Documentación

Inspector de sistemas de archivos

Advanced Operating Systems

Alvaro Bello Garrido

27/06/2025

Índice

Resumen proyecto

El proyecto consiste en un programa inspector de sistemas de archivos, el cual es capaz de reconocer FAT16 y EXT2 (en caso de inspeccionar un sistema de archivos diferente este no será reconocido y se mostrará el debido mensaje de error), y realizar las siguientes operaciones sobre este: mostrar información del sistema de archivos, mostrar un esquema en árbol del sistema de archivos, y mostrar el contenido de un archivo dentro del sistema de archivos.

1. Diseño y detalles de implementación

1.1 Identificación y recopilación de información del sistema de archivos

En primer lugar, resulta fundamental determinar si el sistema de archivos presente es de tipo EXT2 o FAT16. Para ello, es preciso localizar un indicador característico en cada uno de estos sistemas cuya constancia permita su identificación precisa.

Identificación sistemas EXT:

Según la documentación técnica de los sistemas EXT, el superbloque se encuentra ubicado en el offset 1024. Dicho superbloque constituye una estructura de 1024 bytes que almacena información relevante acerca del sistema de archivos.

Table 3.1. Sample Floppy Disk Layout, 1KiB blocks

Block Offset	Length	Description
byte 0	512 bytes	boot record (if present)
byte 512	512 bytes	additional boot record data (if present)
-- block group 0, blocks 1 to 1439 --		
byte 1024	1024 bytes	superblock
block 2	1 block	block group descriptor table
block 3	1 block	block bitmap
block 4	1 block	inode bitmap
block 5	23 blocks	inode table
block 28	1412 blocks	data blocks

Tras localizar el superbloque, la documentación detalla tanto la estructura como el contenido de este:

Table 3.3. Superblock Structure

Offset (bytes)	Size (bytes)	Description
0	4	s_inodes_count
4	4	s_blocks_count
8	4	s_r_blocks_count
12	4	s_free_blocks_count
16	4	s_free_inodes_count
20	4	s_first_data_block
24	4	s_log_block_size
28	4	s_log_frag_size
32	4	s_blocks_per_group
36	4	s_frags_per_group
40	4	s_inodes_per_group
44	4	s_mtime
48	4	s_wtime
52	2	s_mnt_count
54	2	s_max_mnt_count
56	2	s_magic
58	2	s_state
60	2	s_errors
62	2	s_minor_rev_level
64	4	s_lastcheck
68	4	s_checkinterval
72	4	s_creator_os
76	4	s_rev_level
80	2	s_def_resuid
82	2	s_def_resgid
-- EXT2_DYNAMIC_REV Specific --		
84	4	s_first_ino
88	2	s_inode_size

El análisis de cada parámetro del superbloque revela que, en los sistemas de archivos Ext2, el campo s_magic presenta siempre el valor 0xEF53. Así, la detección de sistemas EXT2 puede llevarse a cabo verificando la posición de este parámetro y su correspondencia con el valor especificado.

s_magic

16bit value identifying the file system as Ext2. The value is currently fixed to `EXT2_SUPER_MAGIC` of value 0xEF53.

Una vez obtenida esta información, la identificación de sistemas EXT2 en la implementación propuesta se realiza siguiendo estos pasos:

1. Se lee el superbloque desplazándose hasta su posición inicial mediante la función 'lseek'.
2. Se almacena la información relevante del superbloque en una estructura personalizada (EXT2_Superblock) que contenga todos los parámetros del superbloque.
3. Se comprueba el valor del campo 's_magic' para determinar si se trata de un sistema EXT2 y, en caso afirmativo, se muestra la información correspondiente.

```
if (lseek(fd, EXT2_SUPERBLOCK_OFFSET, SEEK_SET) == -1) {
    perror("Error al buscar el superbloque");
    close(fd);
    return -1;
}

EXT2_Superblock sb;
if (read(fd, &sb, sizeof(EXT2_Superblock)) != sizeof(EXT2_Superblock)) {
    perror("Error al leer el superbloque");
    close(fd);
    return -1;
}

if (sb.s_magic != EXT2_SUPER_MAGIC) {
    //printf("No es un sistema de archivos EXT2/3/4 (magic: 0x%X)\n", sb.s_magic);
    close(fd);
    return -1;
}

// Calcular tamaño de bloque
uint32_t block_size = 1024 << sb.s_log_block_size;

// Mostrar información
printf("--- Filesystem Information ---\n");
printf("Filesystem: EXT2\n\n");

printf("INODE INFO\n");
printf("  Size: %u\n", 256);  // Tamaño fijo típico para EXT2
printf("  Num Inodes: %u\n", sb.s_inodes_count);
```

Identificación sistemas FAT:

Para la identificación de sistemas FAT16, la documentación técnica señala en el apartado “FAT Type Determination” (página 13) el procedimiento a seguir:

Contents

Dicho apartado especifica claramente que la determinación del tipo de sistema de archivos FAT exige el cálculo del número de clusters.

FAT Type Determination

There is considerable confusion over exactly how this works, which leads to many “off by 1”, “off by 2”, “off by 10”, and “massively off” errors. It is really quite simple how this works. The FAT type—one of FAT12, FAT16, or FAT32—is determined by the count of clusters on the volume and *nothing* else.

Y seguidamente se indica como realizar este cálculo del número de clusters, y el rango de número de clusters de los sistemas de archivos FAT16:

```
If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;

If(BPB_TotSec16 != 0)
    TotSec = BPB_TotSec16;
Else
    TotSec = BPB_TotSec32;

DataSec = TotSec - (BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);
```

```
If(CountofClusters < 4085) {
/* Volume is FAT12 */
} else if(CountofClusters < 65525) {
    /* Volume is FAT16 */
} else {
    /* Volume is FAT32 */
}
```

Con esta información diseñamos una función específica para calcular el número de clusters:

```c
uint32_t calculate_cluster_count(const FAT16_BPB *bpb) {
    // Calcular sectores del directorio raíz
    uint32_t RootDirSectors = ((bpb->RootEntries * 32) + (bpb->BytesPerSector - 1)) / bpb->BytesPerSector;

    // Determinar total de sectores (usando 16-bit o 32-bit según corresponda)
    uint32_t TotalSectors = (bpb->TotalSectors16 != 0) ? bpb->TotalSectors16 : bpb->TotalSectors32;

    // Calcular sectores en la región de datos
    uint32_t DataSec = TotalSectors - (bpb->ReservedSectors + (bpb->NumFATs * bpb->FATSize16) + RootDirSectors);

    // Calcular número de clusters (en región de datos)
    return DataSec / bpb->SectorsPerCluster;
}
```

