



SISTEMAS OPERATIVOS Y REDES II

Trabajo Práctico 1

Sistema de Archivos FAT

Profesores: Benjamin Chuquimango
Pablo Rodriguez

Grupo 2

Integrantes: Juan Farias
Pablo Igei Nakagawa
Ezequiel Ravignani
Carlos Caballero
Cristian Yoel Garay

Primer Semestre 2025

Índice

Respuestas	3
1. Montaje del Sistema de Archivos	3
2. Cargando el MBR	3
a. Mostrando el MBR con el Hex Editor	3
b. Leer los datos del MBR utilizando código C	4
c. Verificar si la primera partición es booteable	6
d. Mostrar información de la primera partición	6
3. Cargando la Tabla de Archivos	7
a. Crear un archivo y mostrarlo	7
b. Crear y borrar un archivo	12
c. Mostrar archivos borrados	12
d. Recupero de archivos	12
4. Leyendo Archivos	20
a. Crear un archivo y mostrarlo	20
b. Mostrar el contenido de un archivo no borrado	21
c. Recuperar un archivo borrado	23
Repositorio	23

Respuestas

1. Montaje del Sistema de Archivos

Umask se refiere a la máscara de permisos que se aplicarán a los archivos y directorios creados en el sistema de archivo.

Unmask define qué permisos se deshabilitan de forma predeterminada cuando se crean archivos o directorios, estos permisos se definen en tres tipos de accesos lectura(r), escritura(w) y ejecución(x), y se asignan tres grupos: propietarios, grupos y otros.

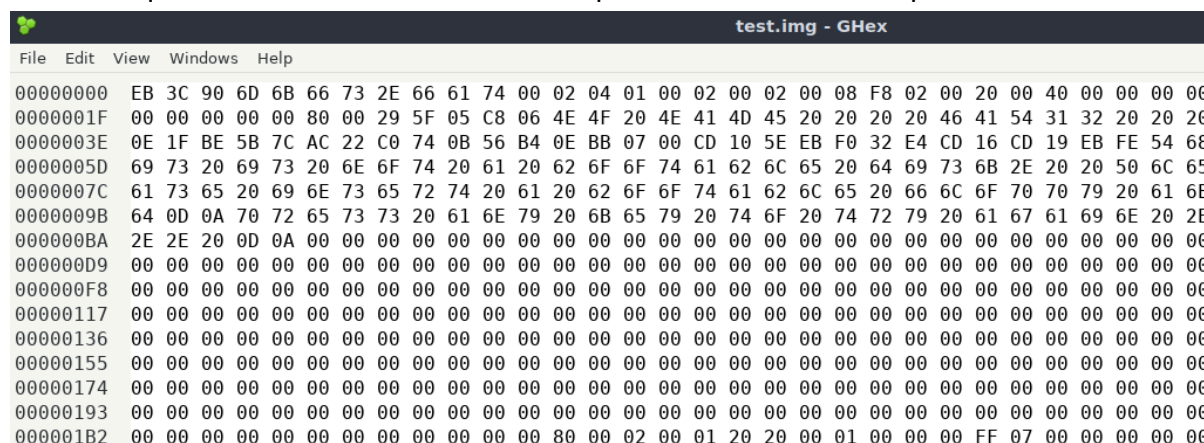
umask=000 significa que no se desactiva ningún permiso, los archivos y directorios creados tendrán los permisos máximos posibles sin restricciones.

2. Cargando el MBR

a. Mostrando el MBR con el Hex Editor

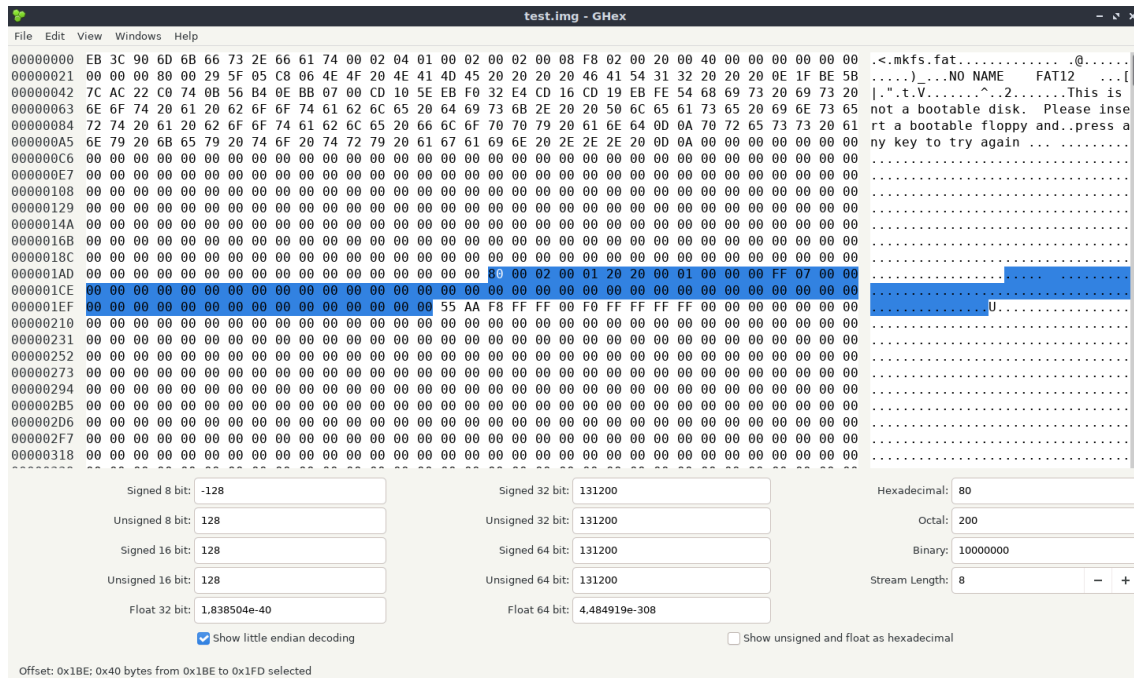
El MBR (Master boot record) es una pequeña parte del área de almacenamiento ubicada en el primer sector del disco duro, que contiene información importante para arrancar el sistema operativo. Está constituida por un tamaño fijo de 512 bytes donde del 0 al 445 se encuentra el boot code, del 446 al 509 las tablas de particiones y finalmente del 510 a 511 el signaturevalue.

