Universidad Nacional de General Sarmiento

Trabajo Práctico 2

FAT 12

Sistemas Operativos y Redes II

Docentes:

Chuquimango Chilon Luis Benjamin

Echabarri Alan Pablo Daniel

Alumnos:

Lopez, Diego (*42823629/2019*)

Ruiz, Kevin (*41672607/2017*)

Sanchez Rodriguez, Camila (*41024628/2018*)

Santillán, Javier (*40226822/2019*)

Soria, Lucía (*43911246/2020*)

Fecha de entrega:

19 de abril de 2024

Índice

Introducción

Un sistema de archivos, conocido en inglés como *file system*, es una estructura fundamental de cualquier sistema informático. Dicho componente se encarga de organizar y gestionar los datos en los dispositivos, permitiendo el acceso y la administración de la información de manera efectiva. En este trabajo, nos centraremos en explorar profundamente el sistema de archivos *FAT12,* una de las versiones más antiguas del sistema de archivos *FAT.*

El sistema de archivos *FAT12(File Allocation Table 12)* fue desarrollado por Microsoft y una de las primeras aplicaciones fue en sistemas operativos como MS-DOS donde estableció bases para versiones posteriores de los sistemas de archivos FAT.

Se abordarán aspectos vitales de este sistema, indagando su estructura, la gestión del espacio en los dispositivos de almacenamiento, así como sus ventajas y desventajas. Para efectuar nuestro cometido, haremos uso de un archivo de imágen provisto por nuestros docentes, llamado *"test.img".* Además, utilizaremos un editor hexadecimal, en particular *"Ghex",* para poder analizar la información a un nivel más bajo y en un entorno controlado.

Desarrollo

Al montarlo ¿Para qué se ha puesto umask=000?

El montaje hace que los sistemas de archivos, los archivos, directorios y dispositivos estén disponibles para utilizarlos en una ubicación determinada. Es la única manera de que un sistema de archivos pueda ser accesible. El comando ***mount*** da instrucciones al sistema operativo de que conecte un *file system* a un directorio especificado. Por lo tanto, para montar nuestro archivo provisto, se utilizó el siguiente comando:

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

Dicho anteriormente, el comando ***mount*** es utilizado para montar la imagen del sistema de archivos. En este caso, especificamos que el punto de montaje local sea el directorio ***'/mnt'***, el cual es utilizado normalmente para montar dispositivos en el disco. Además, se indicó la utilización del *loopback*, el cual nos permite montar y acceder al archivo de imagen como si fuera un dispositivo de bloque real, similar a una partición de disco físico.

En segundo lugar, se emplea el comando ***umask***. El mismo es utilizado para el control de la máscara del modo de creación de archivos, la cual determina el valor inicial de permisos para los archivos recién creados. En otras palabras, la máscara de modo especifica los permisos que no se aplicarán a un archivo cuando este es creado. Actúa, entonces, como un filtro que elimina los bits de permiso, determinando los permisos predeterminados para los nuevos archivos. Puede determinarse mediante símbolos o valores en formato octal.

Valor en octal	Permiso
0	Lectura, escritura y ejecución.
1	Lectura y escritura.
2	Lectura y ejecución
3	Sólo lectura
4	Escritura y ejecución.
5	Sólo escritura.
6	Sólo ejecución
7	Sin permisos.

Por ejemplo, si la máscara es 027 entonces los archivos se crearán con permisos 750, y si la máscara es 007, entonces se crearán con permisos 770.

De esta forma, al incluir ***umask=000***, estamos otorgando permisos de lectura, escritura, y ejecución a todos los usuarios, lo que equivale a un permiso de 777. Esta configuración nos garantiza que, al montar la imagen, no encontraremos restricciones al leer o editar los archivos contenidos en ella.

Cargando el MBR

El **MBR** (*Master Boot Record*), también referido como sector de arranque, constituye el sector inicial de un dispositivo de almacenamiento, como un disco duro. Por lo general, se utiliza para el arranque de un sistema operativo almacenado en otros sectores del disco, pero también contiene una tabla de particiones y sirve para identificar un dispositivo de disco individual.

Usualmente, el MBR se localiza en los primeros 512 bytes del dispositivo de almacenamiento.Su estructura es la siguiente:

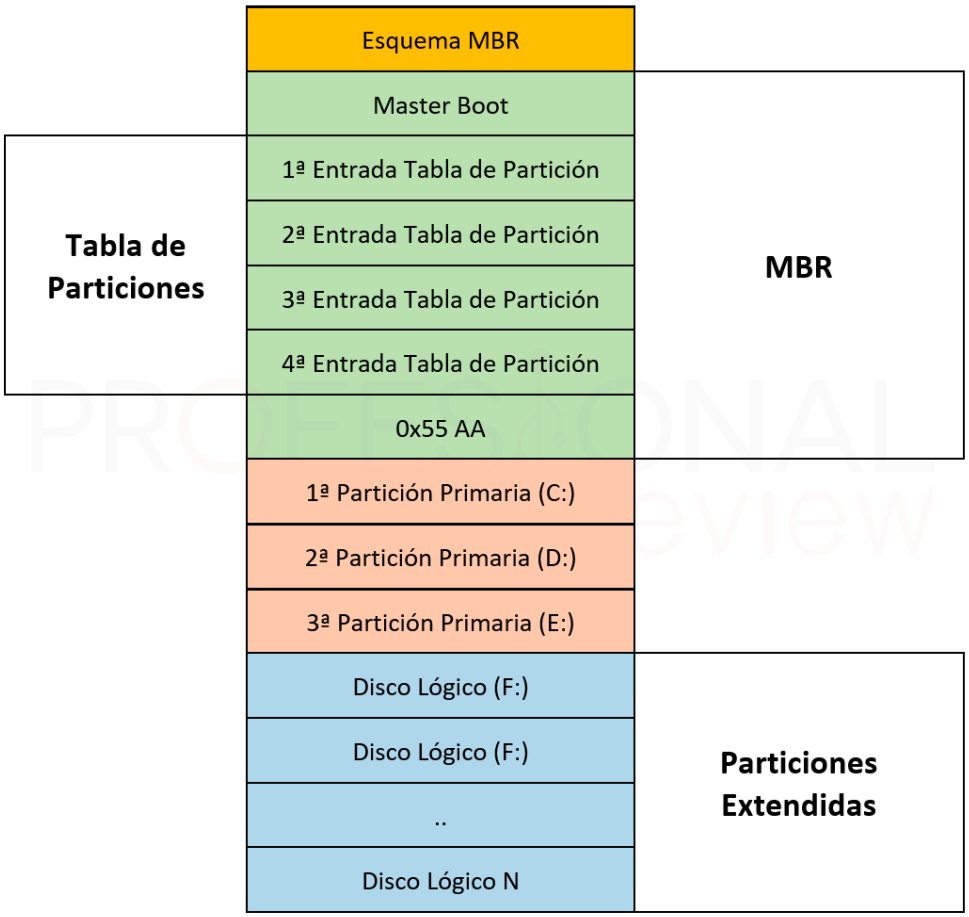
446 bytes	Código máquina
64 bytes	Tabla de particiones
2 bytes	Firma de MBR (0x55AA)

Este sector contiene las entradas primarias en la tabla de particiones. Por convención, el esquema de la Tabla de Particiones consta de exactamente 4 particiones primarias. Cada una de estas ocupa un total de 16 bytes. El formato de cada registro de la tabla de particiones es el siguiente:

Offset	Descripción
0x00	Estado. (Boot indicador bit flag: 0= no, 0x80 = booteable)
0x01	Cilindro, cabezal, sector del primer sector en la partición.
0x04	Tipo de partición.
0x05	Cilindro, cabezal, sector del último sector en la partición.
0x08	Logical block address del primer sector de la partición (4 bytes).
0x0C	Longitud de la partición en sectores (4 bytes).

