es el de estado, y nos indica si es booteable o no. Si el primer byte es 0x80 nos indica que la partición es booteable, por otro lado si es 0 nos indica lo contrario. Veámoslo en ghex:

**80** 00 02 00 01 20 20 00 01 00 00 00 FF 07 00 00  
Primer byte de la primera partición

(Fig. 12)

Podemos ver que el primer byte en 80 (Fig. 12) por lo tanto nuestra primera partición es booteable.

d) Muestre, mediante un programa en C, para la primera partición: el flag de booteable, la dirección Cylinder-head-sector (chs), el tipo de partición y su tamaño en sectores.

Para mostrar mediante un programa en C la primera partición creamos un nuevo archivo llamado **read\_frist\_partition.c** (Fig. 13), parecido al **read\_mbr.c** pero en este caso no hace falta que recorramos toda la tabla de particiones sino que solo mostramos la primera entrada.

```

int main()
{
    FILE * in = fopen("test.img", "rb");
    unsigned int start_sector, length_sectors;

    fseek(in, 0x1BE , SEEK_SET); //Voy a buscar la tabla de particiones

    //Imprimimos los datos correspondientes
    printf("Partición entry \#d: First byte %02X\n", 0, fgetc(in));
    printf("Comienzo de partición en CHS: %02X:%02X:%02X\n", fgetc(in), fgetc(in), fgetc(in));
    printf("Primer byte de dirección LBA: %02X\n", fgetc(in));
    printf("Fin de partición en CHS: %02X:%02X:%02X\n", fgetc(in), fgetc(in), fgetc(in));

    fread(&start_sector, 4, 1, in); //leo elemento y lo guardo en start_sector
    fread(&length_sectors, 4, 1, in); //leo elemento y lo guardo en length_sectors
    printf("-----\n");

    fclose(in);
    return 0;
}

```

(Fig. 13)

Luego compilamos y ejecutamos el programa **read\_first\_partition.c** con los comandos:  
**make all**  
**./read first partition**

```
alumno@alumno-virtualbox:~/Desktop/entregables$ gcc read_first_partition.c -o read_first_partition
alumno@alumno-virtualbox:~/Desktop/entregables$ ./read_first_partition
Particion de disco en CHS: 00-29-20
Comienzo de particion en CHS: 00-02-00
Partition type: 0x01
Fin de particion en CHS: 00-29-20
Direccion LBA relativa 0x00000001, de tamaño en sectores 2047
```

(Fig. 14)

Por lo tanto podemos ver por pantalla los datos solicitados mediante nuestro programa en c (Fig. 14).

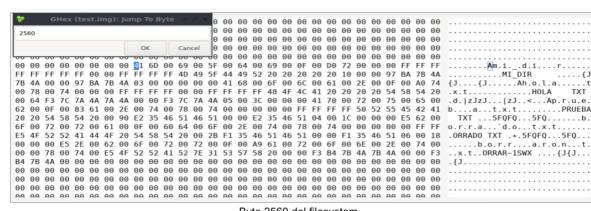
3. a) ¿Cuántos y cuáles archivos tiene el filesystem ? Muestrelos con Ghex y genere el código C para mostrarlos.

Primero para saber cuantos y cuales archivos tiene el filesystem a través de ghex debemos saber en qué byte comienza nuestro directorio raíz. Para saber esto debemos buscar en el boot sector los siguientes datos:

- Número de sectores reservados. Este valor se encuentra en los bytes 14 y 15 del sector de boot. En nuestro caso es igual a 1.
  - Número de sectores por FAT este valor se encuentra en los bytes 22 y 23, en nuestro caso este valor es 2. Cada FAT es una tabla que contiene información sobre los archivos del sistema, y en FAT12 hay dos copias de esta tabla. Por lo tanto, el número total de sectores utilizados para almacenar la tabla FAT es igual al número de copias de la tabla FAT multiplicado por el número de sectores por FAT.