Sabemos que la MBR se encuentra desde la posición 0x000 hasta la posición 0x1FF



00000000	EB 3C 90 6D 6B 66 73 2E 66 61 74 00 02 04 01 00 02 00 02 00 08 F8 02 00 20 00 40 00 00 00 00
0000001F	00 00 00 00 00 80 00 29 5F 05 C8 06 4E 4F 20 4E 41 4D 45 20 20 20 20 46 41 54 31 32 20 20 20
0000003E	0E 1F BE 5B 7C AC 22 C0 74 0B 56 B4 0E BB 07 00 CD 10 5E EB F0 32 E4 CD 16 CD 19 EB FE 54 68
0000005D	69 73 20 69 73 20 6E 6F 74 20 61 20 62 6F 6F 74 61 62 6C 65 20 64 69 73 6B 2E 20 20 50 6C 65
0000007C	61 73 65 20 69 6E 73 65 72 74 20 61 20 62 6F 6F 74 61 62 6C 65 20 66 6C 6F 70 70 79 20 61 6E
0000009B	64 0D 0A 70 72 65 73 73 20 61 6E 79 20 6B 65 79 20 74 6F 20 74 72 79 20 61 67 61 69 6E 20 2E
000000BA	2E 2E 20 0D 0A 00
000000D9	00 00
000000F8	00 00
00000117	00 00
00000136	00 00
00000155	00 00
00000174	00 00
00000193	00 00
000001B2	00 00 00 00 00 00 00 00 00 00 00 00 80 00 02 00 01 20 20 00 01 00 00 00 FF 07 00 00 00 00

Por lo dicho anteriormente sabemos que la tabla de particiones se encuentra del byte 446 al 509, es decir en hexadecimal de la posición 0x1BE hasta la posición 0x1FD.



- Entrada de Partición 1 (Bytes 0x01BE - 0x01CD): La primera entrada de partición se encuentra en las direcciones de memoria que van desde 0x01BE a 0x01CD.
- Entrada de Partición 2 (Bytes 0x01CE - 0x01DD): La segunda entrada de partición se encuentra en las direcciones de memoria que van desde 0x01CE a 0x01DD.
- Entrada de Partición 3 (Bytes 0x01DE - 0x01ED): La tercera entrada de partición se encuentra en las direcciones de memoria que van desde 0x01DE a 0x01ED.
- Entrada de Partición 4 (Bytes 0x01EE - 0x01FD): La cuarta entrada de partición se encuentra en las direcciones de memoria que van desde 0x01EE a 0x01FD.
- Teniendo en cuenta la información dada, solamente hay una partición que se encuentra en la primera entrada desde 0x01BE hasta 0x01CD, ya que las demás particiones poseen todos ceros.

b. Leer los datos del MBR utilizando código C

Para que el programa lea y muestre las tablas de particiones, primero se tiene que completar en el fseek con 0x1BE para que el puntero se mueva al byte 0x1BE que es donde comienza la tabla de particiones en el MBR después de los 446 bytes del bootstrap.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      FILE * in = fopen("test.img", "rb");
6      unsigned int i, start_sector, length_sectors;
7
8      fseek(in, 0x1BE , SEEK_SET); // Voy al inicio. Completar donde dice ...
9
10     for(i=0; i<4; i++) { // Leo las entradas
11         printf("Partition entry %d: First byte %02X\n", i, fgetc(in));
12         printf("  Comienzo de partición en CHS: %02X:%02X:%02X\n", fgetc(in), fgetc(in), fgetc(in));
13         printf("  Partition type %02X\n", fgetc(in));
14         printf("  Fin de partición en CHS: %02X:%02X:%02X\n", fgetc(in), fgetc(in), fgetc(in));
15
16         fread(&start_sector, 4, 1, in);
17         fread(&length_sectors, 4, 1, in);
18         printf("  Dirección LBA relativa 0x%08X, de tamaño en sectores %d\n", start_sector, length_sectors);
19     }
20
21     fclose(in);
22     return 0;
23 }

```

Luego compilamos y ejecutamos el programa. Para correr el programa escribimos por consulta `./read_mbr` lo cual nos mostrará por pantalla los datos correspondiente que se pide en el enunciado.