Podemos afirmar, entonces, que dentro de cada registro de la tabla de particiones tenemos la dirección CHS de comienzo y fin de la partición, su estado, tipo, longitud y el sector donde comienza la misma. CHS es el modo de direccionamiento en MBR para los discos duros IDE. En la misma intervienen cilindro, cabezal y sector del disco.

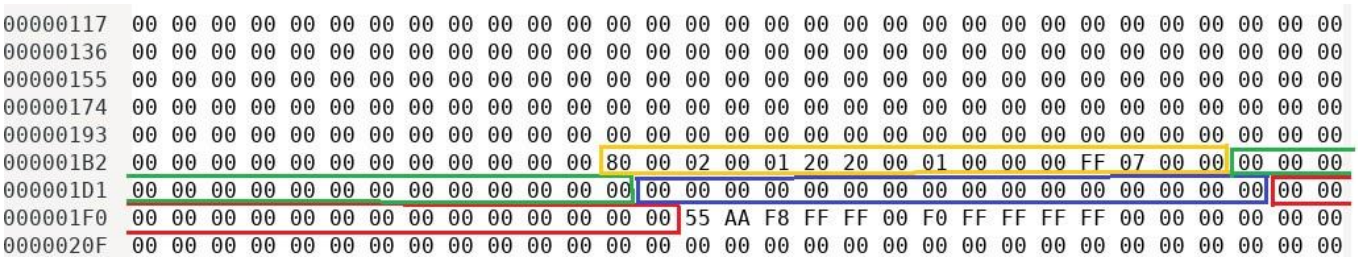
Las particiones primarias se encuentran a continuación del MBR, y luego se encuentran las particiones extendidas.



Para examinar los bytes correspondientes al MBR de nuestra imagen, debemos enfocarnos en los primeros 512 bytes. Para lograrlo haremos uso del editor hexadecimal *Hex Editor.*

Podemos notar que los últimos dos bytes del sector de arranque contienen la firma característica del MBR, que es 55AA.

Como se comentó anteriormente, el MBR contiene 4 *partition table entries,* cada una con un tamaño de 16 bytes. Por lo tanto, estas entradas se encuentran desde el byte 446 al 509, dado que el MBR posee un tamaño de 512 bytes y las particiones ocupan un total de 64 bytes. Para visualizarlo, haremos uso de nuestro editor hexadecimal.



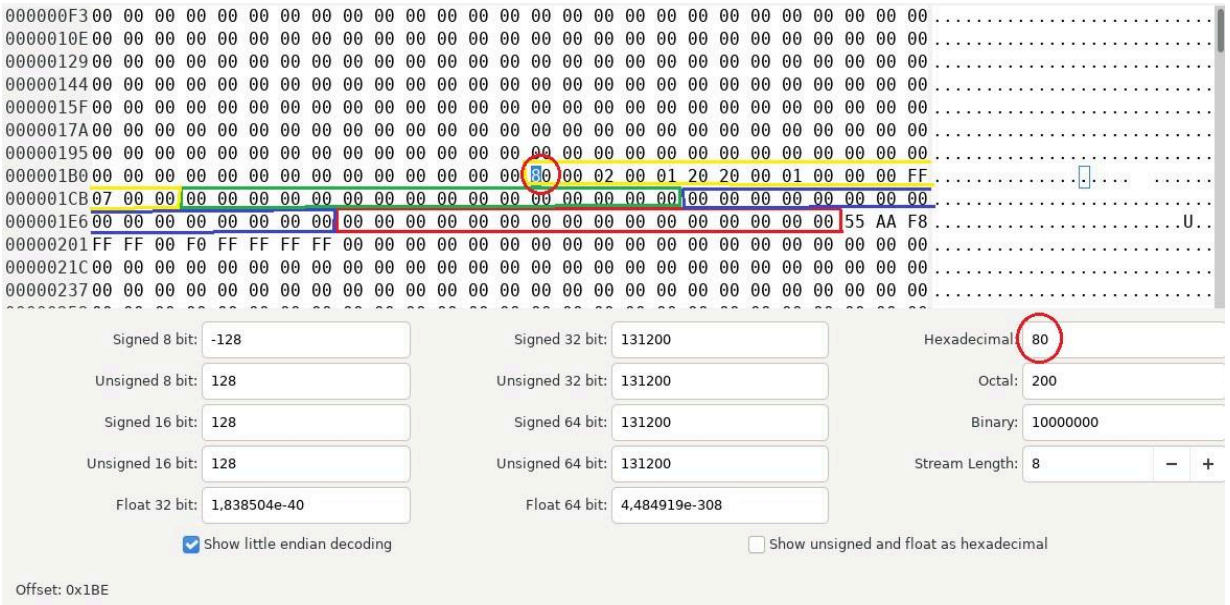
Es evidente que solo la primera partición, indicada con color amarillo, contiene datos. Por consiguiente, podemos inferir que, aunque la tabla de particiones consta de cuatro entradas, solo una está ocupada, como se puede observar en la imagen anterior.

Todos los datos mostrados en el editor hexadecimal también pueden ser leídos utilizando código en el lenguaje C, corriendo el archivo *read_boot.c*. El siguiente es el resultado obtenido:

```
alumno@alumno-virtualbox:~/tp1/repo/tp-fat-sor2$ ./rbexe
Partiion type: 1
Encontrado FAT12 0
  Jump code: EB:3C:90
  OEM code: [mkfs.fat]
  sector_size: 512
  Sectors per cluster: 4
  Reserved area size, in sectors: 1
  Number of FATs: 2
  Max number of files in the root directory: 512
  Number of sectors in the file system: 2048
  Media type: F8
  Size of each FAT, in sectors: 2
  Number of sectors per track in storage device: 32
  Number of heads in storage device: 64
  Number of sectors before the start partition: 0
  Number of sectors in the file system: 0
  BIOS INT 13h drive number: 80
  Byte not used: 01
  Extended boot signature: 29
  volume_id: 0x06C8055F
  Volume label: [NO NAME    ]
  Filesystem type: [FAT12   ]
  Boot sector signature: 0xAA55
alumno@alumno-virtualbox:~/tp1/repo/tp-fat-sor2$
```

A continuación, para poder verificar que nuestra primera partición es booteable, se debe tener en cuenta el primer byte de la partición debido a que este es el flag que indica si es “activa” o no. El valor hexadecimal ‘0x00’ significa que la partición no es activa mientras que ‘0x80’ indica que si es booteable.

