Documentation

File Systems Inspector

Advanced Operating Systems

Alvaro Bello Garrido

27/06/2025

Index

Project summary

The project consists of a file system inspector program capable of recognizing FAT16 and EXT2 (if a different file system is inspected, it will not be recognized and the appropriate error message will be displayed), and performing the following operations on it: display file system information, display a tree schema of the file system, and display the contents of a file within the file system.

1. Design and implementation details

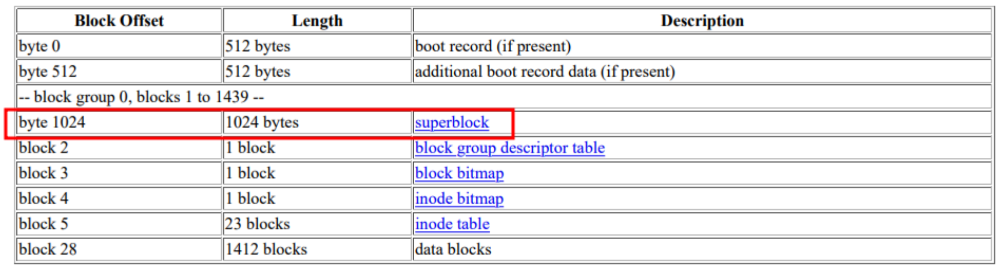
1.1 Identification and collection of file system information

First, it is essential to determine whether the present file system is of type EXT2 or FAT16. To that end, a characteristic indicator must be located in each of these systems whose consistency allows precise identification.

EXT systems identification:

According to the technical documentation for EXT systems, the superblock is located at offset 1024. This superblock is a 1024-byte structure that stores relevant information about the file system.

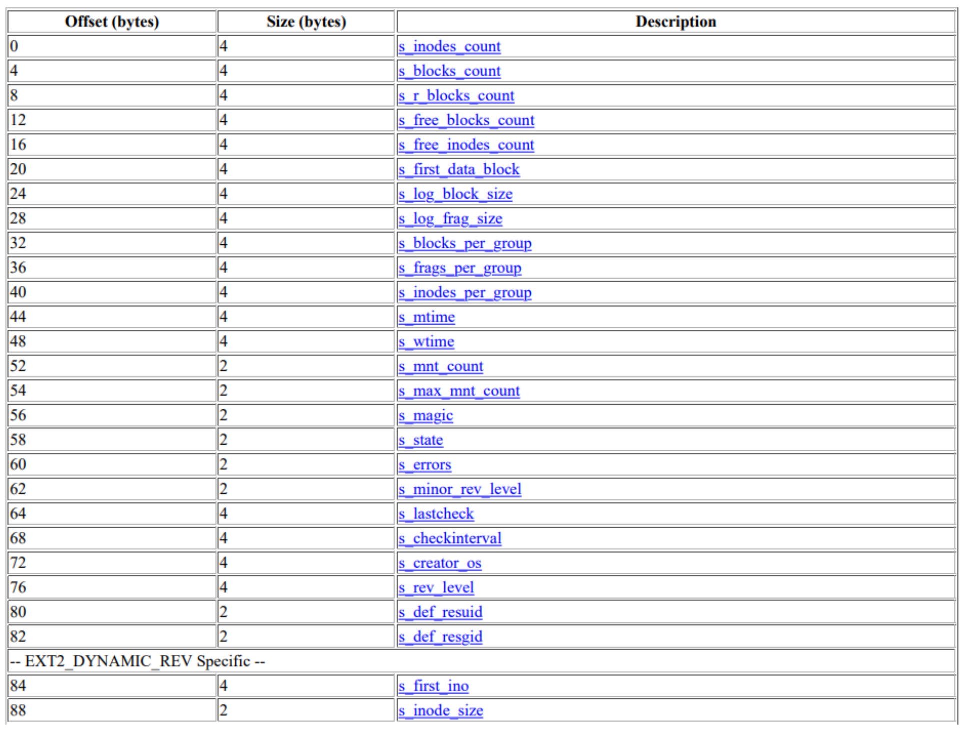
Table 3.1. Sample Floppy Disk Layout, 1KiB blocks