Una vez que tenemos el número de sectores reservados y el número de sectores por FAT, podemos calcular el número del sector que contiene el directorio raíz. Este valor es igual a la suma de los sectores reservados y el número de sectores utilizados para almacenar la tabla FAT. Es decir  $1 + 2 + 2 = 5$ .

Ahora tenemos el número de sector que contiene el directorio raíz (5), podemos calcular la dirección del primer byte del directorio raíz. Para hacer esto, multiplicamos el número del sector por el tamaño de cada sector, que es de 512 bytes para FAT12. Por lo tanto obtenemos  $5 * 512 = 2560$  bytes (Fig. 15) que es el byte en donde comienza nuestro directorio raíz.



(Fig. 15)

Cada entrada del directorio raíz ocupa 32 bytes. En las siguientes imágenes podemos observar cada entrada distinguida con un color diferente (Fig. 16 y 17).

(Fig. 16)

```

.....[Am.i...d.i....r.....
.....MI DIR .....{J
{J...{J...Ah.o.l.a....t
.x.t.....HOLA TXT
.d.|z|zJ...|zJ...<..Ap.r.u.e.
b...a...t.x.t.....PRUEBA
TXT ...5F0F0...5F0...b.
o.r.r.a...`d.o...t.x.t.
.orrado TXT .+..5F0F0...5F0...
...b.o.r.r...a.r.o.n...t.
..x.t...ORRAR-SWX ...{J{J...
.{J...
.....]
.....]
.....]

```

Entradas del directorio raíz vista ascii

(Fig. 17)

Para contar el número de archivos válidos en el directorio raíz de un sistema de archivos FAT12, se pueden recorrer todas las entradas del directorio raíz y verificar si cada una de ellas es válida.

Una entrada de directorio es válida si el primer byte de su nombre de archivo no es igual a 0xE5. También es válido si el primer byte es igual a 0x00, lo que indica el final del directorio raíz. Por lo tanto en las imágenes anteriores podemos ver como las primeras 6 entradas son válidas y las últimas 4 no lo son porque comienzan con E5.

Ahora para saber cuáles de las 6 entradas corresponden a archivo tenemos que verificar si tiene el atributo «Archivo» establecido en su octavo byte. Si el valor de este byte es 0x20 (Fig. 18), significa que la entrada de directorio es un archivo. Si el valor es distinto de 0x20, entonces la entrada de directorio puede ser un directorio, un archivo de sistema u otro tipo de archivo especial.

```

00 00 00 00 00 00 00 00 00 41 60 00 69 00 5F 00 64 00 69 00 0F 00 00 00 FF FF FF
FF FF FF FF 00 00 FF FF FF 4D 49 5F 44 49 52 20 [20] 29 20 20 10 00 00 97 BA 7A 4A
7B 4A 00 00 97 BA 7B 4A 03 00 00 00 00 00 41 68 00 6F 00 6C 00 61 00 2E 00 00 00 A0 74
00 78 00 74 00 00 00 FF FF FF 00 00 FF FF FF 48 4F 4C 41 20 20 [20] 54 58 54 26
00 64 F3 7C 7A 4A 00 00 F3 7C 7A 4A 05 00 3C 00 00 00 41 70 00 72 00 75 00 65 00
62 00 00 83 61 00 00 74 00 78 00 66 00 00 00 FF FF FF 58 52 55 45 42 41
20 [20] 54 58 54 20 00 98 E2 35 46 51 00 00 E2 35 46 51 04 00 1C 00 00 00 E5 62 00
6F 00 72 00 72 00 61 00 00 0F 00 68 00 0F 00 74 00 78 00 00 00 00 00 FF FF

```

Archivos que tienen el atributo "Archivo" establecido en su octavo byte.

(Fig. 18)

Por lo tanto tenemos 3 archivos (Fig. 19) y para ver cuales son podemos observar la vista ascii de ghex.

```

.....Am.i...d.i....r.....
.....MI DIR .....{J
{J...{J...Ah.o.l.a....t
.x.t.....HOLA TXT
.d.|z|zJ...|zJ...<..Ap.r.u.e.
b...a...t.x.t.....PRUEBA
TXT ...5F0F0...5F0...b.