Considerando esta información, y utilizando el editor Ghex, inspeccionamos el primer byte de la primera partición para constatar su valor. Se observa que el primer byte tiene el valor hexadecimal ‘0x80’, lo que indica que la partición es booteable.



Para visualizar todas las particiones de la tabla de particiones, se desarrolló el archivo *read_mbr* en código C. Para lograrlo, posicionamos el puntero de archivo a la posición en la que comienza la tabla de particiones. La misma está almacenada en la variable *partition_table_start*, y su valor es 446 (que es el byte en el que, gracias a lo explicado anteriormente, sabemos que comienza la tabla de particiones). Luego, mediante un ciclo se recorren los bytes de cada partición y se imprime por pantalla lo que indica cada uno de ellos.

```
    unsigned int partition_table_start = 446;
    fseek(in, partition_table_start, SEEK_SET); // nos dirigmos al byte
donde comienzan las particiones
 for(i=0; i<4; i++) {
        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);
    }
```

El resultado de ejecutar el código es el siguiente:

```
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
```

Como mencionamos previamente, sólo la primera partición tiene información, las demás no contienen datos. Nos concentramos, entonces, en la primera partición, la cual contiene la siguiente información relevante:

- **Flag booteable:** conformado por el primer byte, en este caso es el 80, lo cual indica que es booteable.
- **Comienzo de partición en CHS:** la dirección observada es 00:02:00. El primer byte (00) indica al *start head*, el valor 02 indica el *start sector* y el byte 00 indica el *start cylinder.*
- **Partition type**: sirve para indicar el tipo de partición, en este caso, 0x01 indica que es un tipo de partición *12-bit FAT.*
- **Fin de partición en CHS:** La dirección de fin es 00:20:20. Corresponde al *end head, end sector* y *end cylinder.*
- **Dirección relativa y tamaño de sectores:** El valor observado es 0x00000001, el cual identifica el primer sector de la partición. Además, al examinar los últimos 4 bytes de nuestra partición, que son FF 07 00 00, transformados a decimal dan como resultado 2047, lo cual indica que la partición consta de 2047 sectores en total.

Cargando la tabla de archivos

Para visualizar los archivos en Ghex, comenzamos ubicándonos en la entrada del directorio raíz. En nuestro caso, tenemos un MBR de 512 bytes. Posteriormente, encontramos 2 tablas FAT, cada una de ellas con un tamaño de **2 sectores**. Dado que cada sector tiene un tamaño de 512 bytes, nuestro directorio root comienza luego del byte 2560 (calculado como 512 + ((2 * 2) * 512) = 2560), lo que corresponde a la posición 0xA00 en hexadecimal.

Antes de visualizarlos, es necesario entender cómo se almacena la información de los archivos. Esto se logra a través del formato de entradas de directorio de FAT (FAT Directory Entry), el cual consta de 32 bytes y está constituido por los siguientes campos:

Rango de bytes	Descripción
0	Primer carácter del nombre del archivo. Existen algunos caracteres especiales que denotan información adicional acerca del archivo
1-7	Resto de caracteres para el nombre del archivo. Junto con el primer carácter denotan el nombre del archivo
8-10	Extensión del archivo
11	Atributos
12	Reservado
13	Creación del archivo (en décimas de segundos)
14-15	Tiempo de creación (horas, minutos, segundos)
16-17	Fecha de creación

18-19	Fecha de último acceso
20-21	Número del primer cluster MSB (most significant byte, 0 for FAT12/16)
22-23	Tiempo de última modificación (horas, minutos, segundos)
24-25	Fecha de última modificación
26-27	Número del primer cluster LSB (least significant byte)
28-31	Tamaño del archivo (0 para directorios)

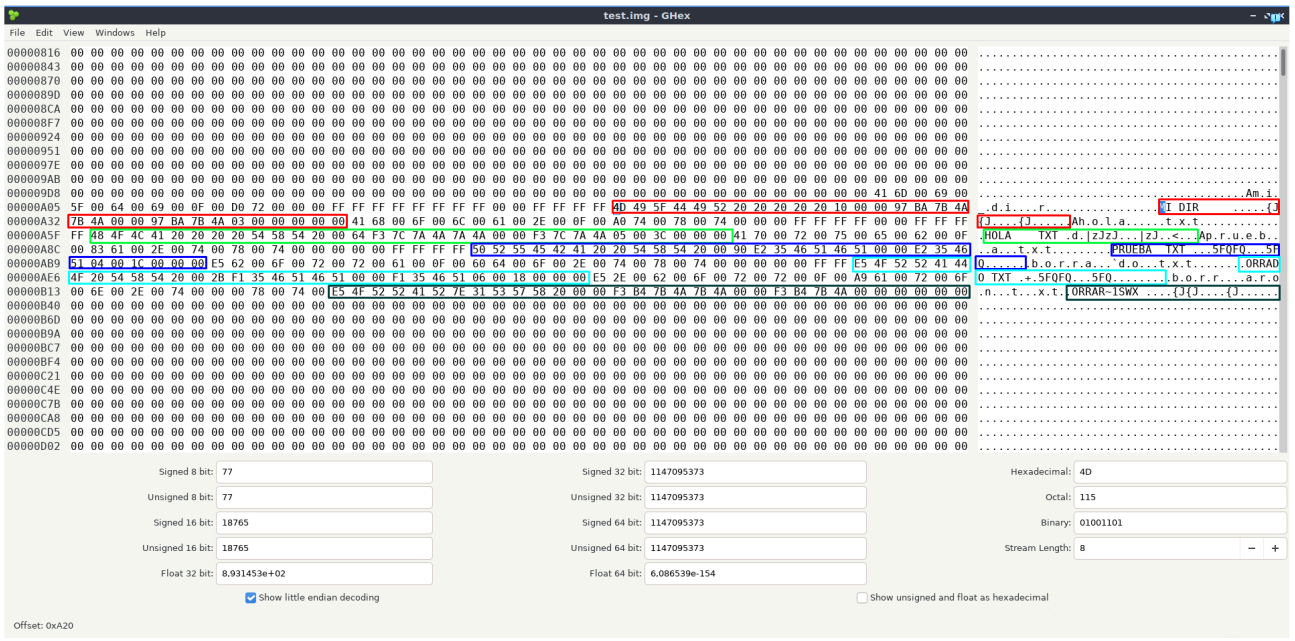
Con el primer byte del nombre del archivo, podemos obtener cierta información. La siguiente tabla muestra el significado asociado a diferentes valores de este byte. Cualquier otro valor que no esté incluido en esta tabla corresponde a un carácter real que se utiliza como primer byte en el nombre del archivo.