```

alumno@alumno-virtualbox:~/Descargas/TP1$ ./read_mbr
Partition entry 0: First byte 80
  Comienzo de partición en CHS: 00:02:00
  Partition type 0x01
  Fin de partición en CHS: 00:20:20
  Dirección LBA relativa 0x00000001, de tamaño en sectores 2047
Partition entry 1: First byte 00
  Comienzo de partición en CHS: 00:00:00
  Partition type 0x00
  Fin de partición en CHS: 00:00:00
  Dirección LBA relativa 0x00000000, de tamaño en sectores 0
Partition entry 2: First byte 00
  Comienzo de partición en CHS: 00:00:00
  Partition type 0x00
  Fin de partición en CHS: 00:00:00
  Dirección LBA relativa 0x00000000, de tamaño en sectores 0
Partition entry 3: First byte 00
  Comienzo de partición en CHS: 00:00:00
  Partition type 0x00
  Fin de partición en CHS: 00:00:00
  Dirección LBA relativa 0x00000000, de tamaño en sectores 0

```

c. Verificar si la primera partición es bootable

Para determinar si la partición es bootable, debemos mirar el primer byte que es el de estado y nos indica si es o no bootable. Si comienza con:

- 0x00 indica que la partición no es bootable
- 0x80 indica la partición es bootable

000001AD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 02 00 01 20 20 00 01 00 00 00 FF 07 00 00 .
000001CE 00 .
000001EF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA F8 FF FF 00 F0 FF FF FF FF 00 00 00 00 00 00 00 .
00000210 00 .
00000231 00 .
00000252 00 .
00000273 00 .
00000294 00 .
000002B5 00 .
000002D6 00 .
000002F7 00 .
00000318 00 .
00000339 00 .

Signed 8 bit:	0	Signed 32 bit:	0
Unsigned 8 bit:	0	Unsigned 32 bit:	0
Signed 16 bit:	0	Signed 64 bit:	0
Unsigned 16 bit:	0	Unsigned 64 bit:	0
Float 32 bit:	0.000000e+00	Float 64 bit:	0.000000e+00

☒ Show little endian decoding ☐ Show unsigned and float as

Offset: 0x1CD: 0x10 bytes from 0x1BE to 0x1CD selected

En este caso la partición es bootable debido a que el primer byte es 80.

d. Mostrar información de la primera partición

Para mostrar la información de la primera partición en esta caso se modifico el archivo read_mbr haciendo que solamente muestre la primera entrada de partición en vez que recorra todas las tablas de particiones.

Código:

```
#include <stdio.h>
#include <stdlib.h>

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

    fseek(in, 0x01BE , SEEK_SET);

    for(i=0; i<1; i++) { // Leo una entrada
        printf("Partition entry %d: First byte %02X\n", i, fgetc(in));
        printf("  Comienzo de partición en CHS: %02X:%02X:%02X\n",
fgetc(in), fgetc(in), fgetc(in));
        printf("  Partition type 0x%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);
        fread(&length_sectors, 4, 1, in);
        printf("    Dirección LBA relativa 0x%08X, de tamaño en sectores
%d\n", start_sector, length_sectors);
    }

    fclose(in);
    return 0;
}

```

Luego compilamos y ejecutamos el programa:

```

alumno@alumno-virtualbox:~/Descargas/FAT12$ ls
Makefile read_boot.c read_mbr read_mbr.c read_root.c test.img TP1_FileSystem_2025_1s.pdf
alumno@alumno-virtualbox:~/Descargas/FAT12$ gcc read_mbr.c -o read_mbr
alumno@alumno-virtualbox:~/Descargas/FAT12$ ./read_mbr
Partition entry 0: First byte 80
Comienzo de partición en CHS: 00:02:00
Partition type 0x01
Fin de partición en CHS: 00:20:20
Dirección LBA relativa 0x00000001, de tamaño en sectores 2047
alumno@alumno-virtualbox:~/Descargas/FAT12$ █

```

3. Cargando la Tabla de Archivos

a. Crear un archivo y mostrarlo

Para mostrar cuantos y cuales archivos tiene el filesystem primeros debemos determinar en cual byte comienza el directorio raíz, lo cual implica analizar el boot sector. Par esto miraremos:

- El número de sectores reservados: esto se encuentra en los bytes 14 y 15 del boot sector, en nuestro caso 1.
- El número de sectores por FAT: se encuentra en los bytes 22 y 23, en nuestro caso es 2. En FAT12, hay dos copias de la tabla, por lo que el número total de sectores ocupados para almacenar la tabla FAT es igual al número de copias de la tabla FAT multiplicado por el número de sectores por FAT, finalmente es $2 \times 2 = 4$.

Ya teniendo el número de sectores reservados y el número de sectores por FAT, calculamos el número del sector que contiene el directorio raíz. Este valor se obtiene sumando los sectores reservados y los sectores utilizados por FAT. En este caso $1 + 4 = 5$. Por lo tanto el directorio raíz comienza en el sector 5

Finalmente calculamos la dirección del primer byte del directorio raíz, para esto multiplicamos el número del sector por el tamaño de cada sector (512 bytes) $5 \times 512 = 2560$ bytes. Este valor es el byte donde comienza el directorio raíz.



Cada entrada del directorio raíz ocupa 32 bytes. Para identificar cuales archivos son válidos revisaremos el primer byte de cada entrada del directorio. Una entrada es válida si su primer byte no es 0xE5 ni tampoco 0x00 ya que indica el final del directorio.

En la imagen montada tenemos 6 entradas que son válidas ya que no comienzan con 0xE5 y 4 entradas que no son válidas porque comienzan con 0xE5

Código:

```
#include <stdlib.h>
#include <stdio.h>

typedef struct {
    unsigned char first_byte;
    unsigned char start_chs[3];
    unsigned char partition_type;
    unsigned char end_chs[3];
    char starting_cluster[4];
    char file_size[4];
} __attribute__((packed)) PartitionTable;

typedef struct {
    unsigned char jmp[3];
    char oem[8];
    unsigned short sector_size;
    unsigned char sector_cluster;
    unsigned short reserved_sectors;
    unsigned char number_of_fats;
```



```

unsigned short root_dir_entries;
unsigned short sector_volumen;
unsigned char descriptor;
unsigned short fat_size_sectors;
unsigned short sector_track;
unsigned short headers;
unsigned int sector_hidden;
unsigned int sector_partition;
unsigned char physical_device;
unsigned char current_header;
unsigned char firm;
unsigned int volume_id;
char volume_label[11];
char fs_type[8];
char boot_code[448];
unsigned short boot_sector_signature;
} __attribute__((packed)) Fat12BootSector;

typedef struct {
    unsigned char filename[8];    // Nombre del archivo (8 caracteres)
    unsigned char extension[3];  // Extensión del archivo (3 caracteres)
    unsigned char attributes;    // Atributos del archivo
    unsigned char reserved;      // Reservado (1 byte)
    unsigned char created_time_seconds;
    unsigned short created_time;
    unsigned short created_date;
    unsigned short accessed_date;
    unsigned short cluster_highbytes_address;
    unsigned short written_time;
    unsigned short written_date;
    unsigned short cluster_lowbytes_address;
    unsigned int size_of_file;    // Tamaño del archivo
} __attribute__((packed)) Fat12Entry;

void print_file_info(Fat12Entry *entry) {
    switch(entry->filename[0]) {
        case 0x00:
            return; // Entrada sin usar
        case 0xE5:

```

```

        printf("Archivo borrado: [?.8s.%.3s]\n", entry->filename+1,
entry->extension);
        return;
    case 0x05:
        printf("Archivo que comienza con 0xE5: [c%.7s.%.3s]\n", 0xE5,
entry->filename+1, entry->extension);
        break;
    default:
        switch(entry->attributes) {
            case 0x10:
                printf("Directorio: [?.8s.%.3s]\n", entry->filename,
entry->extension);
                return;
            case 0x20:
                printf("Archivo: [?.8s.%.3s]\n", entry->filename,
entry->extension);
                return;
        }
    }
}

int main() {
    FILE * in = fopen("test.img", "rb");
    int i;
    PartitionTable pt[4];
    Fat12BootSector bs;
    Fat12Entry entry;
c
    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++) {
        if(pt[i].partition_type == 1) {
            printf("Encontrada particion FAT12 %d\n", i);
            break;
        }
    }

    if(i == 4) {
        printf("No encontrado filesystem FAT12, saliendo...\n");
    }
}

```

```

        return -1;
    }

    fseek(in, 0, SEEK_SET);
    fread(&bs, sizeof(Fat12BootSector), 1, in);
    //{...} Leo boot sector

    printf("En 0x%lX, sector size %d, FAT size %d sectors, %d FATs\n\n",
           ftell(in), bs.sector_size, bs.fat_size_sectors,
           bs.number_of_fats);

    fseek(in, (bs.reserved_sectors-1 + bs.fat_size_sectors *
           bs.number_of_fats) *
           bs.sector_size, SEEK_CUR);

    printf("Root dir_entries %d \n", bs.root_dir_entries);
    for(i=0; i<bs.root_dir_entries; i++) {
        fread(&entry, sizeof(entry), 1, in);
        print_file_info(&entry);
    }

    printf("\nLeido Root directory, ahora en 0x%lX\n", ftell(in));
    fclose(in);
    return 0;
}

```

Procedimiento y resultado en la terminal:

The screenshot shows a terminal window titled 'alumno@alumno-virtualbox: ~/Descargas/FAT12'. The user runs the command 'gcc read_root.c -o read_root' and then './read_root'. The output shows the program finding the FAT12 partition at sector 0, with a sector size of 512 and 2 FATs. It then prints the root directory entries, including a directory entry for 'MI_DIR' and several file entries, some of which are marked as deleted. Finally, it prints the current file pointer position as 0x4A00.