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

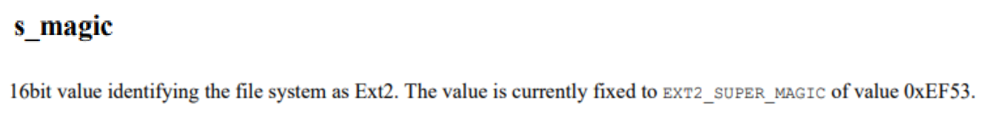
After locating the superblock, the documentation describes both its structure and its contents:

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

Analyzing each parameter of the superblock reveals that, in Ext2 file systems, the s_magic field always has the value 0xEF53. Thus, EXT2 detection can be performed by verifying the position of this parameter and its correspondence with the specified value.

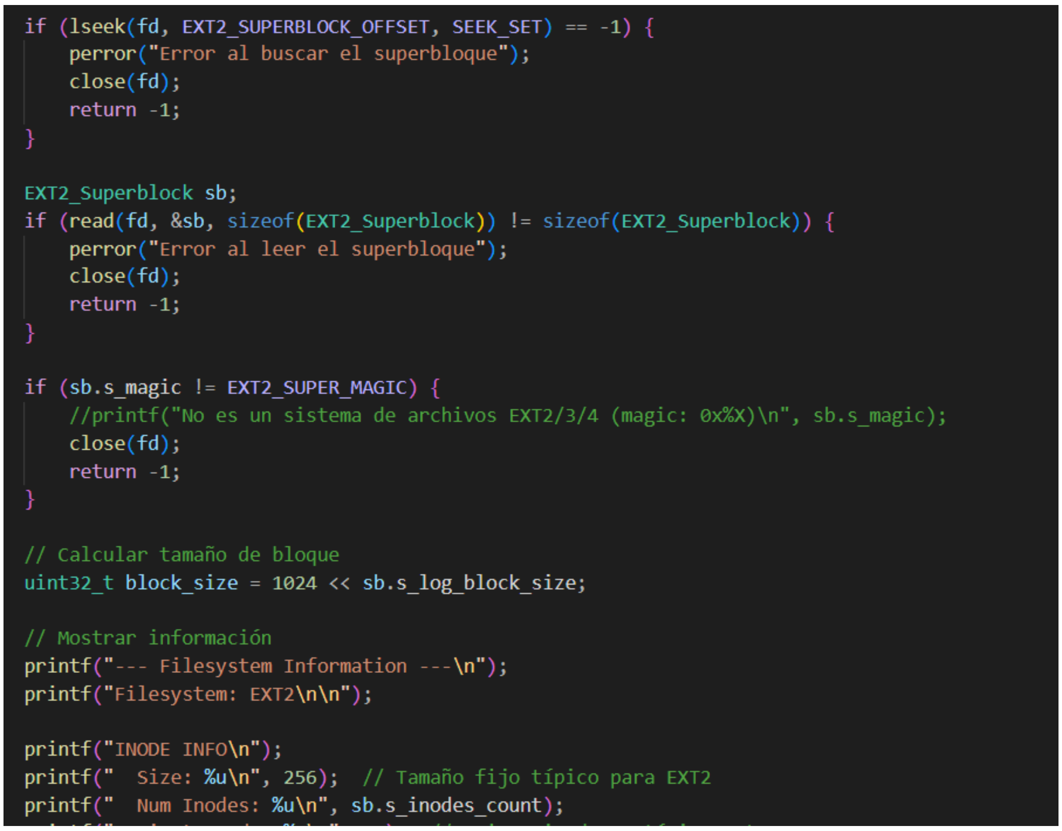


s_magic

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

Once this information is obtained, the identification of EXT2 systems in the proposed implementation is carried out as follows:

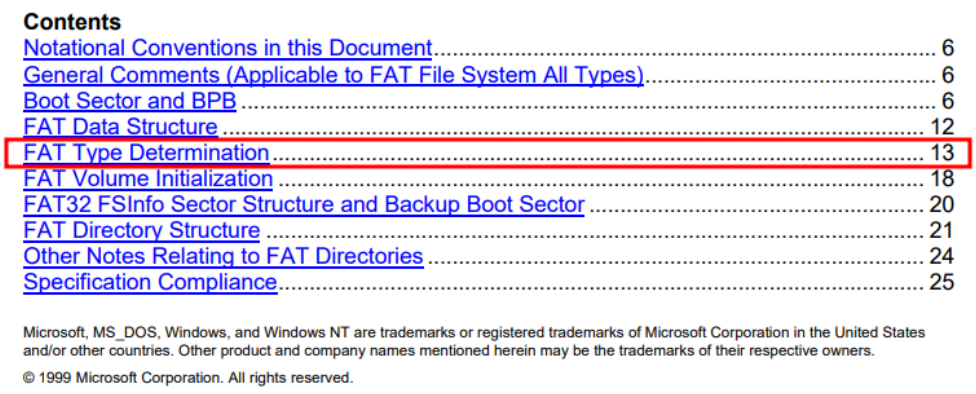
1. Read the superblock by seeking to its initial position using the ‘lseek’ function.
2. Store the relevant superblock information in a custom structure (EXT2_Superblock) containing all superblock parameters.
3. Check the value of the s_magic field to determine whether it is an EXT2 system and, if so, display the corresponding information.



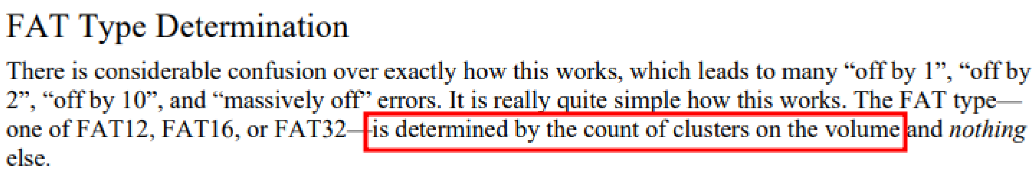
```
if (lseek(fd, EXT2_SUPERBLOCK_OFFSET, SEEK_SET) == -1) {
    perror("Error al buscar el superbloque");
    close(fd);
    return -1;
}

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);
```

FAT systems identification:

For identifying FAT16 systems, the technical documentation specifies in the “FAT Type Determination” section the procedure to follow:

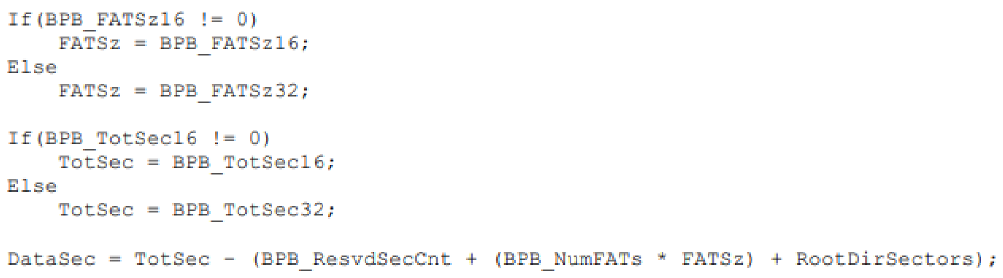

Contents

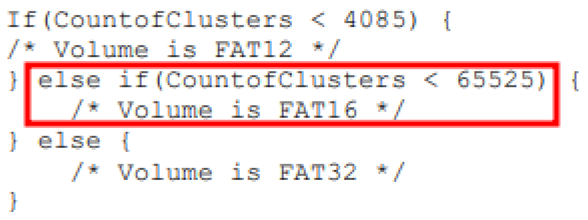
This section clearly states that determining the FAT file system type requires calculating the number of clusters.