Valor	Descripción
0x00	Entrada sin usar
0xE5	Entrada borrada (cuando se borra algún archivo o directorio)
0x05	Indica que el primer carácter debe ser reemplazado por 0xE5 (debido a que este se esta usando para la flag anterior)
0x2E	Indica que la entrada es de un directorio

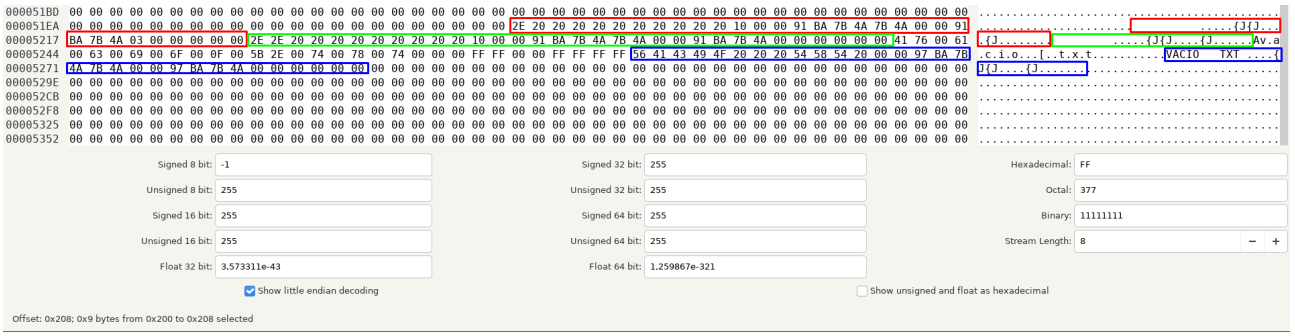
También podemos obtener información observando el byte número 11. Este byte nos indica:

Valor	Descripción
0x01	El archivo es de sólo lectura.
0x02	El archivo está oculto.
0x04	Se trata de un archivo del sistema. También están ocultos.
0x08	Entrada especial, contiene la etiqueta de volumen del disco en lugar de describir un archivo. Sólo aparece en root.
0x10	La entrada es un subdirectorio.
0x20	Bandera del archivo.
0x40	No utilizado, debe establecerse en 0.
0x80	No utilizado, debe establecerse en 0.

Teniendo en cuenta esto, podemos identificar en el editor hexadecimal un directorio (señalado con un cuadro rojo) y cuatro archivos (señalados con un cuadro verde, azul, celeste y negro). En particular, podemos ver que los marcados en celeste y negro son archivos que fueron borrados, ya que comienzan con el valor 0xE5. Podemos diferenciar archivos de directorios por el byte número 11. Como vimos en la tabla anterior, si se trata de un archivo, dicho byte tendrá un 10. En cambio, si se trata de un directorio, tendrá un 20.



También tenemos los siguientes dos directorios más otro archivo. Podemos destacar respecto al segundo directorio que se muestra (señalado con un cuadro verde) que el segundo carácter es también 0x2E. Esto quiere decir que el cluster al que apunta es el cluster del directorio padre de este, que en nuestro caso está marcando 0x00, esto se debe a que su directorio padre es el directorio raíz.



Todos estos archivos pueden ser visualizados mediante código C, utilizando el archivo *read_root*, poniendo restricción a lo que leemos basado en cómo identificamos a los archivos (como mencionamos previamente) y recorriendo el directorio raíz seguido de las entradas faltantes. Obtenemos las entradas faltantes con el siguiente cálculo:

entradas faltantes = ((sectores en el filesystem * tamaño sectores) - bytes leídos) / tamaño de directory entry

Por los datos que recopilamos anteriormente, los sectores en el filesystem son 2048, y el tamaño es de 512 bytes. Luego, los bytes leídos son 0x4A00 (18944 bytes) y el tamaño de las directory entries es de 32 bytes. De esta forma, tenemos que nos faltan 32176 entradas por leer.

De esta forma, obtenemos el siguiente resultado.

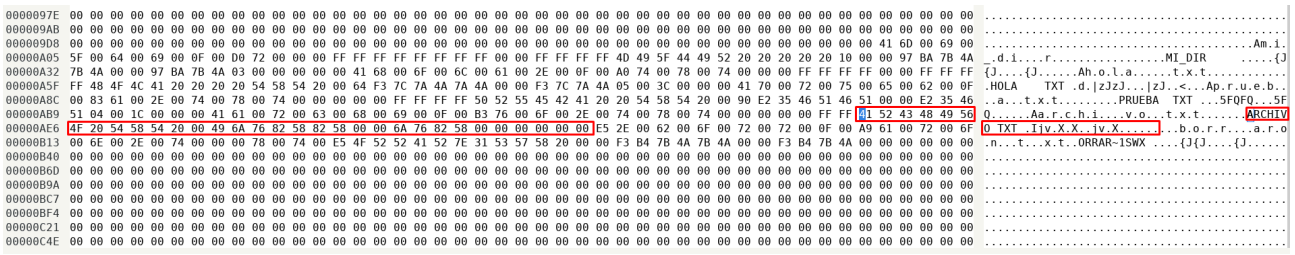
```
alumno@alumno-virtualbox:~/develop/tp-fat-sor2$ ./rrexe
Encontrada particion FAT12 0
  Partition entry
  First byte 80
  Comienzo de partición en CHS: 00:02:00
  Partition type 0x01
  Fin de partición en CHS: 20:20:00
  Dirección LBA relativa (starting cluster) 0x00000001
  Tamaño en sectores 2047


En  0x200, sector size 512, FAT size 2 sectors, 2 FATs, position to start reading: 2048

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

Leido Root directory, ahora en 0x4A00
Directorio: [.       .   ]
Directorio: [..      .   ]
Archivo: [VACIO   .TXT]
```

Creando y eliminando archivo

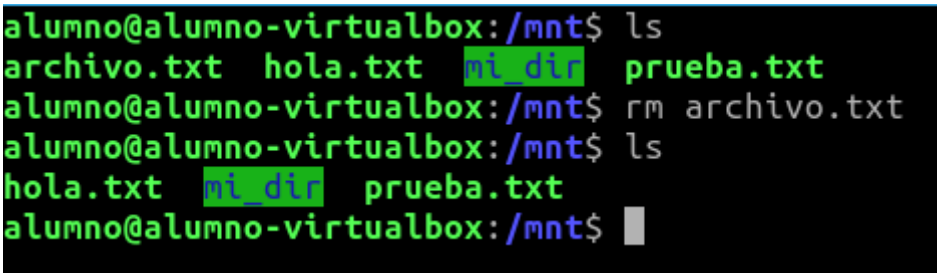
Ahora vamos a ver que sucede en nuestro archivo de test.img cuando creamos y borramos un archivo. Para ello, montamos la imágen y creamos un archivo llamado *archivo.txt* en el directorio /mnt..