```

Archivos en vista ascii

(Fig. 19)

Entonces tenemos 3 archivos una llamada “mi dir”, y dos archivos .txt llamados “hola” y “prueba”.

Para mostrar lo anterior pero en C, modificamos el programa **read\_root.c** de la siguiente manera. Agregamos los datos faltantes a la estructura FAT y agregamos una estructura más que representa nuestra directory entry. A continuación se muestran capturas del código con las mismas (Fig. 20 y 21).

```

typedef struct {
    unsigned short l; // Salto de código de arranque
    char oem[8]; // Nombre del fabricante del sistema de archivos
    unsigned short sector_size; // Tamaño de cada sector en bytes
    unsigned char sector_cluster; // Número de sectores por cluster
    unsigned short reserved_sectors; // Número de sectores reservados en el sistema de archivos
    unsigned char number_of_fats; // Número de tablas FAT en el sistema de archivos
    unsigned short root_dir_entries; // Número de entradas en el directorio raíz
    unsigned short sectors_per_cluster; // Número total de sectores en el volumen
    unsigned short num_heads; // Número de cabezas de la unidad de disco duro
    unsigned short fat_size_sectors; // Tamaño de cada tabla FAT en sectores
    unsigned short sector_track; // Número de sectores por pista
    unsigned short short_headers; // Número de cabezas de la estructura/escritura
    unsigned short sectors_per_fat; // Número de sectores que componen el inicio de la partición
    unsigned int sector_partition; // Número total de sectores en la partición
    unsigned char physical_device; // Número de unidad física
    unsigned char current_header; // Número de cabeza de lectura/escritura actual
    unsigned short first_l; // Número de primera tabla FAT en sectores
    unsigned short last_l; // Última tabla FAT en sectores
    unsigned int volume_id; // Identificación única del volumen
    char volume_label[11]; // Etiqueta del volumen
    char fs_type[8]; // Nombre del sistema de archivos (FAT12 en este caso)
    char cbs[448]; // Código de arranque
    unsigned short boot_sector_signature; // Código del sector de arranque
} _attribute__((packed)) Fat12BootSector;

```

Estructura fat con sus nuevas variables

(Fig. 20)

```

typedef struct {
    unsigned char filename[8]; // Nombre del archivo (8 caracteres)
    unsigned char extension[3]; // Extensión del archivo (3 caracteres)
    unsigned char attributes[1]; // Atributos del archivo
    unsigned char reserved; // Campos reservados
    unsigned long create_time; // Tiempo de creación
    unsigned long create_time_minutes_seconds; // Hora en que se creó el archivo
    unsigned short create_date; // Fecha de creación del archivo
    unsigned short access_date; // Fecha de acceso al archivo
    unsigned short modify_date; // Fecha de modificación
    unsigned short modify_time; // Hora de modificación del archivo
    unsigned short cluster_low; // Número de clúster bajo
    unsigned int size_of_file; // Tamaño del archivo en bytes
} _attribute__((packed)) Fat12Entry;

```

Estructura del entry

(Fig. 21)

Luego añadimos una función (Fig. 22) para verificar la información del entry y dependiendo si es una entrada válida, un directorio o un archivo, imprime por pantalla su nombre y extensión.

```

void print_file_info(Fat12Entry *entry)
{
    // Verificar si la entrada está vacía
    if (entry->filename[0] == 0x00) {
        return;
    }

    // Verificar si la entrada representa un archivo borrado
    if (entry->filename[0] == 0xE5) {
        printf("Deleted file: [%s.%s]\n", entry->filename, entry->extension);
        return;
    }

    // Verificar el atributo de la entrada
    switch(entry->attributes[0]) {
        case 0x10: // Entrada tipo directorio
            printf("Directory: [%s]\n", entry->filename); // Imprimo directorio
            break;
        case 0x20: // Entrada tipo archivo
            printf("File: [%s.%s]\n", entry->filename, entry->extension); // Imprimo archivo
            break;
    }
}