FAT Type Determination

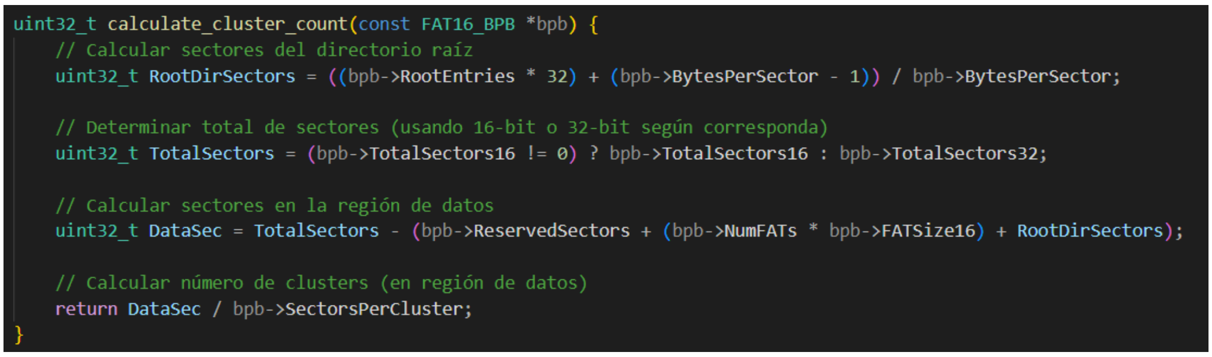
There is considerable confusion over exactly how this works, which leads to many “off by 1”, “off by 2”, “off by 10”, and “massively off” errors. It is really quite simple how this works. The FAT type—one of FAT12, FAT16, or FAT32—is determined by the count of clusters on the volume and *nothing* else.

It then explains how to perform this calculation and provides the valid cluster-count range for FAT16 file systems.


```
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);
```

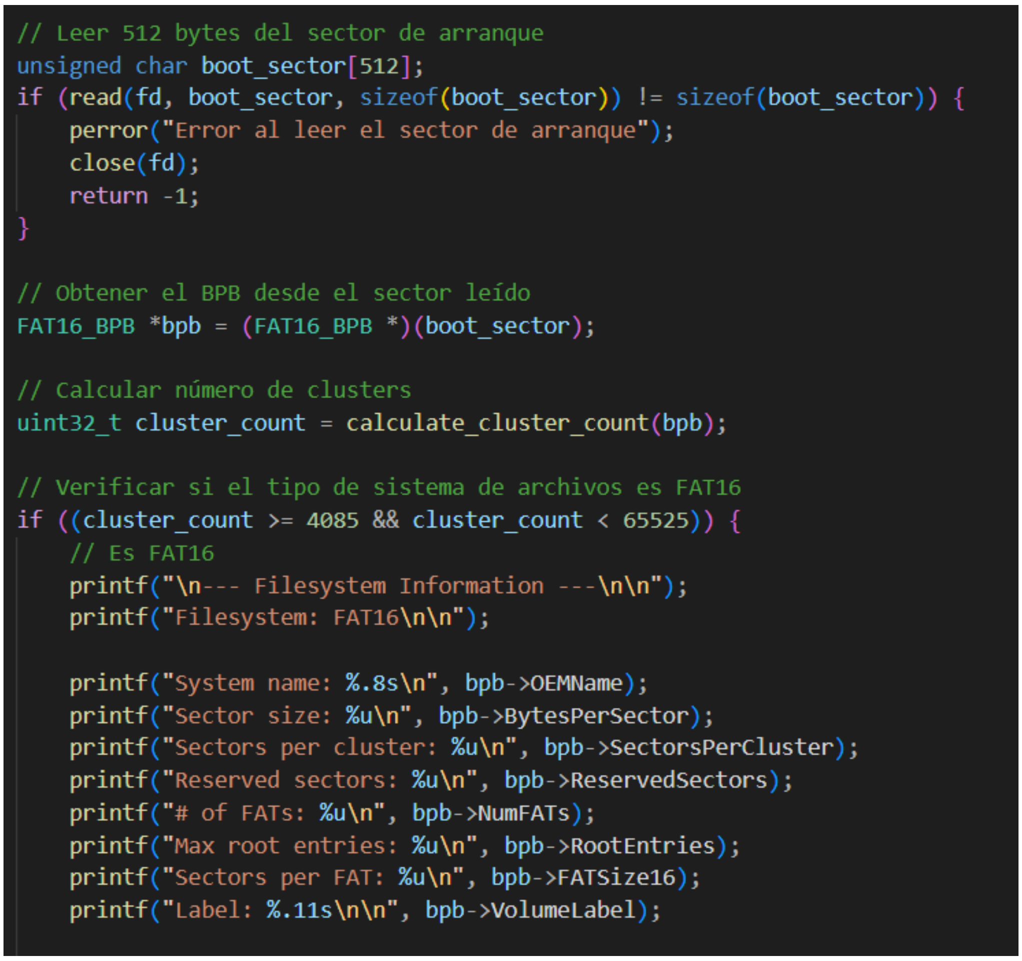

```
If(CountofClusters < 4085) {
/* Volume is FAT12 */
} else if(CountofClusters < 65525) {
    /* Volume is FAT16 */
} else {
    /* Volume is FAT32 */
}
```