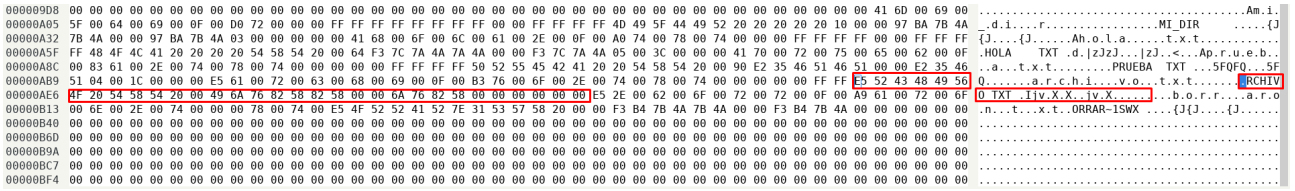
```
alumno@alumno-virtualbox:~/tp1/repo/tp-fat-sor2$ sudo mount test.img /mnt -o loop,umask=000
[sudo] contraseña para alumno:
alumno@alumno-virtualbox:~/tp1/repo/tp-fat-sor2$ cd /
bin/        dev/        lib/        libx32/     mnt/        root/       snap/       tmp/
boot/       etc/        lib32/      lost+found/ opt/        run/        srv/        usr/
.cache/     home/       lib64/      media/      proc/       sbin/       sys/        var/
alumno@alumno-virtualbox:~/tp1/repo/tp-fat-sor2$ cd /
bin/        dev/        lib/        libx32/     mnt/        root/       snap/       tmp/
boot/       etc/        lib32/      lost+found/ opt/        run/        srv/        usr/
.cache/     home/       lib64/      media/      proc/       sbin/       sys/        var/
alumno@alumno-virtualbox:~/tp1/repo/tp-fat-sor2$ cd /mnt/
alumno@alumno-virtualbox:/mnt$ ls
hola.txt  mi_dir  prueba.txt
alumno@alumno-virtualbox:/mnt$ touch archivo.txt
alumno@alumno-virtualbox:/mnt$ ls
archivo.txt  hola.txt  mi_dir  prueba.txt
alumno@alumno-virtualbox:/mnt$
```

Nos dirigimos al editor Ghex para ver si la creación del archivo provocó que se modificara test.img, y, efectivamente, podemos ver el archivo recién creado en la sección de root.

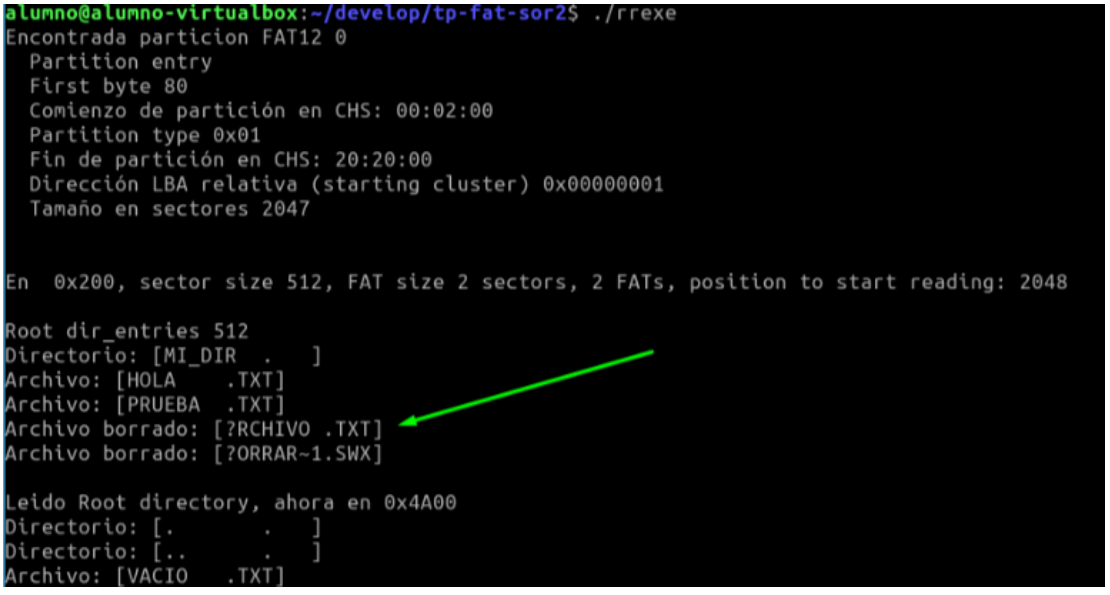


Al eliminar el archivo, lo que debería pasar es que la entrada relacionada ahora debería comenzar con 0xE5, como habíamos visto previamente. Entonces eliminamos el archivo y verificamos nuevamente el editor.





Efectivamente, podemos observar que el archivo comienza con 0xE5, indicando que fue borrado. Luego podemos ver este resultado ejecutando el código C, *read_root,* para el ejemplo anterior.



Podemos notar que tanto en el editor hexadecimal como por el código ya no podemos visualizar el archivo borrado que estaba anteriormente, el cual estaba denotado como **[?ORRADO .TXT]**. Esto quiere decir que cuando creamos nuestro archivo, se usó la porción de memoria que estaba ocupada por el archivo borrado. Esto es interesante ya que nos indica que no cualquier archivo que eliminemos podría ser recuperable, debido a que si bien queda grabada la información, esta puede ser sobreescrita por nuevas entradas.

Leyendo archivos

Para empezar a trabajar en este punto, luego de montar la imagen, procedemos a crear el archivo *lapapa.txt* en el root directory. Posteriormente, mediante el comando *echo* escribimos el texto “hola papa”.

```
alumno@alumno-virtualbox:~/tp-fat-sor2$ sudo mount test.img /mnt -o loop,umask=000
alumno@alumno-virtualbox:~/tp-fat-sor2$ cd /mnt
alumno@alumno-virtualbox:/mnt$ touch lapapa.txt
alumno@alumno-virtualbox:/mnt$ echo "hola papa" > lapapa.txt
```