```

alumno@alumno-virtualbox:~/Descargas/FAT12$ gcc read_root.c -o read_root
alumno@alumno-virtualbox:~/Descargas/FAT12$ ./read_root
Encontrada particion FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATs

Root dir_entries 512
Directorio: [MI_DIR . ]
Archivo: [HOLA .TXT]
Archivo: [PRUEBA .TXT]
Archivo borrado: [?b.]
Archivo borrado: [?ORRADO T.TXT]
Archivo borrado: [?..]
Archivo borrado: [?ORRAR-1S.SWX]

Leido Root directory, ahora en 0x4A00
alumno@alumno-virtualbox:~/Descargas/FAT12$

```

b. Crear y borrar un archivo

Primero montamos la imagen en el sistema

```
alumno@alumno-virtualbox:~$ sudo mount test.img /mnt -o loop,umask=000
```

Luego creamos un archivo utilizando el comando touch, en esta caso el archivo se llamará tp1sor2.txt

```
alumno@alumno-virtualbox:~$ touch /mnt/tp1sor2.txt
alumno@alumno-virtualbox:~$ ls /mnt
hola.txt  mi_dir  prueba.txt  tp1sor2.txt
```

Luego borramos el archivo con el comando rm:

```
alumno@alumno-virtualbox:/mnt$ rm tp1sor2.txt
alumno@alumno-virtualbox:/mnt$ ls /mnt
hola.txt  mi_dir  prueba.txt
```

Muestra de como se observa en nuestro programa en C el archivo borrado:

```
Encontrada particion FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATs

Root dir_entries 512
Directorio: [MI_DIR . ]
Archivo: [HOLA .TXT]
Archivo: [PRUEBA .TXT]
Archivo borrado: [?t.]
Archivo borrado: [?P1SOR2 T.TXT]
Archivo borrado: [?..]
Archivo borrado: [?ORRAR~1S.SWX]
```

c. Mostrar archivos borrados

Para visualizar el archivo borrado en GHEX primero debemos buscar el directorio raíz (ver 3a calculo para sacar el byte del directorio raíz), una vez en el, debemos mirar las entradas, cada entrada tiene 32 bytes y fijarnos si el primer byte comienza con 0xE5, esto indica que el archivo a sido borrado. Para saber si el archivo borrado es el nuestro, miramos la tabla ascii si corresponde al nombre que se le había puesto.

```
00000AB3 51 00 00 E2 35 46 51 04 00 1C 00 00 00 E5 74 00 70 00 31 00 73 00 6F 00 0F 00 A1 72 00 32 00 2E 00 Q...5FQ.....t.p.l.s.o....r.2...
00000AD4 74 00 78 00 74 00 00 00 00 00 FF FF E5 50 31 53 4F 52 32 20 54 58 54 20 00 44 17 92 7B 5A 7B 5A 00 t.x.t.....P1SOR2 TXT .D..{Z{Z
00000AF5 00 17 92 7B 5A 00 00 00 00 00 00 E5 2E 00 62 00 6F 00 72 00 72 00 0F 00 A9 61 00 72 00 6F 00 6E 00 ..{Z.....b.o.r.r.a.r.o.n.
00000B16 2E 00 74 00 00 00 78 00 74 00 E5 4F 52 52 41 52 7E 31 53 57 58 20 00 00 F3 B4 7B 4A 7B 4A 00 00 F3 ..t...x.t..ORRAR~1SWX .....{J{J...
00000B37 B4 7B 4A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .{J.....
```

d. Recupero de archivos

La recuperación de archivos en FAT12 se basa en la estructura de la tabla FAT y las entradas del directorio. Cuando un archivo es borrado solo se borra la referencia a memoria en el directorio y se marca como 0xE5 indicando como espacio libre en la FAT, pero los datos siguen en el disco. La tabla FAT guarda la ubicación de los clusters de datos del archivo. Para recuperar un archivo, se buscan los cluster del archivo en la FAT. Si los clusters no han sido sobrescritos la recuperación es posible. Sin embargo si han sido sobrescritas se vuelve imposible. Para la recuperación del archivo, se localiza la entrada de directorio, se identifica el

primer cluster del archivo y se sigue la cadena de cluster en la tabla FAT. Luego se extraen los datos de esos clusters. Por último se restaura el archivo.

Programa para la recuperación de archivos (recuperar_archivo.c).

Código:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

typedef struct {
    unsigned char jmp[3];
    char oem[8];
    unsigned short bytes_per_sector;
    unsigned char sectors_per_cluster;
    unsigned short reserved_sectors;
    unsigned char num_fats;
    unsigned short root_dir_entries;
    unsigned short total_sectors;
    unsigned char media_descriptor;
    unsigned short fat_size_sectors;
    unsigned short sectors_per_track;
    unsigned short num_heads;
    unsigned int hidden_sectors;
    unsigned int partition_sectors;
    unsigned char physical_drive;
    unsigned char boot_signature;
    unsigned int volume_id;
    char volume_label[11];
    char fs_type[8];
} __attribute__((packed)) BootSector;

typedef struct {
    unsigned char filename[8];
    unsigned char extension[3];
    unsigned char attributes;
    unsigned char reserved[10];
    unsigned short creation_time;
    unsigned short creation_date;
    unsigned short first_cluster;
```

```

    unsigned int file_size;
} __attribute__((packed)) DirEntry;

typedef struct {
    long entry_position;
    DirEntry entry;
    uint16_t *clusters; // Cadena de clusters del archivo
    int cluster_count;
} RecoverableFile;

// Función para leer la tabla FAT
uint16_t* read_fat(FILE *disk, BootSector *bs, int *error) {
    uint16_t *fat = NULL;
    size_t fat_size = bs->fat_size_sectors * bs->bytes_per_sector;

    fat = (uint16_t*)malloc(fat_size);
    if (!fat) {
        *error = 1;
        return NULL;
    }

    fseek(disk, bs->reserved_sectors * bs->bytes_per_sector, SEEK_SET);
    if (fread(fat, 1, fat_size, disk) != fat_size) {
        free(fat);
        *error = 1;
        return NULL;
    }

    *error = 0;
    return fat;
}

// Función para seguir la cadena de clusters
int get_cluster_chain(uint16_t *fat, uint16_t start_cluster, uint16_t
**chain, int *cluster_count) {
    uint16_t current_cluster = start_cluster;
    int capacity = 10;
    int count = 0;

    *chain = (uint16_t*)malloc(capacity * sizeof(uint16_t));