```

Función que verifica e imprime la información

(Fig. 22)

Luego desde el main debemos calcular en qué posición empieza el directorio raíz (Fig. 23) y luego movernos a ahí para recorrer las entradas.

```

position_root_directory = (bs.reserved_sectors-1 + bs.fat_size_sectors
                           * bs.number_of_fats) * bs.sector_size; /* Calculo posicion inicial
                                                               | del root directory */

fseek(in, position_root_directory, SEEK_CUR); // Ir al inicio del root directory

printf("Root dir entries %d \n", bs.root_dir_entries);

for(i=0; i<bs.root_dir_entries; i++) // Recorro las entradas e imprimo
{
    fread(&entry, sizeof(entry), 1, in);
    print_file_info(&entry);
}

```

Calcula la posición inicial del directorio raíz y recorre las entradas

(Fig. 23)

Por último para compilar y ejecutar nuestro programa debemos ejecutar los comandos **make all** y **./read\_root** esto nos imprimirá por pantalla los archivos del filesystem. (Fig. 24)

```

Partition type: 1
Encuentrado FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATS
Root dir entries 512
Directory: [MI DIR ]
File: [HOLA TXT ]
File: [PRUEBA ]
Deleted file: [ ]
Deleted file: [BORRADOTXT]
Deleted file: [ ]
Deleted file: [BORRAR-SWX]

```

Salida por pantalla de read\_root

(Fig. 24)



Contenido en vista Ascii

(Fig. 33)

Para mostrar el contenido de los archivos de nuestro filesystem por un programa en C, creamos **read\_file.c** el mismo recorre el directorio raíz, busca el primer cluster de cada archivo y lee los datos correspondientes.

Primero calculamos la posición del primer cluster (Fig. 34 y 35) y su tamaño para luego pasarlo por parámetro en la función **print\_file\_info** la cual se encargará de imprimir el contenido del archivo y llamar a la función que lee sus datos.

```
Position_root_directory = (bs.reserved_sectors * bs.fat_size_sectors
                           + bs.number_of_fats) * bs.sector_size; /* Calculo posicion inicial
                           | del root directory */

fseek(in, position_root_directory, SEEK_CUR); // Ir al inicio del root directory

printf("\nRoot dir_entries %d \n", bs.root_dir_entries);

firstCluster = tell(in) + (bs.root_dir_entries * sizeof(entry)); /* Obtenemos la posición del
                     | primer byte del primer cluster de datos*/
                     | sizeofCluster = bs.sector_size * bs.sector_cluster; // Tamaño en bytes de un cluster.

for(i=0; i<bs.root_dir_entries; i++) // Recorro las entradas e imprimo
{
    fread(&entry, sizeof(entry), 1, in);
    print_file_info(&entry, firstCluster, sizeofCluster);
}
```

Parte del código donde se calcula la posición del primer cluster y su tamaño para pasarselos a **print\_file\_info()**

(Fig. 34)

```
void print_file_info(Fat12Entry *entry, unsigned short firstCluster, unsigned short clusterSize)
{
    // Verificar el atributo de la entrada
    switch(entry->attributes[0])
    {
        case 0x10: // Entrada tipo directorio
            return;

        case 0x00: // Entrada tipo archivo
            printf("\nEl contenido del archivo %.75s.%s : ", entry->filename, entry->extension);
            leer(firstCluster, entry->cluster_low, clusterSize, entry->size_of_file);
            return;
    }
}

Función print_file_info() verifica si es un archivo y llama a la función leer()
```

(Fig. 35)