Finalmente, empleando una estructura (FAT16_BPB) en la que se guarda la información leída del BPB (Bios Parameter Block), se utiliza la función mencionada para verificar si el sistema de archivos corresponde a FAT16, en conformidad con lo indicado en la documentación técnica:

```c
// Leer 512 bytes del sector de arranque
unsigned char boot_sector[512];
if (read(fd, boot_sector, sizeof(boot_sector)) != sizeof(boot_sector)) {
    perror("Error al leer el sector de arranque");
    close(fd);
    return -1;
}

// Obtener el BPB desde el sector leído
FAT16_BPB *bpb = (FAT16_BPB *)(boot_sector);

// Calcular número de clusters
uint32_t cluster_count = calculate_cluster_count(bpb);

// Verificar si el tipo de sistema de archivos es FAT16
if ((cluster_count >= 4085 && cluster_count < 65525)) {
    // Es FAT16
    printf("\n--- Filesystem Information ---\n\n");
    printf("Filesystem: FAT16\n\n");

    printf("System name: %.8s\n", bpb->OEMName);
    printf("Sector size: %u\n", bpb->BytesPerSector);
    printf("Sectors per cluster: %u\n", bpb->SectorsPerCluster);
    printf("Reserved sectors: %u\n", bpb->ReservedSectors);
    printf("# of FATs: %u\n", bpb->NumFATs);
    printf("Max root entries: %u\n", bpb->RootEntries);
    printf("Sectors per FAT: %u\n", bpb->FATSize16);
    printf("Label: %.11s\n\n", bpb->VolumeLabel);
```

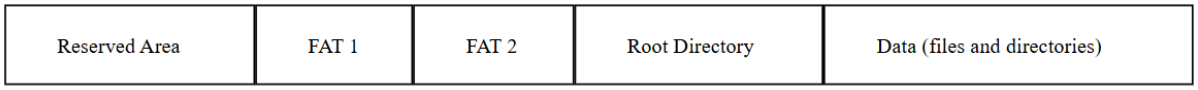
1.2 Mostrar árbol de directorios

Para la segunda fase del proyecto, se deberán mostrar los archivos y directorios del sistema. Con este objetivo, se han implementado funciones recursivas tanto para FAT16 como para EXT2, permitiendo recorrer de manera completa la estructura jerárquica de directorios desde la raíz.

FAT16:

Comenzando con FAT16, conocemos el siguiente esquema proporcionado por "https://filesystems.jiahuichen.dev/fat-16/" donde podemos ver que para leer el árbol de directorios deberemos:

1. Leer el directorio raíz con sus correspondientes entradas.
2. Procesar recursivamente estas entradas, accediendo a estas mediante sus cadenas de clusters siguiendo las FAT (File Allocation tables).



El proceso comienza identificando el sector de inicio del directorio raíz, utilizando la información detallada en el apartado "FAT Directory Structure":

FAT Directory Structure

This is the most simple explanation of FAT directory entries. This document totally ignores the Long File Name architecture and only talks about short directory entries. For a more complete description of FAT directory structure, see the document "FAT: Long Name On-Media Format Specification".

A FAT directory is nothing but a "file" composed of a linear list of 32-byte structures. The only special directory, which must always be present, is the root directory. For FAT12 and FAT16 media, the root directory is located in a fixed location on the disk immediately following the last FAT and is of a fixed size in sectors computed from the BPB_RootEntCnt value (see computations for RootDirSectors earlier in this document). For FAT12 and FAT16 media, the first sector of the root directory is sector number relative to the first sector of the FAT volume:

```
FirstRootDirSecNum = BPB_ResvdSecCnt + (BPB_NumFATs * BPB_FATSz16);
```

Una vez obtenida la dirección de este sector, es posible calcular el tamaño total del directorio raíz mediante la expresión (RootSectors * BytesPerSector), siendo "RootSectors" el número de sectores que ocupa, dato calculable conforme a la documentación técnica en el apartado el apartado "FAT Data Structure":

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec;
```

Una vez determinado el sector y tamaño del directorio raíz, se procede a su lectura, mostrando todas sus entradas. En caso de que una entrada corresponda a un subdirectorio, la exploración continúa de manera recursiva.

Para mostrar las entradas, la función print_indent() se encarga de imprimir las tabulaciones respectivas al nivel en que se encuentre el archivo, con el objetivo de mostrar el esquema de árbol de directorios de forma estructurada y jerárquica.

Mientras que la función print_entry_name() muestra el nombre y la extensión de la entrada, siguiendo el formato especificado de 8 caracteres para el nombre y 3 caracteres para la extensión (dejando espacios en blanco entre medio en caso de no ocuparse los 8 caracteres de nombre), siguiendo la información indicada en la sección “FAT 32 Byte Directory Entry Structure->DIR_Name” (a pesar de ser la sección para sistemas FAT32, FAT16 utiliza la misma estructura para sus entradas de directorios).

The DIR_Name field is actually broken into two parts+ the 8-character main part of the name, and the 3-character extension. These two parts are “trailing space padded” with bytes of 0x20.

DIR_Name[0] may not equal 0x20. There is an implied ‘.’ character between the main part of the name and the extension part of the name that is not present in DIR_Name. Lower case characters are not allowed in DIR_Name (what these characters are is country specific).

The following characters are not legal in any bytes of DIR_Name:

- Values less than 0x20 except for the special case of 0x05 in DIR_Name[0] described above.
- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C.

Here are some examples of how a user-entered name maps into DIR_Name:

```
"foo.bar"           -> "FOO     BAR"
"FOO.BAR"           -> "FOO     BAR"
"Foo.Bar"           -> "FOO     BAR"
"foo"               -> "FOO        "
"foo."              -> "FOO        "
"PICKLE.A"          -> "PICKLE  A  "
"prettybg.big"      -> "PRETTYBGBIG"
".big"              -> illegal, DIR_Name[0] cannot be 0x20
```

Además, el proceso verifica si una entrada está vacía (0xE5) o es la última del directorio (0x00), finalizando la exploración en tales casos. Las entradas correspondientes a “long filename” (valor de atributo 0x0F) se omiten, dado que la gestión de estos casos se encuentra fuera del alcance del documento técnico original.

DIR_Name[0]
Special notes about the first byte (DIR_Name[0]) of a FAT directory entry:

- If DIR_Name[0] == 0xE5, then the directory entry is free (there is no file or directory name in this entry).
- If DIR_Name[0] == 0x00, then the directory entry is free (same as for 0xE5), and there are no allocated directory entries after this one (all of the DIR_Name[0] bytes in all of the entries after this one are also set to 0).

  The special 0 value, rather than the 0xE5 value, indicates to FAT file system driver code that the rest of the entries in this directory do not need to be examined because they are all free.

Para subdirectorios distintos del directorio raíz, la localización de su primer sector se efectúa mediante la fórmula pertinente indicada en el apartado “FAT Data Structure” de la documentación (utilizando el primer cluster del directorio como ‘N’):

```
FirstDataSector = BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;