```

```

    if (!*chain) return 1;

    while (current_cluster < 0xFF8) { // Mientras no sea el último cluster
        if (current_cluster == 0xFF7) { // Cluster defectuoso
            free(*chain);
            return 2;
        }

        if (count >= capacity) {
            capacity *= 2;
            uint16_t *temp = realloc(*chain, capacity * sizeof(uint16_t));
            if (!temp) {
                free(*chain);
                return 1;
            }
            *chain = temp;
        }

        (*chain)[count++] = current_cluster;
        current_cluster = fat[current_cluster];
    }

    *cluster_count = count;
    return 0;
}

// Función para verificar clusters
int verify_clusters(FILE *disk, BootSector *bs, uint16_t *chain, int
cluster_count) {
    unsigned char *buffer = (unsigned char*)malloc(bs->bytes_per_sector *
bs->sectors_per_cluster);
    if (!buffer) return 1;

    int result = 0;
    for (int i = 0; i < cluster_count; i++) {
        // Calcular posición del cluster en el disco
        long position = ((bs->reserved_sectors + bs->num_fats *
bs->fat_size_sectors +
                        (bs->root_dir_entries * 32 + bs->bytes_per_sector
- 1) / bs->bytes_per_sector) +

```

```

        (chain[i] - 2) * bs->sectors_per_cluster) *
bs->bytes_per_sector;

    fseek(disk, position, SEEK_SET);
    if (fread(buffer, 1, bs->bytes_per_sector *
bs->sectors_per_cluster, disk) !=
        bs->bytes_per_sector * bs->sectors_per_cluster) {
        result = 1;
        break;
    }
}

free(buffer);
return result;
}

int main() {
    FILE *disk = fopen("test.img", "rb+");
    if (!disk) {
        printf("Error al abrir la imagen de disco.\n");
        return 1;
    }

    BootSector bs;
    fread(&bs, sizeof(BootSector), 1, disk);

    // Leer la tabla FAT
    int error;
    uint16_t *fat = read_fat(disk, &bs, &error);
    if (error) {
        printf("Error al leer la FAT.\n");
        fclose(disk);
        return 1;
    }

    // Leer el directorio raíz
    long root_dir_position = (bs.reserved_sectors + bs.num_fats *
bs.fat_size_sectors) * bs.bytes_per_sector;
    fseek(disk, root_dir_position, SEEK_SET);

```



```

RecoverableFile files[100];
int file_count = 0;

// Escanear entradas del directorio
for (int i = 0; i < bs.root_dir_entries; i++) {
    DirEntry entry;
    long current_position = ftell(disk);
    fread(&entry, sizeof(DirEntry), 1, disk);

    // Buscar archivos borrados (0xE5) o entradas no usadas (0x00)
    if (entry.filename[0] == 0xE5 || entry.filename[0] == 0x00) {
        if (entry.filename[0] == 0xE5 && entry.first_cluster != 0) {
            // Verificar cadena de clusters
            uint16_t *cluster_chain = NULL;
            int cluster_count = 0;
            int res = get_cluster_chain(fat, entry.first_cluster,
&cluster_chain, &cluster_count);

            if (res == 0 && cluster_count > 0) {
                // Verificar si los clusters están intactos
                if (verify_clusters(disk, &bs, cluster_chain,
cluster_count) == 0) {
                    files[file_count].entry_position =
current_position;

                    memcpy(&files[file_count].entry, &entry,
sizeof(DirEntry));

                    files[file_count].clusters = cluster_chain;
                    files[file_count].cluster_count = cluster_count;
                    file_count++;

                    printf("%d. Archivo recuperable: [?.8s.%.3s]
Tamaño: %u bytes, Clusters: %d\n",
                        file_count, entry.filename+1,
entry.extension, entry.file_size, cluster_count);
                } else {
                    free(cluster_chain);
                }
            }
        }
        continue;
    }
}