Luego la función leer recibe como parámetros **firstCluster** que es la posición del primer cluster, **fileFirstCluster** que indica el número de clúster en el que comienza el archivo, **clusterSize** que es el tamaño del cluster y por último **fileSize** que indica el tamaño del archivo. (Fig. 36) Dentro de la función abrimos el archivo de imagen en modo lectura binaria, creamos una variable para almacenar (char leer[]), nos movemos a la posición del primer cluster, leemos el contenido y lo guardamos en la variable para poder imprimir el contenido.

```
void leer(unsigned short firstCluster, unsigned short fileFirstCluster, unsigned short clusterSize,
         int fileSize)
{
    FILE * in = fopen("test.img", "rb");
    int i;
    char leer[fileSize]; // Creamos un array de caracteres para almacenar el contenido del archivo

    fseek(in, firstCluster - (fileFirstCluster - 2) * clusterSize, SEEK_SET); /* Nos posicionamos en el
                           | primer cluster del archivo */

    fread(leer, fileSize, 1, in); // Leemos el contenido del archivo y lo almacenamos en el array
    for(i=0; i<fileSize; i++) // Recorremos el array e imprimimos su contenido
    {
        printf("%c", leer[i]);
    }

    fclose(in);
}
```

Código de la función que lee el archivo

(Fig. 36)

Por último, compilamos y ejecutamos el código (Fig. 37) con los comandos anteriores y obtenemos la siguiente salida por pantalla.

Salida por pantalla del programa **read\_file**

(Fig. 37)

c) Cree código C para que dado un archivo (o una parte), lo busque y si lo encuentra y el mismo se encuentra borrado, lo recupere.

Para recuperar un archivo creamos el programa **recovery\_file.c** el cual primero busca si el archivo (o una parte) coincide con alguno de los archivos borrados. Luego, si coincide, recorre los clusters de datos y los escribe en un nuevo archivo. A continuación se mostrará el código y su funcionamiento con más detalle.

Tenemos que indicar que archivo o parte del archivo recupero, eso lo hacemos con una variable en el main (Fig. 38) que luego le pasamos a las demás funciones.

```
int main()
{
    FILE *in = fopen("test.img", "rb");
    int i, position_root_directory, size_cluster;
    PartitionTable pt[4];
    Fat12BootSector bs;
    Fat12Entry entry;
    unsigned short first_cluster;
    const char *filename = "LAPAPA.TXT"; // Nombre del archivo a recuperar

    fseek(in, 0x1BE, SEEK_SET); // Ir al inicio de la tabla de particiones.
    fread(pt, sizeof(PartitionTable), 4, in); // Leo entradas

    for (i = 0; i < 4; i++)
    { // Buscar particiones booteables
```

Código donde indicamos el nombre del archivo a recuperar.

(Fig. 38)

Para verificar si la variable anterior coincide con alguno de los archivos borrados del sistema (Fig. 39) tenemos que recorrer el directorio raíz buscando los archivos borrados. Una vez que tenemos un archivo borrado, verificamos si el nombre dado y la extensión están contenidos en este, si esto sucede pasamos a recuperarlo con la función **recovery\_file**.

```
void print_file_info(Fat12Entry *entry, unsigned short firstCluster, unsigned short clusterSize,
                     const char *filename, FILE *in)
{
    // Verificar si la entrada representa un archivo borrado
    if (entry->filename[0] == 0xE5)
    {
        char delete_filename[100]; // Variable para guardar el archivo a borrar
        strcpy(delete_filename, filename); // Pasamos el nombre a la variable anterior

        char *name = strtok(delete_filename, "."); // Dividimos el nombre de la extensión
        char *extension = strtok(NULL, ".");

        char *compare_name = strtrunc(entry->filename, name + 1); // Si el nombre indicado lo contiene
        char *compare_ext = strtrunc(extension, extension); // Si la extensión indicada lo contiene

        if (*compare_name != NULL && compare_ext != NULL) // Si el nombre y la extensión estan
        { // contenidas lo recuperamos
            printf("Deleted file: [%s.%s]\n", entry->filename, entry->extension);
            recovery_file(firstCluster, entry->cluster_low, clusterSize, entry->size_of_file,
                          name, extension);
        }
    }
}
```

Código donde se verifica si el archivo borrado contiene el nombre y la extensión dada.