FirstSectorofCluster = ((N - 2) * BPB_SecPerClus) + FirstDataSector;
```

A continuación, se leerá el primer cluster del directorio y mostraremos sus entradas recursivamente tal y como hicimos con el directorio raíz.

Una vez recorridas estas entradas, se recorrerán los clusters desde la fat hasta alcanzar un cluster con el valor 0xFFF8, el cual indicará el fin de la cadena de clusters.
En este proceso, para obtener el siguiente cluster de la cadena, en el apartado “FAT Type Determination” se indica que deberemos dirigirnos a la posición del cluster actual dentro de la FAT multiplicada por 2 (debido a que cada entrada de la FAT ocupa 2 bytes).

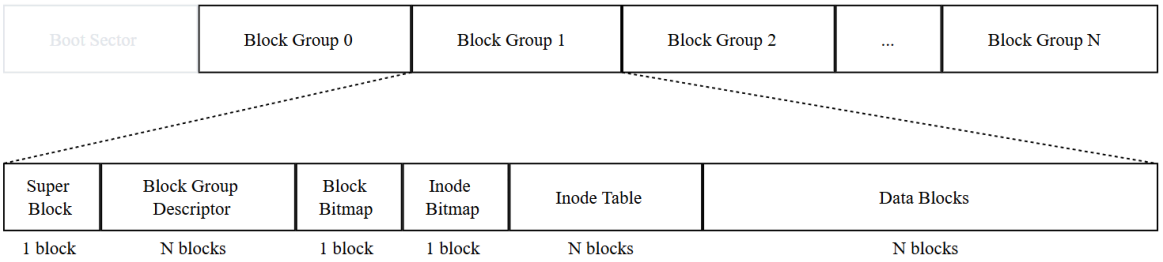
```
If(FATType == FAT16)
    FATOffset = N * 2;
```

El proceso se repetirá hasta explorar la totalidad de la estructura.

EXT2:

Siguiendo con EXT2, conocemos el siguiente esquema proporcionado por "https://filesystems.jiahuichen.dev/ext-2/" donde podemos ver que para leer el árbol de directorios deberemos:

1. Leer el inodo raíz y obtener sus bloques directos e indirectos.
2. Recorrer todos los bloques directos e indirectos del directorio.
3. Recorrer cada entrada de directorio dentro de los bloques, obteniendo el número de inodo de cada archivo o subdirectorio.
4. Para las entradas de tipo subdirectorio, aplicar recursivamente el proceso de lectura sobre el inodo asociado y sus bloques.



El proceso comienza leyendo el inodo raíz, que en ext2 siempre es el inodo número 2, como se menciona en el apartado "Defined Reserved Inodes" del documento.

Table 3.14. Defined Reserved Inodes

Constant Name	Value	Description
EXT2_BAD_INO	1	bad blocks inode
EXT2_ROOT_INO	2	root directory inode
EXT2_ACL_IDX_INO	3	ACL index inode (deprecated?)
EXT2_ACL_DATA_INO	4	ACL data inode (deprecated?)
EXT2_BOOT_LOADER_INO	5	boot loader inode
EXT2_UNDEL_DIR_INO	6	undelete directory inode

Para ello, la función read_inode() calcula la posición de este inodo dentro de la tabla de inodos mediante la fórmula "inode_offset = table_offset + (inode_num - 1) * sb->s_inode_size" donde:

- 'table_offset' es el offset de la tabla de inodos.
- Al inodo se le resta 1, ya que los inodos comienzan desde el número 1 tal y como se indica en el apartado "Locating an Inode" del documento.
- La tabla de inodos está ubicada en el bloque especificado por el campo bg_inode_table del descriptor de grupo.

Tras acceder al inodo raíz, accedemos a su campo i_block, que contiene 15 entradas (bloques directos, indirectos, doble y triple indirecto). Esta estructura está detallada en el apartado “Inode Structure” de la documentación técnica

i_blocks

32-bit value representing the total number of 512-bytes blocks reserved to contain the data of this inode, regardless if these blocks are used or not. The block numbers of these reserved blocks are contained in the i_block array.

Since this value represents 512-byte blocks and not file system blocks, this value should not be directly used as an index to the i_block array. Rather, the maximum index of the i_block array should be computed from i_blocks / ((1024<<s_log_block_size)/512), or once simplified, i_blocks/(2<<s_log_block_size).

Utilizaremos la función print_directory_recursive() para acceder de manera recursiva a estos bloques del inodo, procesando en primer lugar los bloques directos y mostrando sus entradas; en caso de encontrar subdirectorios, se realiza nuevamente el recorrido de manera recursiva.

Si el inodo tiene bloques indirectos, la lectura sigue siendo recursiva:

- El bloque indirecto de nivel 1 contiene punteros a bloques de datos.
- El de nivel 2 (doble indirecto) apunta a bloques indirectos, que a su vez apuntan a bloques de datos.
- El de nivel 3 (triple indirecto) apunta a bloques doble indirecto.

La función process_indirect_blocks() implementa este recorrido recursivo, respetando la jerarquía.

Adicionalmente, la función process_directory_block() es la encargada de procesar y mostrar las entradas incluidas en los bloques de datos directos. Para lograrlo, recorre de manera secuencial las entradas del bloque (cada una de tamaño dinámico rec_len), mostrando los nombres con la indentación adecuada según el nivel de profundidad en el árbol de directorios. Durante este proceso, se filtran las entradas especiales “.” y “..”, y, en caso de tratarse de un subdirectorio, se obtiene el inodo correspondiente y se continúa la exploración recursivamente. De este modo, la función permite recorrer de forma completa y estructurada la jerarquía de directorios, mostrando de manera visual la arquitectura en árbol del sistema de archivos.

1.3 Mostrar contenido de un archivo

La fase final del proyecto tiene como objetivo mostrar el contenido de un archivo específico almacenado en un sistema de archivos FAT16 (Para EXT2 no ha sido implementada esta funcionalidad). Esta operación implica varios pasos relacionados al encontrar el archivo y acceder a los clústeres de datos.

En primer lugar, se cambia el formato del nombre del archivo introducido por el usuario a partir de la función 'format_name' la cual divide el nombre en dos partes: una primera de 8 caracteres, con la parte del nombre principal, y una segunda de 3 caracteres con la extensión, completando ambas partes con espacios en los caracteres no utilizados, y aplicando mayúsculas, según lo especificado en las entradas de directorio de FAT16.

Posteriormente, una vez que el nombre objetivo ha sido adaptado, se procede a localizar la entrada correspondiente en el Root Directory a partir del BIOS Parameter Block (BPB):

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec;

FirstDataSector = BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;
```

La búsqueda del archivo se lleva a cabo con la función find_entry, la cual lee sucesivamente las entradas de 32 bytes presentes en el directorio raíz (cluster 0) o en los distintos clústeres de un subdirectorio. Esta exploración se realiza siguiendo la cadena FAT hasta alcanzar un valor EOC (End Of Chain) mayor o igual a 0xFFF8, comparando para cada entrada los primeros 11 bytes con el nombre ya formateado. Tras recorrer todas las entradas del Root Directory, en caso de hallar una coincidencia, se obtiene el clúster inicial a partir de los bytes localizados en los offsets 26–27, mientras que los bytes 28–31 proporcionan el tamaño del archivo en bytes.

A continuación, la función dump_file itera a través de los clústeres del archivo, continuando mientras el valor del siguiente clúster sea inferior a 0xFFF8.