```

```

    }

}

// Menú de recuperación
if (file_count > 0) {
    printf("\nSe encontraron %d archivos recuperables.\n", file_count);
    printf("Ingrese el número del archivo a recuperar (0 para salir):
");

    int option;
    scanf("%d", &option);

    if (option > 0 && option <= file_count) {
        printf("Ingrese el primer carácter del nombre original: ");
        char first_char;
        scanf(" %c", &first_char);

        // Restaurar entrada en el directorio
        files[option-1].entry.filename[0] = first_char;
        fseek(disk, files[option-1].entry_position, SEEK_SET);
        fwrite(&files[option-1].entry, sizeof(DirEntry), 1, disk);
        fflush(disk);

        printf("Archivo recuperado: [%c%.7s%.3s]\n", first_char,
            files[option-1].entry.filename+1,
files[option-1].entry.extension);
    }
} else {
    printf("\nNo se encontraron archivos recuperables.\n");
}

// Liberar memoria
for (int i = 0; i < file_count; i++) {
    free(files[i].clusters);
}
free(fat);
fclose(disk);
return 0;
}

```

Explicaciones y resultados en la terminal:

Primero leemos qué archivos hay borrados en la imagen compilando y ejecutando el programa read_root. Como vemos en la imagen hay 4 archivos borrados, pero luego al ejecutar el programa para recuperar archivos se podrá saber qué archivos son recuperables.

```
Encontrada particion FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATs

Root dir_entries 512
Directorio: [MI_DIR . ]

Archivo: [HOLA .TXT] su contenido es: Hola, bienvenidos !
Pueden ahora tratar de leerme desde c !

Archivo: [PRUEBA .TXT] su contenido es: este es un archivo de prueba
Archivo borrado: [?b.]
Archivo borrado: [?ORRADO T.TXT]
Archivo borrado: [?..]
Archivo borrado: [?ORRAR~1S.SWX]

Leido Root directory, ahora en 0x4A00
alumno@alumno-virtualbox:~/Descargas/FAT12$
```

Al ejecutar el programa de recuperar archivos, se ve que solamente 1 archivo de 4 son recuperables, en este caso “?ORRADO.TXT”.

```
alumno@alumno-virtualbox:~/Descargas/FAT12$ gcc recuperar_archivo.c -o recuperar
alumno@alumno-virtualbox:~/Descargas/FAT12$ ./recuperar
1. Archivo recuperable: [?ORRADO T.TXT] Tamaño: 24 bytes, Clusters: 2

Se encontraron 1 archivos recuperables.
Ingrese el número del archivo a recuperar (0 para salir):
```

El programa pide ingresar el número del archivo a recuperar, en este caso ingresamos 1 ya que hay uno solo. Y luego nos pide ingresar el primer carácter para reemplazarlo y poder recuperar el archivo además de que será el primer carácter del nombre del archivo recuperado. En este caso ponemos la “B”, ya que deducimos que el archivo era “BORRADO.TXT”.

```
alumno@alumno-virtualbox:~/Descargas/FAT12$ gcc recuperar_archivo.c -o recuperar
alumno@alumno-virtualbox:~/Descargas/FAT12$ ./recuperar
1. Archivo recuperable: [?ORRADO T.TXT] Tamaño: 24 bytes, Clusters: 2

Se encontraron 1 archivos recuperables.
Ingrese el número del archivo a recuperar (0 para salir): 1
Ingrese el primer carácter del nombre original: B
Archivo recuperado: [BORRADO .TXT]
alumno@alumno-virtualbox:~/Descargas/FAT12$
```

Después al leer los archivos que contiene la imagen, vemos cómo se recuperó el archivo, ahora con nombre “BORRADO.TXT” por la “B” que ingresamos anteriormente y se muestra su contenido dentro.

```

alumno@alumno-virtualbox:~/Descargas/FAT12$ ./read_root
Encontrada particion FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATs

Root dir_entries 512
Directorio: [MI_DIR . ]

Archivo: [HOLA .TXT] su contenido es: Hola, bienvenidos !
Pueden ahora tratar de leerme desde c !

Archivo: [PRUEBA .TXT] su contenido es: este es un archivo de prueba
Archivo borrado: [?b.]
Archivo: [BORRADO .TXT] su contenido es: este archivo fue borrado
Archivo borrado: [?..]
Archivo borrado: [?ORRAR~1S.SWX]

Leido Root directory, ahora en 0x4A00
alumno@alumno-virtualbox:~/Descargas/FAT12$ █

```

4. Leyendo Archivos

a. Crear un archivo y mostrarlo

Primero montamos la imagen en el sistema

```

alumno@alumno-virtualbox:~$ sudo mount test.img /mnt -o loop,umask=000

```

Luego creamos un archivo utilizando el comando echo ya que va tener contenido dicho archivo, en esta caso el archivo contendrá un string "Prueba archivo la papa" con el nombre de archivo "lapapa.txt"

```

alumno@alumno-virtualbox:/mnt$ echo "Prueba archivo la papa" > lapapa.txt
alumno@alumno-virtualbox:/mnt$ ls
hola.txt  lapapa.txt  mi_dir  prueba.txt

```

Usando nuestro código read_root echo en lenguaje C, nos mostraría de la siguiente manera:

```

alumno@alumno-virtualbox:~/Descargas/TP1$ ./read_root
Encontrada particion FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATs

Root dir_entries 512
Directorio: [MI_DIR . ]
Archivo: [HOLA .TXT]
Archivo: [PRUEBA .TXT]
Archivo: [LAPAPA .TXT]
Archivo borrado: [?..]
Archivo borrado: [?ORRAR~1S.SWX]

Leido Root directory, ahora en 0x4A00