(Fig. 39)

Luego, la función **recovery\_file** es la encargada de recuperar el archivo de la siguiente manera:

Primero creamos un archivo nuevo donde se van a guardar los datos e inicializamos las variables necesarias.

Después de esto, vamos a recorrer verificando si el cluster inicial no es el último y la cantidad de bytes a leer es mayor a cero.

Si se cumple la condición anterior lo que hacemos es pararnos en la posición del primer cluster, verificamos la cantidad de bytes que voy a escribir y escribo en el nuevo archivo.

Por último me muevo al siguiente cluster con la función `get_next_cluster`. Este bucle se repetirá hasta que se recuperen los datos de todo el archivo. (Fig. 40)

```
unsigned short current_cluster = deletedFileFirstCluster; // Cluster actual
FILE *new_file = fopen(new_file_name, "wb"); // Creamos nuevo archivo para almacenar

// Recorra si no llega al ultimo cluster y si la cant de bytes restantes es mayor a cero/
while (current_cluster < 0xFF && bytes_remaining > 0)
{
    fseek(in, firstCluster + ((current_cluster - 2) * clusterSize), SEEK_SET); // Nos posicionamos
    fread(cluster, clusterSize, 1, in);

    if (bytes_remaining > clusterSize) // Si la cantidad restante es mayor al tamaño del cluster
        bytes_to_write = clusterSize;
    else
        bytes_to_write = bytes_remaining; // Si no la cantidad a escribir son los bytes restantes

    fwrite(cluster, 1, bytes_to_write, new_file); // Escribo el cluster

    // Actualizar el numero de bytes restantes y el número del siguiente cluster
    bytes_remaining -= bytes_to_write;
    current_cluster = get_next_cluster(current_cluster, fat_table);
}
```

Código que recupera el archivo.

(Fig. 40)

La función `get_next_cluster` (Fig. 41) toma como parámetros el número del cluster actual y una tabla FAT en forma de arreglo de bytes. Lo que hace es calcular la posición en la tabla FAT que le corresponde al clúster actual.

El formato de la tabla FAT en el sistema de archivos FAT12 se divide en entradas de 12 bits. Para acceder a la entrada correspondiente a un cluster en particular, se necesita calcular un offset en la tabla FAT. Este offset se calcula sumando el número del cluster actual más la mitad del número del cluster actual (que se calcula como la división entera por 2).

Una vez que ya calculamos el offset se lee el primer byte y el segundo byte de la entrada para guardarnos en sus respectivas variables. Estos dos bytes se combinan en un solo valor de 16 bits que representa el número del siguiente cluster. Dependiendo de si el número de clúster es par o impar, los bits en estos dos bytes se ensamblan de manera diferente para obtener el número del siguiente clúster.