Given any valid data cluster number **N**, the sector number of the first sector of that cluster (again relative to sector 0 of the FAT volume) is computed as follows:

```
FirstSectorofCluster = ((N - 2) * BPB_SecPerClus) + FirstDataSector;
```

Para cada clúster, se procede a calcular la cantidad de datos a leer, correspondiente a SectorsPerCluster * BytesPerSector, o bien al resto de bytes en caso de encontrarse en el clúster final. Los datos leídos se envían directamente a stdout para mostrarlos por pantalla, y el siguiente clúster de la cadena se determina leyendo 2 bytes en la posición calculada como:

*off_t fat_off = bpb->ReservedSectors * bpb->BytesPerSector + cur * 2;*

2. Estructuras de datos utilizadas y su justificación

2.1 Identificación y recopilación de información del sistema de archivos:

Para la primera fase utilizamos 2 estructuras de datos. Una para leer el **BPB** (Bios Parameter Block) de los sistemas de archivos FAT16, y otra para leer el **superbloque** de los sistemas de archivos EXT2. Asignamos cada uno de los valores de estas estructuras con el tamaño indicado en la documentación.

FAT16:

```c
// Estructura del BPB (BIOS Parameter Block) para FAT16
#pragma pack(push, 1)       // Guarda la configuración actual de alineación y Establece alineación a 1 byte (sin padding)
typedef struct {
    uint8_t jmpBoot[3];
    char OEMName[8];
    uint16_t BytesPerSector;
    uint8_t SectorsPerCluster;
    uint16_t ReservedSectors;
    uint8_t NumFATs;
    uint16_t RootEntries;
    uint16_t TotalSectors16;
    uint8_t Media;
    uint16_t FATSize16;
    uint16_t SectorsPerTrack;
    uint16_t NumHeads;
    uint32_t HiddenSectors;
    uint32_t TotalSectors32;
    uint8_t DriveNumber;
    uint8_t Reserved1;
    uint8_t BootSignature;
    uint32_t VolumeID;
    char VolumeLabel[11];
    char FileSystemType[8];
} FAT16_BPB;
#pragma pack(pop)       // Restaura la configuración de alineación previa (antes del push)
```

EXT2:

```c
// Estructura del superbloque EXT2
#pragma pack(push, 1)
typedef struct {
    uint32_t s_inodes_count;         // 0x00: Total inodes count
    uint32_t s_blocks_count;         // 0x04: Total blocks count
    uint32_t s_r_blocks_count;       // 0x08: Reserved blocks count
    uint32_t s_free_blocks_count;    // 0x0C: Free blocks count
    uint32_t s_free_inodes_count;    // 0x10: Free inodes count
    uint32_t s_first_data_block;     // 0x14: First data block
    uint32_t s_log_block_size;       // 0x18: Block size (log2)
    uint32_t s_log_frag_size;        // 0x1C: Fragment size (log2)
    uint32_t s_blocks_per_group;     // 0x20: Blocks per group
    uint32_t s_frags_per_group;      // 0x24: Fragments per group
    uint32_t s_inodes_per_group;     // 0x28: Inodes per group
    uint32_t s_mtime;                // 0x2C: Mount time
    uint32_t s_wtime;                // 0x30: Write time
    uint16_t s_mnt_count;            // 0x34: Mount count
    uint16_t s_max_mnt_count;        // 0x36: Max mount count
    uint16_t s_magic;                // 0x38: Magic signature (0xEF53)
    uint16_t s_state;                // 0x3A: File system state
    uint16_t s_errors;               // 0x3C: Behaviour when detecting errors
    uint16_t s_minor_rev_level;      // 0x3E: Minor revision level
    uint32_t s_lastcheck;            // 0x40: Time of last check
    uint32_t s_checkinterval;        // 0x44: Max time between checks
    uint32_t s_creator_os;           // 0x48: OS that created the filesystem
    uint32_t s_rev_level;            // 0x4C: Revision level
    uint16_t s_def_resuid;           // 0x50: Default uid for reserved blocks
    uint16_t s_def_resgid;           // 0x52: Default gid for reserved blocks
    char s_volume_name[16];          // 0x78: Volume name
} EXT2_Superblock;
#pragma pack(pop)
```

A su vez, cabe recalcar el uso de #pragma antes y después de declarar cada estructura:

- #pragma pack(push, 1): En esta línea de código establecemos la alineación a 1 byte (sin padding), y hacemos push para guardar la configuración actual.
- #pragma pack(pop): En esta línea de código utilizamos pop para obtener y restaurar la configuración de alineación anterior.

Esto se hace para desactivar el relleno automático de estructuras, porque el compilador por defecto puede insertar relleno (padding) para alinear los campos según la arquitectura y así optimizar accesos en memoria. Esa alineación introduce huecos internos y puede cambiar los desplazamientos de cada campo dentro de la estructura, lo cual generaría un problema al dejar inválidos los desplazamientos (offsets) de la documentación.

2.2 Mostrar árbol de directorios:

Para esta fase añadimos estructuras que nos permitieran almacenar de forma más sencilla y estructurada la información de los elementos involucrados en el recorrido recursivo a través del árbol de directorios. Permitiéndonos almacenar las entradas de directorios tanto en FAT16 como en EXT2, junto a la información de los inodos y los descriptores de grupo en EXT2.

FAT16:

```
typedef struct {
    char name[12];      // Nombre legible del archivo/directorio
    uint8_t attr;       // Atributos (archivo, directorio, permisos)
    uint16_t firstCluster; // Cluster inicial del archivo/directorio
} FAT16_DirEntry;
```

EXT2:

```
typedef struct {
    uint32_t inode;
    uint16_t rec_len;
    uint8_t name_len;
    uint8_t file_type;
    char name[];
} EXT2_DirEntry;
```

```c
typedef struct {
    uint16_t mode;
    uint16_t uid;
    uint32_t size;
    uint32_t atime;
    uint32_t ctime;
    uint32_t mtime;
    uint32_t dtime;
    uint16_t gid;
    uint16_t links_count;
    uint32_t blocks;
    uint32_t flags;
    uint32_t osd1;
    uint32_t block[15]; // bloques directos, indirectos, etc.
    uint32_t generation;
    uint32_t file_acl;
    uint32_t dir_acl;
    uint32_t faddr;
    uint32_t osd2[3];
} EXT2_Inode;

typedef struct {
    uint32_t bg_block_bitmap;
    uint32_t bg_inode_bitmap;
    uint32_t bg_inode_table;
    uint16_t bg_free_blocks_count;
    uint16_t bg_free_inodes_count;
    uint16_t bg_used_dirs_count;
    uint16_t bg_pad;
    uint8_t bg_reserved[12];
} EXT2_GroupDesc;
```

3. Diferentes resultados de las pruebas para ambos tipos de sistemas de archivos

A continuación se muestran los resultados de las pruebas realizadas con sistemas de archivos de varios tamaños, tanto FAT16 como EXT2, utilizando las opciones que ofrece el programa:

3.1 Identificación y recopilación de información del sistema de archivos:

FAT16

DEVINfat:

```
carla.francos@matagalls:~/soa>./program.exe --info FAT16_Doc/DEVINfat

--- Filesystem Information ---

Filesystem: FAT16

System name: mkfs.fat
Sector size: 512
Sectors per cluster: 32
Reserved sectors: 32
# of FATs: 2
Max root entries: 512
Sectors per FAT: 256
Label: NO NAME
```

libfat:

```
carla.francos@matagalls:~/soa>./program.exe --info FAT16_Doc/libfat

--- Filesystem Information ---

Filesystem: FAT16

System name: mkdosfs
Sector size: 1024
Sectors per cluster: 1
Reserved sectors: 1
# of FATs: 2
Max root entries: 512
Sectors per FAT: 16
Label: TEST2
```

studentfat100MB:

```
carla.francos@matagalls:~/soa>./program.exe --info FAT16_Doc/studentfat100MB

--- Filesystem Information ---

Filesystem: FAT16

System name: mkfs.fat
Sector size: 512
Sectors per cluster: 4
Reserved sectors: 4
# of FATs: 2
Max root entries: 512
Sectors per FAT: 200
Label: NO NAME
```