```

Podemos observar como el archivo creado aparece en el filesystem

En ghex lo identificamos:

Como dicho archivo no fue borrado el primer byte contendrá el valor hexadecimal de la primera letra del nombre del archivo

```
00000A92 74 00 78 00 74 00 00 00 00 00 FF FF FF FF 50 52 55 45 42 41 20 20 54 58 54 20 00 90 E2 35 46 51 46 t.x.t.....PRUEBA TXT ...5FQI
00000AB3 51 00 00 E2 35 46 51 04 00 1C 00 00 00 41 6C 00 61 00 70 00 61 00 70 00 0F 00 E3 61 00 2E 00 74 00 Q...5FQ.....Al.a.p.a.p....a...t
00000AD4 78 00 74 00 00 00 00 00 FF FF FF FF 4C 41 50 41 50 41 20 20 54 58 54 20 00 C4 57 15 7C 5A 7C 5A 00 x.t.....LAPAPA TXT ..W.|Z|Z
00000AF5 00 57 15 7C 5A 00 00 17 00 00 00 E5 2E 00 62 00 6F 00 72 00 72 00 0F 00 A9 61 00 72 00 6F 00 6E 00 .W.|Z.....b.o.r.....a.r.o.n
00000B16 2E 00 74 00 00 00 78 00 74 00 E5 4F 52 52 41 52 7E 31 53 57 58 20 00 00 F3 B4 7B 4A 7B 4A 00 00 F3 ..t...x.t...ORRAR-ISWX .....{J{J...
00000B37 B4 7B 4A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .{J.....
```

b. Mostrar el contenido de un archivo no borrado

Contenido del archivo mostrado en ghex:

```
000059D9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000059FA 00 00 00 00 00 00 65 73 74 65 20 65 73 20 75 6E 20 61 72 63 68 69 76 6F 20 64 65 20 70 72 75 65 62 .....este es un archivo de pruel
00005A1B 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 a.....
00005A3C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Para mostrar el contenido de los archivos del filesystem por un programa hecho en C, modificamos el `read_root`. El mismo recorre el directorio raíz, busca el primer cluster de cada archivo y lee los datos que le corresponde a dicho archivo.

Inicialmente se calcula la posición del primer cluster y su tamaño para luego pasarlo como parámetro a la función que se encargará de imprimir por pantalla el contenido del archivo.

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

fseek(in, position_root_directory, SEEK_CUR); // Vamos al inicio del root diretory

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

firstCluster = ftell(in) + (bs.root_dir_entries*sizeof(entry)); //Obtenemos la posicion del primer
                        //primer byte del primer cluster de datos

sizeofCluster = bs.sector_size * bs.sector_cluster;//Tamaño del cluster

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

Con respecto a la función `print_file_info()` dicha función sufre una modificación para que a la hora de leer el nombre de un archivo también llame a la función `leer()` para que muestre por pantalla el contenido de dicho archivo.

```

void print_file_info(Fat12Entry *entry, unsigned short firstCluster, unsigned short clusterSize)
{
    switch(entry->filename[0]){
        case 0x00:
            return;

        case 0xE5:
            printf("Archivo borrado: [%s.%s]\n", entry->filename+1, entry->extension);
            return;

        case 0x05:
            printf("Archivo que comienza con 0xE5: [%c%.7s.%s]\n", 0xE5, entry->filename+1, entry->extension);
            break;

        default:
            switch (entry->attributes)
            {
                case 0x10:
                    printf("Directorio: [%s.%s]\n\n", entry->filename, entry->extension);
                    return;

                case 0x20:
                    printf("Archivo: [%s.%s] su contenido es: ", entry->filename, entry->extension);
                    leer(firstCluster, entry->cluster lowbytes address, clusterSize, entry->size of file);
                    return;
            }
    }
}

```

Función leer()

```

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

    fseek(in, firstCluster + ((fileFirstCluster - 2) * clusterSize), SEEK_SET); //posicionamiento del primer cluster
    //del archivo

    fread(leer, fileSize, 1, in); //Lee el contenido del archivo y lo almacena en el array
    for (i = 0; i < fileSize; i++) //Recorre el array para imprimirlo por pantalla
    {
        printf("%c", leer[i]);
    }
    printf("\n");

    fclose(in);
}

```

Compilación del código:

```

alumno@alumno-virtualbox:~/Descargas/TP1$ ./read_root
Encontrada particion FAT12 0
En 0x200, sector size 512, FAT size 2 sectors, 2 FATs

Root dir_entries 512
Directorio: [MI_DIR . ]

Archivo: [HOLA .TXT] su contenido es: Hola, bienvenidos !
Pueden ahora tratar de leerme desde c !

Archivo: [PRUEBA .TXT] su contenido es: este es un archivo de prueba
Archivo: [LAPAPA .TXT] su contenido es: Prueba archivo la papa
Archivo borrado: [?..]
Archivo borrado: [?ORRAR~1S.SWX]

Leido Root directory, ahora en 0x4A00

```

c. Recuperar un archivo borrado

Está explicado en detalle en el punto 3 C, tanto el cómo se recupera un archivo, su código en C y sus resultados en la terminal.

Repositorio

Link: <https://github.com/Ging1991/SOR2>