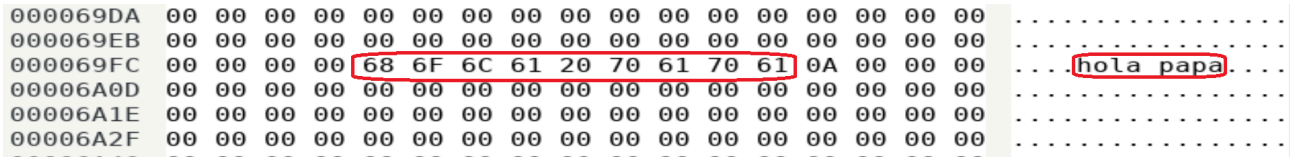
Veamos este archivo primero en el editor hexadecimal, revisando las direcciones a partir del root directory. Como ya hemos mencionado, este comienza luego del espacio reservado para las tablas FAT, es decir, a partir del byte 2560 (calculado como 512 + ((2 * 2) * 512) = 2560), lo que corresponde a la posición 0xA00 en hexadecimal. Finalmente, encontramos nuestro archivo recientemente creado.

```
00000A7E  00 00 41 70 00 72 00 75 00 65 00 62 00 0F 00 83 61  ..Ap.r.u.e.b....a
00000A8F  00 2E 00 74 00 78 00 74 00 00 00 00 00 FF FF FF FF  ...t.x.t.........
00000AA0  50 52 55 45 42 41 20 20 54 58 54 20 00 90 E2 35 46  PRUEBA  TXT ...5F
00000AB1  51 46 51 00 00 E2 35 46 51 04 00 1C 00 00 00 41 6C  QFQ...5FQ......Al
00000AC2  00 61 00 70 00 61 00 70 00 0F 00 E3 61 00 2E 00 74  .a.p.a.p....a...t
00000AD3  00 78 00 74 00 00 00 00 00 FF FF FF FF 4C 41 50 41  .x.t.........LAPA
00000AE4  50 41 20 20 54 58 54 20 00 BD 6B 03 85 58 85 58 00  PA  TXT ..k..X.X.
00000AF5  00 6B 03 85 58 06 00 0A 00 00 00 E5 2E 00 62 00 6F  .k..X.........b.o
00000B06  00 72 00 72 00 0F 00 A9 61 00 72 00 6F 00 6E 00 2E  .r.r....a.r.o.n..
```

A continuación, veamos el mismo archivo utilizando el programa del punto anterior, es decir, mediante *read_root.c*

```
alumno@alumno-virtualbox:~/tp-fat-sor2$ ls
Makefile  read_boot.c  read_file.c  read_mbr.c  README.md  read_root.c  test.img
alumno@alumno-virtualbox:~/tp-fat-sor2$ make read_root
cc     read_root.c   -o read_root
alumno@alumno-virtualbox:~/tp-fat-sor2$ ./read_root
Encontrada particion FAT12 0
  Partition entry
  First byte 80
  Comienzo de partición en CHS: 00:02:00
  Partition type 0x01
  Fin de partición en CHS: 20:20:00
  Dirección LBA relativa (starting cluster) 0x00000001
  Tamaño en sectores 2047

En  0x200, sector size 512, FAT size 2 sectors, 2 FATs, position to start reading: 2048

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

Leido Root directory, ahora en 0x4A00
Directorio: [.       .   ]
Directorio: [..      .   ]
Archivo: [VACIO   .TXT]
Se encontraron 4 archivos.
```

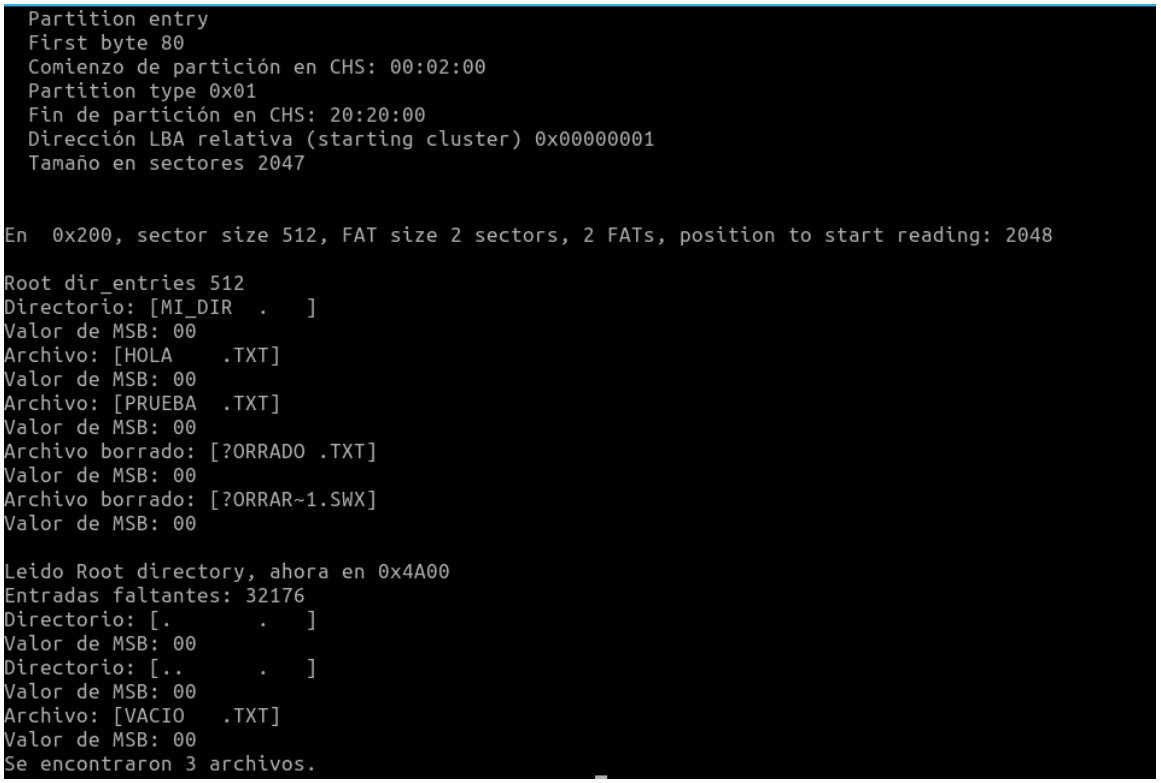
Veamos el contenido del archivo en el editor hexadecimal del archivo no borrado.



Antes de explorar el resultado de la segunda parte de este punto, es importante explicar la lógica utilizada para su realización mediante código.

Esta lógica se fundamenta en conceptos clave del filesystem FAT, como el tamaño de los clústers, la comprensión de la estructura del root directory y la utilización de sus atributos, como el campo *first_cluster_number_LSB*, para determinar la ubicación del primer clúster del archivo. El *first_cluster_number_LSB* es el byte menos significativo (*least significant byte*), mientras que el *first_cluster_number_MSB* es el byte más significativo (*most significant byte*) de los números de cluster. En nuestro caso, tenemos 502 clusters en el data sector (esto porque teníamos para leer 32176 entradas de directorio en el data sector, luego 32176 * 32 bytes = 1029632 bytes, que posteriormente dividimos por el tamaño de clusters y obtenemos 1029632 / 2048 = 502,75). Por lo tanto, como los *first_cluster_numbers* son de dos bytes y como el LSB nos indicaría hasta los primeros 0xFFFF = 65535 clusters del disco, se decidió utilizar únicamente el *first_cluster_number_LSB*, ya que es suficiente para nuestro caso.