EXT2

D3VINext:

```
carla.francos@matagalls:~/soa>./program.exe --info EXT2_Doc/D3V1Next
--- Filesystem Information ---
Filesystem: EXT2

INODE INFO
  Size: 256
  Num Inodes: 2608
  First Inode: 11
  Inodes Group: 1304
  Free Inodes: 2589

INFO BLOCK
  Block size: 1024
  Reserved blocks: 521
  Free blocks: 9657
  Total blocks: 10420
  First block: 1
  Group blocks: 8192
  Group flags: 8192

INFO VOLUME
  Volume name:
  Last Checked: Sat Mar 22 21:50:25 2025
  Last Mounted: Sat Mar 22 22:01:22 2025
  Last Written: Sat Mar 22 22:01:22 2025
```

lolext:

```
carla.francos@matagalls:~/soa>./program.exe --info EXT2_Doc/lolext
--- Filesystem Information ---
Filesystem: EXT2

INODE INFO
  Size: 256
  Num Inodes: 2560
  First Inode: 11
  Inodes Group: 1280
  Free Inodes: 2539

INFO BLOCK
  Block size: 1024
  Reserved blocks: 512
  Free blocks: 9466
  Total blocks: 10240
  First block: 1
  Group blocks: 8192
  Group flags: 8192

INFO VOLUME
  Volume name:
  Last Checked: Mon Apr 10 14:32:05 2023
  Last Mounted: Mon Apr 10 14:58:42 2023
  Last Written: Mon Apr 10 14:58:42 2023
```

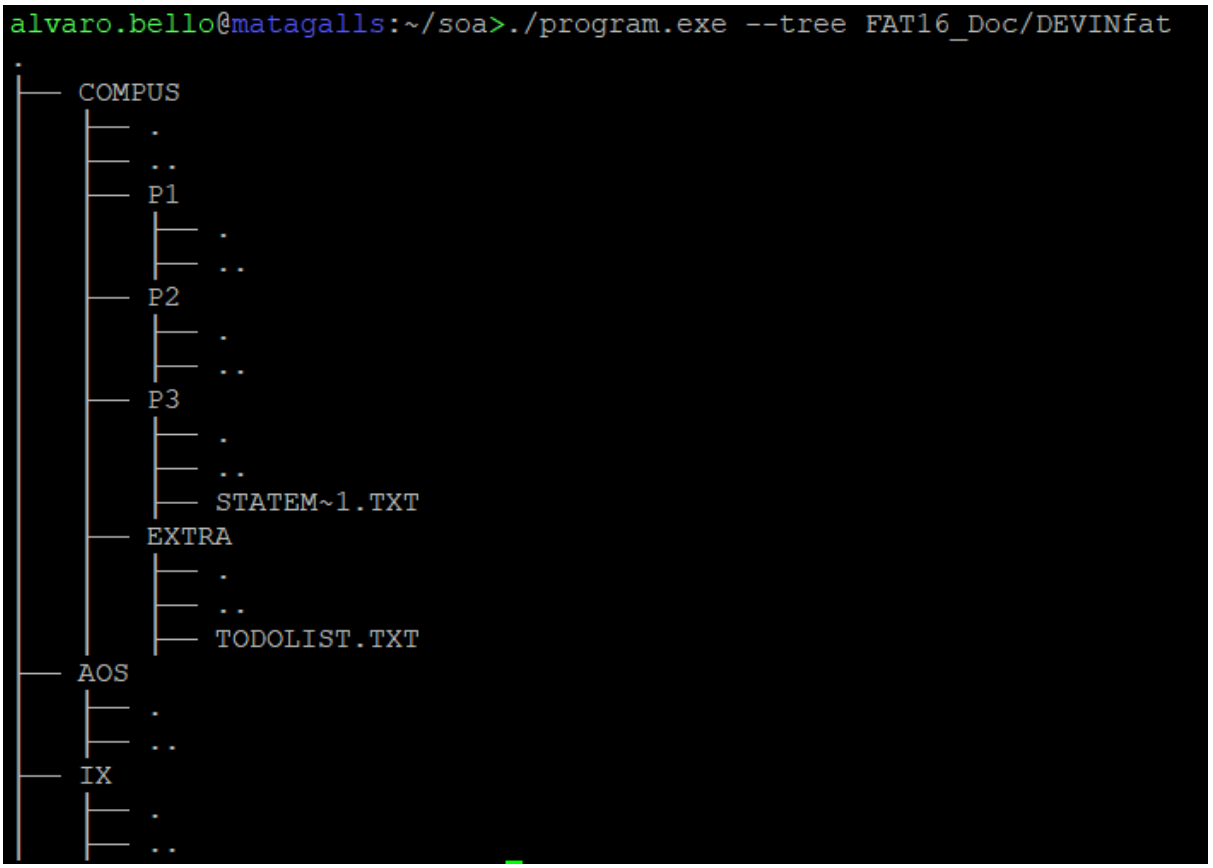
studentext100MB:

```
carla.francos@matagalls:~/soa>./program.exe --info EXT2_Doc/studentext100MB
--- Filesystem Information ---
Filesystem: EXT2

INODE INFO
  Size: 256
  Num Inodes: 25600
  First Inode: 11
  Inodes Group: 25600
  Free Inodes: 25542

INFO BLOCK
  Block size: 4096
  Reserved blocks: 1280
  Free blocks: 23933
  Total blocks: 25600
  First block: 0
  Group blocks: 32768
  Group flags: 32768

INFO VOLUME
  Volume name:
  Last Checked: Fri Mar 28 21:07:27 2025
  Last Mounted: Fri Mar 28 21:09:24 2025
  Last Written: Fri Mar 28 21:09:24 2025
```