With this information, a specific function was designed to calculate the number of 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;
}
```

Finally, using a structure (FAT16_BPB) that stores the information read from the BPB (BIOS Parameter Block), the aforementioned function is used to verify whether the file system corresponds to FAT16, in accordance with the technical documentation:



```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);
```

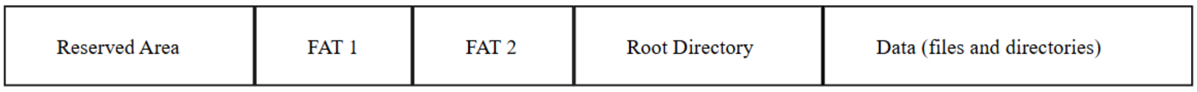
1.2 Display directory tree

For the second phase of the project, the files and directories of the system must be displayed. To that end, recursive functions have been implemented for both FAT16 and EXT2, allowing a complete traversal of the hierarchical directory structure from the root.

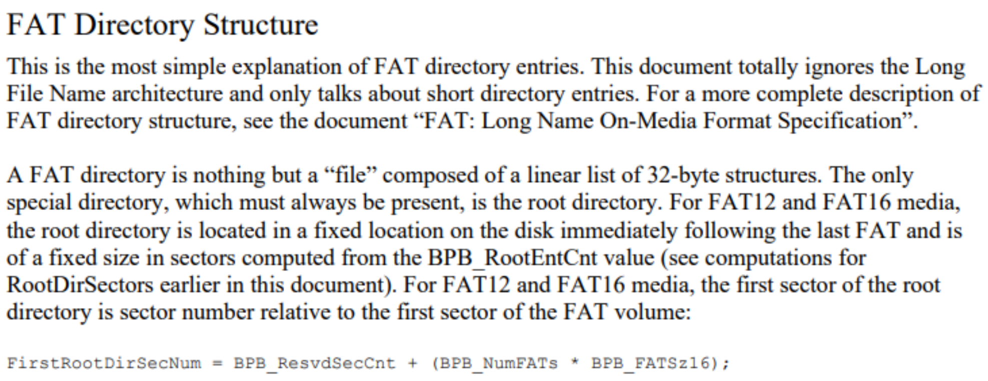
FAT16:

Starting with FAT16, the following scheme is known from "https://filesystems.jiahuichen.dev/fat-16/", where we can see that to read the directory tree we must:

1. Read the root directory with its corresponding entries.
2. Process these entries recursively, accessing them via their cluster chains by following the FAT (File Allocation Tables).



The process begins by identifying the starting sector of the root directory, using the information described in the "FAT Directory Structure" section.



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:

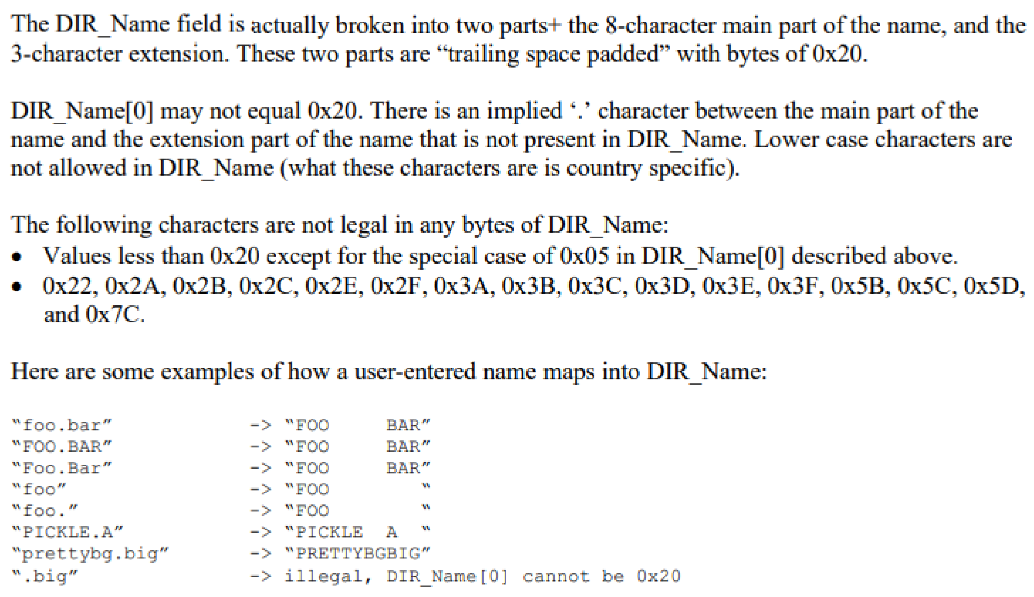
```
FirstRootDirSecNum = BPB_ResvdSecCnt + (BPB_NumFATs * BPB_FATSz16);
```

Once the address of this sector is obtained, it is possible to calculate the total size of the root directory using the expression (RootSectors * BytesPerSector), where "RootSectors" is the number of sectors it occupies, a value that can be computed according to the technical documentation in the "FAT Data Structure" section:

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

Once the sector and size of the root directory have been determined, it is read and all its entries are displayed. If an entry corresponds to a subdirectory, the exploration continues recursively.

To display the entries, the print_indent() function prints the indent levels corresponding to the depth at which the file is located, in order to display the directory tree in a structured, hierarchical manner. Meanwhile, the print_entry_name() function displays the entry's name and extension, following the specified format of 8 characters for the name and 3 characters for the extension (leaving blank spaces in the middle if the 8 name characters are not used), according to the information indicated in the "FAT 32-Byte Directory Entry Structure -> DIR_Name" section (despite being a section for FAT32 systems, FAT16 uses the same structure for directory entries).



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).

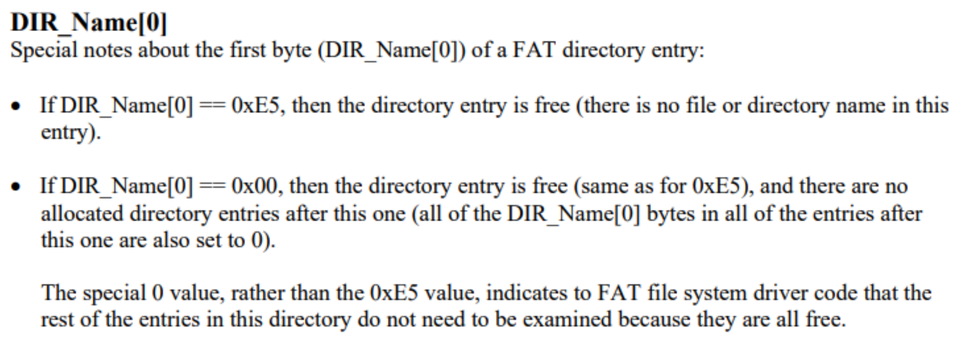
The following characters are not legal in any bytes of DIR_Name:

- 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
```

In addition, the process checks whether an entry is empty (0xE5) or is the last in the directory (0x00), ending the exploration in such cases. Entries corresponding to long file names (attribute value 0x0F) are omitted, since handling these cases is outside the scope of the original technical document.

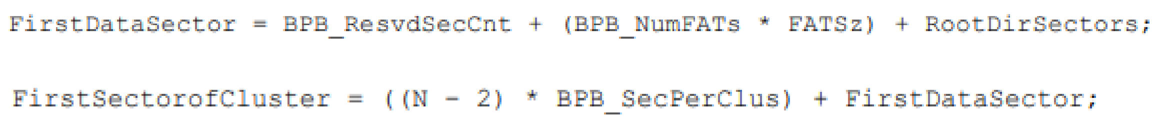


DIR_Name[0]
Special notes about the first byte (DIR_Name[0]) of a FAT directory entry:

- 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.