A continuación, la captura de los valores de MSB de los archivos. Como podemos observar todos dan cero, por lo cual no nos aporta información relacionada al número de cluster.



Luego, para determinar la ubicación del primer clúster en nuestro filesystem, comenzamos desde la posición del *root directory* calculada anteriormente y añadimos el tamaño en bytes que ocupa este directorio. Esto implica multiplicar el tamaño de cada entrada en el *root directory* por la cantidad total de entradas en dicho directorio y sumarlo a la posición inicial.

A continuación, para calcular el tamaño de cada clúster, multiplicamos la cantidad de sectores asignados a cada clúster por el tamaño del sector en bytes. Estos cálculos se escriben en el código de la siguiente forma:

```
root_directory_position = ftell(in);

first_cluster = root_directory_position + (bs.root_dir_entries * sizeof(entry));

size_cluster = bs.sectors_per_cluster * bs.sector_size;
```

Para encontrar la posición de inicio de un archivo específico, utilizamos el número del primer clúster (*first_cluster_number_LSB*) obtenido de cada entrada del root directory, el cual nos indica dónde comienza el primer clúster del archivo en cuestión. A partir de la posición del primer clúster calculada previamente, nos desplazamos a la posición relativa del archivo sumando la diferencia entre el número del primer clúster (*first_cluster_number_LSB*) y 2 (ya que las dos primeras posiciones de clústeres están reservadas). Multiplicamos esta diferencia por el tamaño del clúster y lo sumamos a la posición de inicio del clúster calculada anteriormente. De esta manera, obtenemos la posición de inicio del archivo. A continuación. adjuntamos el código con la implementación de esta sección.

```
unsigned int get_file_position(Fat12Entry *entry, unsigned short first_cluster, unsigned short
cluster_size) {
           unsigned short first_cluster_number_lsb = entry->first_cluster_number_LSB;
           unsigned int file_position =
                 first_cluster + ((first_cluster_number_lsb - 2) * cluster_size);
           return file_position;
}
```

Finalmente, utilizando esta posición inicial del archivo y el tamaño del archivo obtenido de la entrada del *root directory*, nos desplazamos hasta la posición correspondiente y leemos el archivo, carácter por carácter.

```
void read_and_print_file(unsigned short file_position, unsigned int file_size)
{
    FILE *fs = fopen("test.img", "rb");
    char file_content[file_size];
    fseek(fs, file_position, SEEK_SET);    // nos movemos a la posición del archivo
    fread(file_content, file_size, 1, fs); // leemos bytes de acuerdo al tamaño del archivo desde donde habíamos
posicionado el puntero, y lo guardamos en file_content
    printf("\n");
    for (int i = 0; i < file_size; i++)
    {
        printf("%c", file_content[i]); // imprimimos el contenido
    }
    printf("\n");
    fclose(fs);
}
```

El archivo *read_file.c* contiene la implementación de esta lógica. A continuación, compilamos el archivo y lo ejecutamos para ver el resultado.

Recuperando archivos

Hemos creado el programa *file_recovery* para recuperar archivos eliminados de un sistema de archivos FAT12 en una imagen de disco llamada "test.img". A continuación, se explicará cómo funciona cada parte del código:

Definición de estructuras

Se han definido tres estructuras: *PartitionTable*, *Fat12BootSector*, y *Fat12Entry*. Cada una de estas representa un componente del sistema: la tabla de particiones, el sector de arranque (Boot Sector) y una entrada de directorio en el sistema de archivos FAT12, respectivamente. Cada estructura está definida con los campos necesarios para representar la información específica de ese componente.

Función *recover_file*

Esta función recibe un pointer como argumento, el cual indica la posición en la que se encuentra el archivo que se desea recuperar. Abre test.img en modo lectura y escritura ("r+"). Luego, inserta una letra "L" en la posición especificada por el puntero, lo que efectivamente "restaura" el archivo. Finalmente, lo cierra. Esta función está representada de la siguiente forma:

```c
void recover_file(unsigned short pointer)
{
    FILE * in = fopen("test.img", "r+");

    char charAux[1] ="L";
    fseek(in, pointer , SEEK_SET);
    fwrite(charAux, sizeof(char), sizeof(charAux), in);
    printf("Archivo se ha restaurado correctamente!\n");

    fclose(in);
}
```

Función *print_file_info*

Esta función recibe como argumentos una estructura *Fat12Entry* y un pointer. En primer lugar, verifica el primer byte del nombre del archivo en la entrada de directorio. Si el byte es 0x00, significa que el archivo no está asignado y la función no realiza ninguna acción. Si el byte es 0xE5, indica que el archivo ha sido eliminado; en este caso, la función imprime un mensaje indicando el nombre del archivo eliminado y llama a la función ***recover_file*** para intentar recuperarlo. En cualquier otro caso, la función no realiza ninguna acción. El código de esta función es:

```
void print_file_info(Fat12Entry *entry, unsigned short pointer_recover_file)
{
    switch(entry->filename[0])
    {
        case 0x00:
            return;
        case 0xE5:
            printf("Archivo borrado: [%c%.7s.%.3s]\n", 0xE5, entry->name, entry->extension);
            recover_file(pointer_recover_file);
            return;
        default:
            return;
    }
}
```

Función *main*

Primero, abre **test.img** en modo lectura binaria ("rb"). Luego, se declaran variables para almacenar la información de las particiones, el sector de arranque y una entrada de directorio, así como una variable para guardar el puntero del archivo que será utilizado para recuperar archivos eliminados.

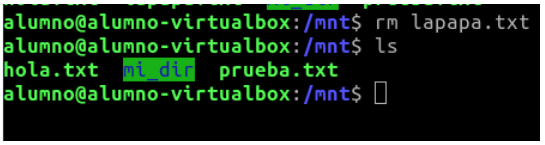
A continuación, se lee la tabla de particiones del archivo de imagen de disco y busca una partición FAT12. Si se encuentra, se guarda la información del boot sector en una variable ***'bs'***. Luego se posiciona el puntero del archivo en el sector root del sistema de archivos FAT12. Se procede a leer cada entrada de directorio en el sector root, imprimiendo información sobre los archivos y llamando a la función *print_file_info* para determinar si se debe intentar recuperar algún archivo eliminado. Finalmente, se cierra el archivo ***test.img*** y el programa termina su ejecución.