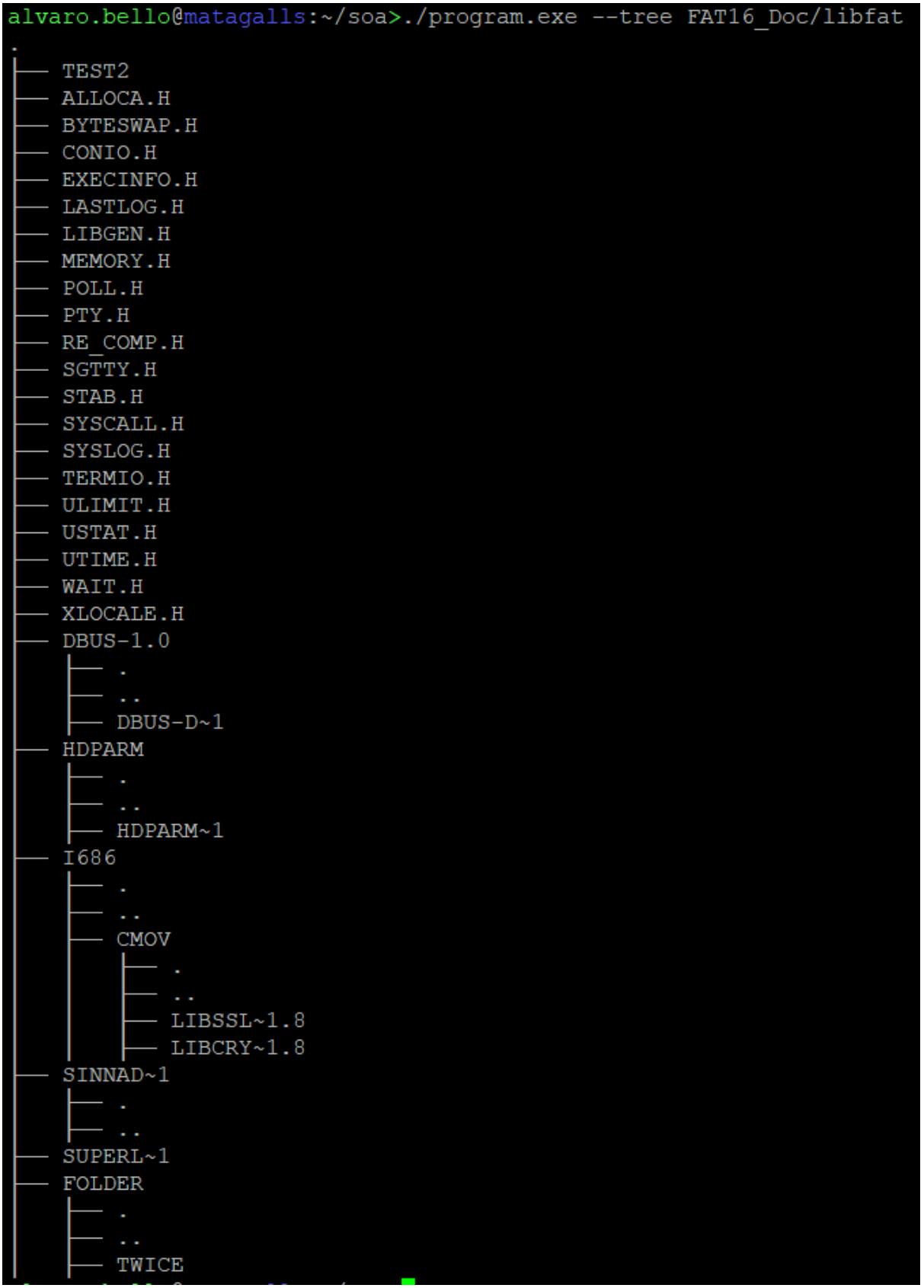
3.2 Mostrar árbol de directorios:

FAT16

DEVINfat:

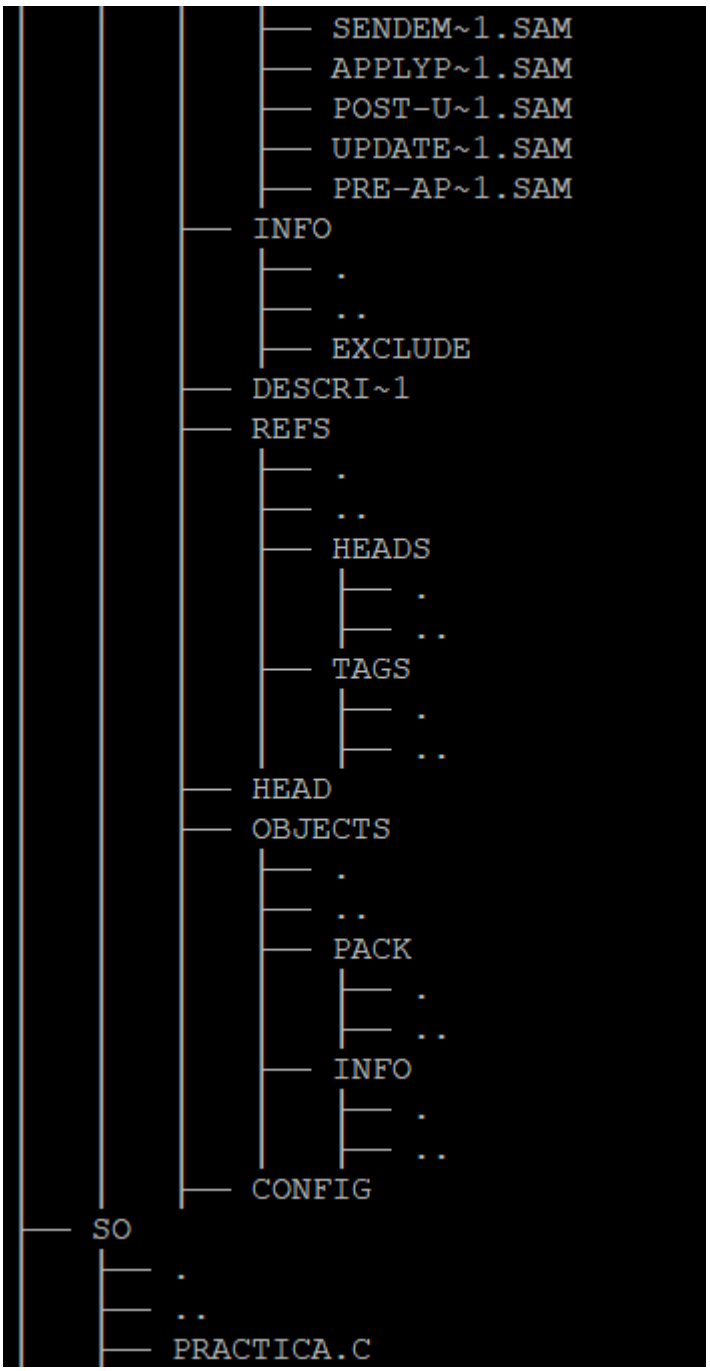


libfat:


alvaro.bello@matagalls:~/soa>./program.exe --tree FAT16_Doc/libfat
.
├── TEST2
├── ALLOCA.H
├── BYTESWAP.H
├── CONIO.H
├── EXECINFO.H
├── LASTLOG.H
├── LIBGEN.H
├── MEMORY.H
├── POLL.H
├── PTY.H
├── RE_COMP.H
├── SGTTY.H
├── STAB.H
├── SYSCALL.H
├── SYSLOG.H
├── TERMIO.H
├── ULIMIT.H
├── USTAT.H
├── UTIME.H
├── WAIT.H
├── XLOCALE.H
├── DBUS-1.0
│   ├── .
│   ├── ..
│   └── DBUS-D~1
├── HDPARM
│   ├── .
│   ├── ..
│   └── HDPARM~1
├── I686
│   ├── .
│   ├── ..
│   └── CMOV
│   ├── .
│   ├── ..
│   ├── LIBSSL~1.8
│   └── LIBCRY~1.8
├── SINNAD~1
│   ├── .
│   └── ..
├── SUPERL~1
├── FOLDER
│   ├── .
│   ├── ..
│   └── TWICE