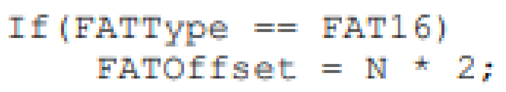
For subdirectories other than the root directory, the location of their first sector is determined using the formula indicated in the “FAT Data Structure” section of the documentation (using the directory’s first cluster as N).



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

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

Next, the first cluster of the directory is read and its entries are displayed recursively just as with the root directory.

After iterating through these entries, clusters are traversed via the FAT until reaching a cluster with the value 0xFFF8, which indicates the end of the cluster chain. In this process, to obtain the next cluster in the chain, the “FAT Type Determination” section indicates that the position within the FAT for the current cluster must be addressed multiplied by 2 (because each FAT entry occupies 2 bytes).



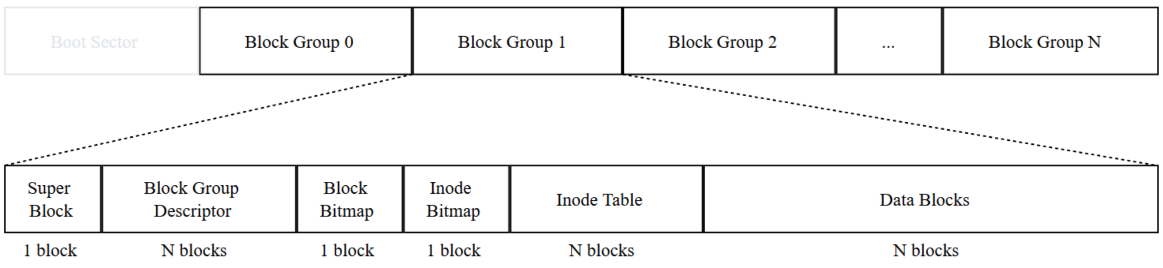
```
If(FATType == FAT16)
    FATOffset = N * 2;
```

The process is repeated until the entire structure has been explored.

EXT2:

Continuing with EXT2, the next scheme provided by "https://filesystems.jiahuichen.dev/ext-2/" shows that to read the directory tree we need to:

1. Read the root inode and obtain its direct and indirect blocks.
2. Traverse all direct and indirect blocks of the directory.
3. Iterate over each directory entry within those blocks, obtaining the inode number for each file or subdirectory.
4. For entries that are subdirectories, recursively apply the reading process on the associated inode and its blocks.



The process begins by reading the root inode, which in ext2 is always inode number 2, as noted in the "Defined Reserved Inodes" section of the documentation.