Seguidamente, se mostrará un conjunto de capturas del código implementado.
Creamos el archivo *lapapa.txt*.

```
alumno@alumno-virtualbox:/mnt$ echo "Hola Diego, Javi, Cami, Kevo y Lu" > lapapa.txt
alumno@alumno-virtualbox:/mnt$ ls
hola.txt  lapapa.txt  mi_dir  prueba.txt
alumno@alumno-virtualbox:/mnt$
```

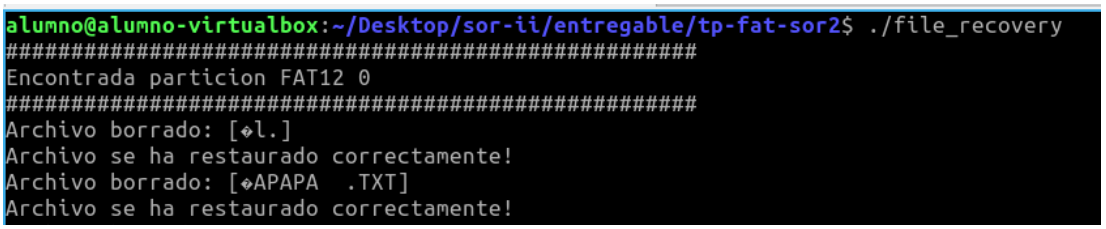
Podemos observar nuestro archivo y su contenido con nuestro *read_file*.

Se elimina el archivo *lapapa.txt*.

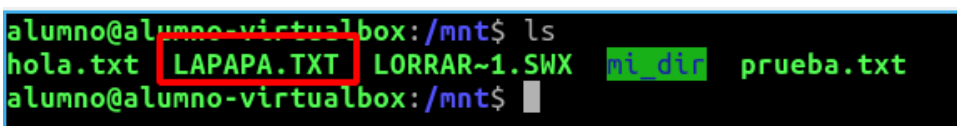


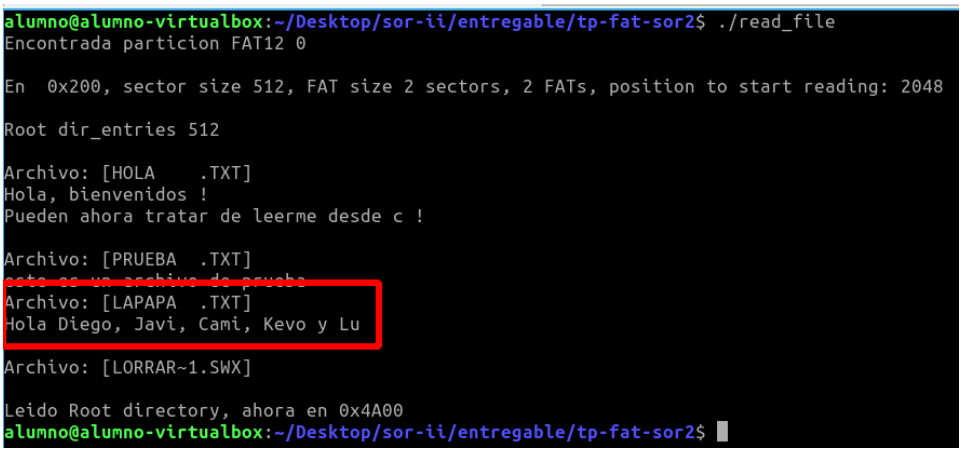
Podemos observar que el archivo *read_file* no lo encuentra.

Se ejecuta el archivo *file_recovery.*



Desmontamos y volvemos a montar.





Finalmente se vuelve a tener el archivo con su contenido.

Repositorio en Github

En el siguiente link se encuentra el repositorio con el código de los puntos del informe:
https://github.com/jisantillan/tp-fat-sor2

Bibliografía

(*A Tutorial on the FAT File System (Tavi.co.uk)*, n.d.)
A tutorial on the FAT file system (tavi.co.uk)

(*An Overview of FAT12 The File Allocation Table (FAT) Is a Table Stored on a Hard Disk or Floppy Disk That Indicates the Status*, n.d.)
An overview of FAT12 The File Allocation Table (FAT) is a table stored on a hard disk or floppy disk that indicates the status

(*Bloque De Arranque*, n.d.)
Bloque de arranque - Wikipedia, la enciclopedia libre

(*Comando De Montaje En Linux*, n.d.)
comando de montaje en Linux con ejemplos – Barcelona Geeks

(*Código De Tipo De Partición*, n.d.)
Código de tipo de partición - Wikipedia, la enciclopedia libre

(*C Programming*, n.d.)
C Programming - File Input/Output

(*Design of the FAT File System*, n.d.)
Design of the FAT file system - Wikipedia

(*Microsoft FAT Specification - Microsoft Corporation August 30 2005*, 2005)
FAT.pdf (mit.edu)

(*An Overview of FAT12 The File Allocation Table (FAT) Is a Table Stored on a Hard Disk or Floppy Disk That Indicates the Status*, n.d.)
fat12_description.pdf (lth.se)

(*File System Data Structures*, n.d.)
File System Data Structures (c-jump.com)

(Sedory, n.d.)
Introduction to Partition Tables

(*Loop Device*, n.d.)
Loop device - Wikipedia

(*Montaje*, n.d.)
Montaje

(*Mandato Mount*, n.d.)
Mandato mount

(*MANEJO DE ARCHIVOS n.d.*)
MANEJO DE ARCHIVOS

(*MBR Vs GPT: Diferencias Entre Los Estándares Más Usados*, n.d.)
MBR vs GPT: Diferencias entre los estándares más usados

(*Partition Table*, 2024)
Partition Table - OSDev Wik

(*Paul's 8051 Code Library: Understanding the FAT32 Filesystem*, 2005)
Paul's 8051 Code Library: Understanding the FAT32 Filesystemi

(Aschenbrenner, 2022)
Reading Files from a FAT12 Partition – SQLpassion

(Registro de arranque principal, n.d.)
Registro de arranque principal - Wikipedia, la enciclopedia libre

(The FAT filesystem: FAT, n.d.)
The FAT filesystem: FAT (tue.nl)

(Tabla de particiones, n.d.)
Tabla de particiones - Wikipedia, la enciclopedia libre

(Tabla de asignación de archivos, n.d.)
Tabla de asignación de archivos - Wikipedia, la enciclopedia libre

(*Understanding FAT 12*, n.d.)
UnderstandingFAT12.pdf (drexel. (*Understanding FAT 12*, n.d.)edu)

(*Umask (Español) - ArchWiki*, 2022)
umask (Español) - ArchWiki

(Gite, 2024)
What is Umask and How To Setup Default umask Under Linux? - nixCraft

(*FAT File Name and Extension Representation*, n.d.)
Zkouska_Principy_pocitacu_2019-20_-_varianta_01_-_priloha1_-_format_root_directory_na_FAT_FS.pdf (cuni.cz)

(*Preparación De Equipos En Centros Docentes Para El Uso De Las TIC"*, n.d.)
2.5.1 Esquemas o estilos de partición

(Operating System Concepts)
Abraham-Silberschatz-Operating-System-Concepts-10th-2018.pdf(CAP 13 - 15)

"File System Forensic Analysis" de Brian Carrier (Descarga) (CAP 9-10)