studentfat100MB:

```
alvaro.bello@matagalls:~/soa>./program.exe --tree FAT16_Doc/studentfat100MB
.
├── ASO
│   ├── .
│   ├── ..
│   ├── SHOPPI~1.TXT
│   └── PIPES.SH
├── DONUT
│   ├── .
│   ├── ..
│   ├── DONUT.C
│   └── DONETE~1
│       ├── .
│       ├── ..
│       └── DONETE~1.C
├── PAED
│   ├── .
│   ├── ..
│   └── BOGO.C
├── PROG
│   ├── .
│   ├── ..
│   ├── SEM1
│   │   ├── .
│   │   ├── ..
│   │   └── ITERAT~1.C
│   └── SEM2
│       ├── .
│       ├── ..
│       └── RECURS~1.C
└── PRPRI
    ├── .
    ├── ..
    ├── MAIN.C
    ├── GLOBALS.H
    └── GIT~1
        ├── .
        ├── ..
        ├── BRANCHES
        │   ├── .
        │   └── ..
        └── HOOKS
            ├── .
            ├── ..
            ├── PRE-RE~1.SAM
            ├── PRE-RE~2.SAM
            ├── PUSH-T~1.SAM
            ├── PRE-ME~1.SAM
            ├── PRE-PU~1.SAM
            ├── PREPAR~1.SAM
            ├── PRE-CO~1.SAM
            ├── COMMIT~1.SAM
            └── FSMONI~1.SAM
```


SENDEM~1.SAM
APPLYP~1.SAM
POST-U~1.SAM
UPDATE~1.SAM
PRE-AP~1.SAM
INFO
.
..
EXCLUDE
DESCRI~1
REFS
.
..
HEADS
.
..
TAGS
.
..
HEAD
OBJECTS
.
..
PACK
.
..
INFO
.
..
CONFIG
SO
.
..
PRACTICA.C

Mayúsculas

Al ejecutar nuestro programa vemos los archivos del sistema mostrados correctamente pero todos en mayúsculas. Como vimos anteriormente en la sección 1.2 Mostrar árbol de directorios, esto se debe a que tal y como se indica en el apartado *"FAT 32 Byte Directory Entry Structure->DIR_Name" del documento, las entradas solo admiten carácteres en mayúsculas, por lo que si se intenta crear un archivo con carácteres en minúscula dentro de un sistema de archivos FAT16, estos se transformarán a mayúsculas.*

The DIR_Name field is actually broken into two parts+ the 8-character main part of the name, and the 3-character extension. These two parts are "trailing space padded" with bytes of 0x20.

DIR_Name[0] may not equal 0x20. There is an implied '.' character between the main part of the name and the extension part of the name that is not present in DIR_Name. Lower case characters are not allowed in DIR_Name (what these characters are is country specific).

The following characters are not legal in any bytes of DIR_Name:

- Values less than 0x20 except for the special case of 0x05 in DIR_Name[0] described above.
- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C.

Here are some examples of how a user-entered name maps into DIR_Name:

```
"foo.bar"           -> "FOO     BAR"
"FOO.BAR"           -> "FOO     BAR"
"Foo.Bar"           -> "FOO     BAR"
"foo"               -> "FOO        "
"foo."              -> "FOO        "
"PICKLE.A"          -> "PICKLE  A  "
"prettybg.big"      -> "PRETTYBGBIG"
".big"              -> illegal, DIR_Name[0] cannot be 0x20
```

EXT 2

D3VINext:

```
alvaro.bello@matagalls:~/soa>./program.exe --tree EXT2_Doc/D3V1Next
Bloque del mapa de inodos: 44
Bloque del mapa de bloques: 43
Primer bloque de inodos: 45
.
|   |__ lost+found
|   |__ pwii
|   |__ aos
|   |__ compilers
|   |__ parser
|   |__ lexer
|   |__ tree
```

lolext:

```
carla.francos@matagalls:~/soa>./program.exe --tree EXT2_Doc/lolext
Bloque del mapa de inodos: 43
Bloque del mapa de bloques: 42
Primer bloque de inodos: 44
.
|   |__ lost+found
|   |__ kda
|   |__ ahri
|   |   |__ ahri_bio
|   |   |__ ahri_lore
```

studentext100MB:

```
carla.francos@matagalls:~/soa>./program.exe --tree EXT2_Doc/studentext100MB
Bloque del mapa de inodos: 9
Bloque del mapa de bloques: 8
Primer bloque de inodos: 10
.
|   |__ lost+found
|   |__ ASO
|   |   |__ shopping_list.txt
|   |   |__ pipes.sh
|   |__ donut
|   |   |__ donut.c
|   |   |__ .donete
|   |   |   |__ donete_pro.c
|   |__ PAED
|   |   |__ bogo.c
|   |__ Prog
|   |   |__ sem1
|   |   |   |__ iterative_fibonacci.c
|   |   |__ sem2
|   |   |   |__ recursive_fibonacci.c
|   |__ PrPrI
|   |   |__ main.c
|   |   |__ globals.h
|   |   |__ .git
|   |   |   |__ branches
|   |   |   |__ hooks
|   |   |   |   |__ pre-rebase.sample
|   |   |   |   |__ pre-receive.sample
|   |   |   |   |__ push-to-checkout.sample
|   |   |   |   |__ pre-merge-commit.sample
|   |   |   |   |__ pre-push.sample
|   |   |   |   |__ prepare-commit-msg.sample
|   |   |   |   |__ pre-commit.sample
|   |   |   |   |__ commit-msg.sample
|   |   |   |   |__ fsmonitor-watchman.sample
|   |   |   |   |__ sendemail-validate.sample
|   |   |   |   |__ applypatch-msg.sample
|   |   |   |   |__ post-update.sample
|   |   |   |   |__ update.sample
|   |   |   |   |__ pre-applypatch.sample
|   |   |   |__ info
|   |   |   |   |__ exclude
|   |   |   |__ description
|   |   |   |__ refs
|   |   |   |   |__ heads
|   |   |   |   |__ tags
|   |   |   |__ HEAD
|   |   |   |__ objects
|   |   |   |   |__ pack
|   |   |   |   |__ info
|   |   |   |__ config
|   |__ SO
|   |   |__ practica.c
```