Si el número de clúster actual es par, el siguiente clúster se obtiene combinando los 4 bits menos significativos del segundo byte con los 8 bits del primer byte. Si el número de clúster actual es impar, el siguiente clúster se obtiene combinando los 4 bits más significativos del primer byte con los 8 bits del segundo byte. Por ejemplo, si el número de clúster actual es par lo que hacemos es agarrar el segundo byte, hacerle una operación AND con 0X0F (00001111) lo que nos devuelve los 4 bits menos significativos. Luego desplazamos 8 veces hacia la izquierda (0000111100000000) y por último con la operación OR combinamos lo anterior con el primer byte. De esta manera obtenemos el número completo del siguiente clúster.

```
unsigned short get_next_cluster(unsigned short current_cluster, unsigned char *fat_table)
{
    unsigned fat_offset = current_cluster + (current_cluster / 2); // Offset
    unsigned short next_cluster; // Siguiente cluster
    unsigned char fat_entry[2]; // Entrada de la tabla FAT

    fat_entry[0] = fat_table[fat_offset]; // Primer byte
    fat_entry[1] = fat_table[fat_offset + 1]; // Segundo byte

    if (current_cluster % 2 == 0)
    {
        next_cluster = ((fat_entry[1] & 0x0F) << 8) | fat_entry[0]; // Ensamble si es par
    }
    else
    {
        next_cluster = (fat_entry[1] << 4) | ((fat_entry[0] & 0xF0) >> 4); // Ensamble si es impar
    }

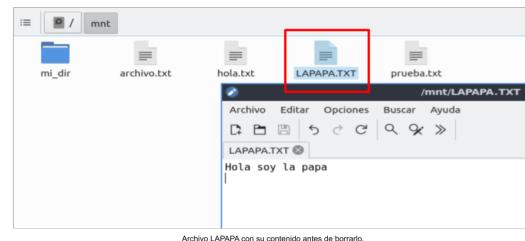
    return next_cluster;
}
```

Fuente: <http://www.cse.yorku.ca/~mccoll/courses/4300/cluster.html>

(Fig. 41)

Ahora para probar el funcionamiento del código de crear un archivo, en nuestro caso llamado **LAPAPA.TXT** con

un contenido dentro. (Fig. 42)



Archivo LAPAPA con su contenido antes de borrar.

(Fig. 42)

Procedemos a borrar el archivo y ahora podemos visualizar mediante nuestro programa `read_root` si el archivo se encuentra borrado. (Fig. 43)

```
alumno@alumno-virtualbox:~/Desktop/entregable/punto 4$ ./read_root
Partition type: 1
Encontrado FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATS
Root dir_entries 512
Directory: [MI_DTR ]
File: [HOLA .TXT]
File: [PRUEBA .TXT]
Deleted file: [LAPAPA .TXT] ←
File: [ARCHIVO .TXT]
Deleted file: [FORRAR~1SWX]
```

Salida por pantalla de read\_root

(Fig. 43)

Una vez borrado el archivo podemos ejecutar nuestro programa `recovery_file` con el comando `./recovery_file` para recuperar nuestro archivo. Y nuevamente con el programa `read_root` podemos ver si el archivo se encuentra nuevamente. (Fig. 44)

```
alumno@alumno-virtualbox:~/Desktop/entregable/punto 4$ ./recovery_file
Partition type: 1
Encontrado FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATS
Root dir_entries 512
Deleted file: [LAPAPA .TXT] ←
Archivo recuperado exitosamente.
```

Archivo restaurado mediante recovery\_file()

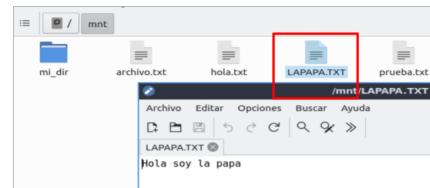
(Fig. 44)

```
alumno@alumno-virtualbox:~/Desktop/entregable/punto 4$ ./read_root
Partition type: 1
Encontrado FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATS
Root dir_entries 512
Directory: [MI_DTR ]
File: [HOLA .TXT]
File: [PRUEBA .TXT]
File: [LAPAPA .TXT] ←
File: [ARCHIVO .TXT]
Deleted file: [FORRAR~1SWX]
```

Salida por pantalla de read\_root después de recuperar

(Fig. 45)

Por último podemos ir a nuestro explorador de archivos a verificar si se recuperó efectivamente el archivo.



Archivo LAPAPA.txt recuperado junto a su contenido desde el explorador de archivos

(Fig. 46)

Cabe destacar que cuando realizamos el proceso de borrar el archivo y mostrarlo mediante el programa `read_root` puede tardar aproximadamente un minuto. De la misma manera cuando restauramos y queremos mostrar con `read_root` que se recuperó. Por lo tanto las acciones habría que realizarlas teniendo en cuenta lo anterior para ver el correcto funcionamiento.