3.3 Mostrar contenido de un archivo:

FAT16

DEVINfat:

```
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/DEVINfat COMPUS/EXTRA/TODOLIST.TXT
TODO LIST:

1. START P1!!
2. BUY CHATGPT4 :/
3. CRY
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/DEVINfat COMPUS/P3/STATEM~1.TXT
Statement

Implement a system that allows CyberSalle Systems Corp. to monitor and collect your data!
```

libfat:

```
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/libfat ALLOCA.H
/* Copyright (C) 1992, 1996, 1997, 1998, 1999 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, write to the Free
   Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307 USA.  */

#ifndef _ALLOCA_H
#define _ALLOCA_H       1

#include <features.h>

#define __need_size_t
#include <stddef.h>

__BEGIN_DECLS

/* Remove any previous definitions.  */
#undef  alloca

/* Allocate a block that will be freed when the calling function exits.  */
extern void *alloca (size_t __size) __THROW;

#ifdef  __GNUC__
# define alloca(size)   __builtin_alloca (size)
#endif /* GCC.  */

__END_DECLS

#endif /* alloca.h */
```

```
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/libfat UTIME.H
/* Copyright (C) 1991, 92, 96, 97, 98, 99, 2004 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, write to the Free
   Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307 USA.  */

/*
 *      POSIX Standard: 5.6.6 Set File Access and Modification Times  <utime.h>
 */

#ifndef _UTIME_H
#define _UTIME_H        1

#include <features.h>

__BEGIN_DECLS

#include <bits/types.h>

#if defined __USE_XOPEN || defined __USE_XOPEN2K
# define __need_time_t
# include <time.h>
#endif

/* Structure describing file times.  */
struct utimbuf
  {
    __time_t actime;            /* Access time.  */
    __time_t modtime;           /* Modification time.  */
  };

/* Set the access and modification times of FILE to those given in
   *FILE_TIMES.  If FILE_TIMES is NULL, set them to the current time.  */
extern int utime (__const char *__file,
                  __const struct utimbuf *__file_times)
     __THROW __nonnull ((1));

__END_DECLS

#endif /* utime.h */
```

studentfat100MB:

```
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/studentfat100MB SO/PRACTICA.C
#include <stdio.h>

// TODO: Absolutely everything :)

// Dear lord, save us again!

carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/studentfat100MB DONUT/DONETE~1/DONETE~1.C
#include <stdio.h>

// TODO: WHY IT DOESN'T WOOOOOOOOOOOOOOOOOOORK!!!!!!!!!!!!!!!!

_,x,y,o       ,N;char       b[1840]       ;p(n,c)
{for(;n       --;x++)       c==10?y       +=80,x=
o-1:x>=       0?80>x?       c!='~'?       b[y+x]=
c:0:0:0       ;}c(q,l       ,r,o,v)       char*l,
      *r;{for       (;q>=0;       )q=("A"       "YLrZ^"
      "w^?EX"           "novne"     "bYV"       "dO}LE"
      "{yWlw"      "Jl_Ja|[ur]zovpu"   ""       "i]e|y"
      "ao_Be"   "osmIg}r]]r]m|wkZU}{O}"         "xys]]\
x|ya|y"        "sm||{uel}|r{yIcsm||ya[{uE"  "{qY\
w|gGor"      "VrVWioriI}Qac{{BIY[sXjjsVW]aM"  "T\
tXjjss"     "sV_OUkRUlSiorVXp_qOM>E{BadB"[_/6  ]-
62>>_++    %6&1?r[q]:l[q])-o;return q;}E(a){for (
      o= x=a,y=0,_=0;1095>_;)a= " <.,`'/)(\n-"  "\\_~"[
      c  (12,"!%*/')#3"  ""     "+-6,8","\"(.$" "01245"
      " &79",46)+14],  p(""       "#$%&'()0:439 "[ c(10
      , "&(*#,./1345" ,"')"       "+%-$02\"! ", 44)+12]
-34,a);  }main(k){float     A=0,B= 0,i,j,z[1840];
puts(""  "\x1b[2J");;;      for(;; ){float e=sin
(A), n=  sin(B),g=cos(      A),m=  cos(B);for(k=
0;1840>   k;k++)y=-10-k/    80   ,o=41+(k%80-40
      )* 1.3/y+n,N=A-100.0/y,b[k]=".#"[o+N&1],  z[k]=0;
      E(  80-(int)(9*B)%250);for(j=0;6.28>j;j   +=0.07)
      for  (i=0;6.28>i;i+=0.02){float c=sin(    i),  d=
      cos(  j),f=sin(j),h=d+2,D=15/(c*h*e+f     *g+5),l
=cos(i)        ,t=c*h*g-f*e;x=40+2*D*(l*h*  m-t*n
),y=12+       D  *(l*h*n+t*m),o=x+80*y,N  =8*((f*
e-c*d*g       )*m   -c*d*e-f*g-l*d*n)     ;if(D>z
[o])z[o       ]=D,b[     o]=" ."          ".,,-+"
      "+=#$@"       [N>0?N:       0];;;;}       printf(
      "%c[H",       27);for       (k=1;18       *100+41
      >k;k++)       putchar       (k%80?b       [k]:10)
      ;;;;A+=       0.053;;       B+=0.03       ;;;;;}}
```

4. Estimación del tiempo de dedicación para cada fase del proyecto

Investigación (lectura y comprensión de la documentación)	10 horas
Identificación y recopilación de información del sistema de archivos (FAT16)	4 horas
Identificación y recopilación de información del sistema de archivos (EXT2)	2 horas
Mostrar árbol de directorios (FAT16)	8 horas
Mostrar árbol de directorios (EXT2)	20 horas
Mostrar contenido de un archivo (FAT16)	5 horas
Testing	6 horas
Memoria	8 horas
TOTAL	**63 horas**

En este proyecto se ha realizado una gran investigación sobre cómo funcionan los dos sistemas de archivos y cómo tienen organizados sus datos, siendo capaz de leer y comprender dos “papers” técnicos sobre éstos. Además, se puede apreciar una diferencia entre las horas dedicadas a realizar el código relacionado con EXT2 que con FAT16, ya que el sistema de archivos EXT2 tiene los datos organizados de una forma algo más compleja.

Una vez se tuvo todo en funcionamiento, se realizó un testing exhaustivo haciendo varias pruebas para asegurarnos del correcto funcionamiento del proyecto.

6. Conclusiones

Durante el desarrollo de esta práctica, se ha adquirido un conocimiento profundo acerca de cómo funcionan y se estructuran internamente dos de los sistemas de archivos más utilizados: FAT16 y EXT2. A lo largo de las distintas fases, enfrentado desafíos técnicos que han permitido desarrollar habilidades prácticas de lectura de archivos oficiales, manipulación de estructuras de datos específicas de cada sistema de archivos y comprensión de conceptos avanzados de sistemas operativos como el recorrido recursivo de árboles de directorios y la gestión de punteros indirectos.

También se ha sido capaz de detectar la diferencia en la complejidad de ambos sistemas. Debido a que mientras FAT16 presenta una estructura más sencilla, basada en cadenas de clusters enlazados a través de la tabla FAT, el sistema EXT2 incorpora mecanismos más sofisticados y complejos como bloques directos, indirectos y triple indirectos, que han supuesto un reto adicional a la hora de implementar el recorrido de sus árboles de directorios. También se ha aprendido la importancia de ajustar correctamente el tamaño de las estructuras de datos, y cómo pequeños errores en los offsets pueden causar lecturas incorrectas de todo el sistema de archivos o recorridos incompletos.

Finalmente, la fase de pruebas y validación permitió detectar y solventar problemas relacionados con la detección de los sistemas de archivos y la correcta interpretación de los metadatos, consolidando aún más la comprensión de los conceptos teóricos en un entorno real. Como resultado, se ha realizado un programa capaz de identificar, recorrer y mostrar con precisión la estructura de directorios de ambos sistemas de archivos, reflejando tanto el esfuerzo dedicado como el aprendizaje adquirido durante esta práctica.

7. Bibliografía

Toda la información ha sido obtenida de los siguientes documentos incluidos dentro de la carpeta del proyecto:

- FAT: General Overview of On-Disk Format
- The Second Extended File System