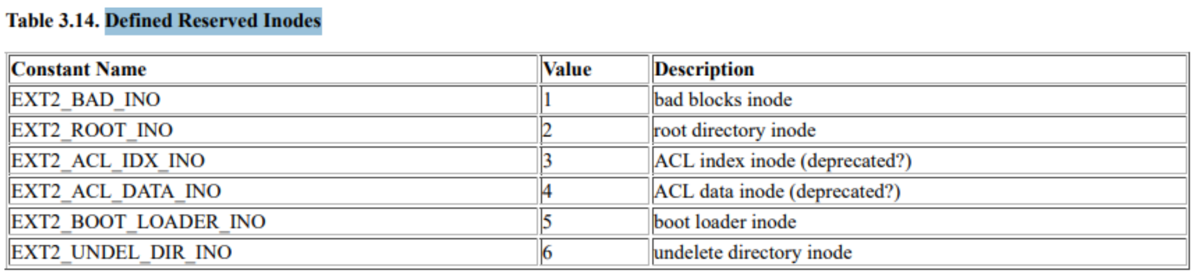


Table 3.14. Defined Reserved Inodes

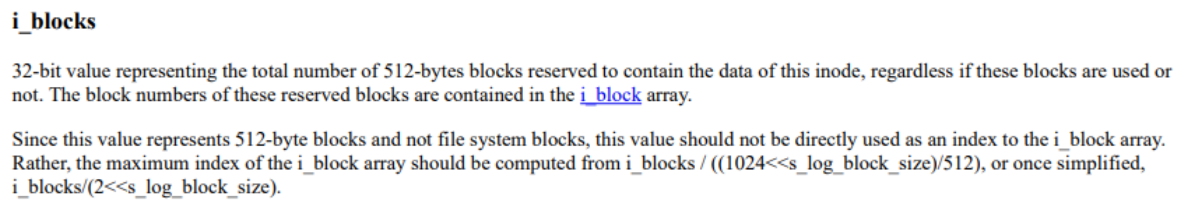
Constant Name	Value	Description
EXT2_BAD_INO	1	bad blocks inode
EXT2_ROOT_INO	2	root directory inode
EXT2_ACL_IDX_INO	3	ACL index inode (deprecated?)
EXT2_ACL_DATA_INO	4	ACL data inode (deprecated?)
EXT2_BOOT_LOADER_INO	5	boot loader inode
EXT2_UNDEL_DIR_INO	6	undelete directory inode

To achieve this, the read_inode() function computes the position of this inode within the inode table using the formula: inode_offset = table_offset + (inode_num - 1) * sb->s_inode_size

where:

- table_offset is the offset of the inode table.
- The inode number is reduced by 1, since inodes start at number 1 as indicated in the "Locating an Inode" section of the documentation.
- The inode table is located in the block specified by the bg_inode_table field of the group descriptor.

After accessing the root inode, its i_block field is read; this field contains 15 entries (direct, single-indirect, double-indirect, and triple-indirect blocks). This structure is detailed in the "Inode Structure" section of the technical documentation.



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).

The print_directory_recursive() function is used to recursively access these inode blocks, first processing direct blocks and displaying their entries; if subdirectories are found, the traversal is repeated recursively.

If the inode has indirect blocks, the reading remains recursive:

- The level-1 indirect block contains pointers to data blocks.

- The level-2 (double-indirect) block points to indirect blocks, which in turn point to data blocks.

- The level-3 (triple-indirect) block points to double-indirect blocks.

The process_indirect_blocks() function implements this recursive traversal while respecting the hierarchy.

Additionally, the process_directory_block() function processes and displays the entries contained in direct data blocks. It sequentially iterates through the entries in the block (each with dynamic size rec_len), printing names with indentation appropriate to the current depth in the directory tree. During this process, the special "." and ".." entries are filtered out; when encountering a subdirectory, the corresponding inode is fetched and traversal continues recursively. In this way, the function enables a complete and structured traversal of the directory hierarchy, visually reflecting the file system's tree architecture.

1.3 Display file contents

The final phase of the project aims to display the contents of a specific file stored in a FAT16 file system (this functionality has not been implemented for EXT2). This operation involves several steps related to locating the file and accessing its data clusters.

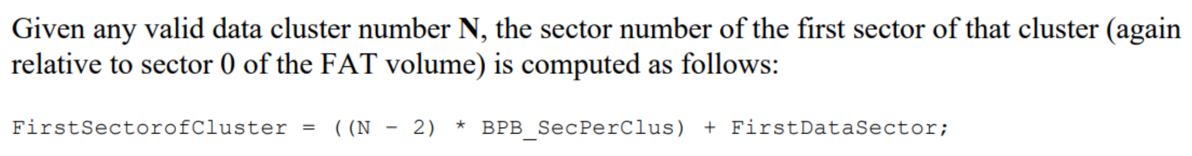
First, the user-provided filename is reformatted using the format_name function, which splits the name into two parts: an 8-character main name and a 3-character extension, padding unused characters with spaces and converting to uppercase, as specified by FAT16 directory entries.

After adapting the target name, the corresponding entry is located in the Root Directory based on the BIOS Parameter Block (BPB):

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

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

The file search is performed by the find_entry function, which sequentially reads the 32-byte entries present in the root directory (cluster 0) or the various clusters of a subdirectory. This exploration follows the FAT chain until reaching an End Of Chain (EOC) value greater than or equal to 0xFFF8, comparing for each entry the first 11 bytes against the already formatted name. After scanning all Root Directory entries, if a match is found, the starting cluster is obtained from bytes at offsets 26–27, while bytes 28–31 provide the file size in bytes.

Next, the dump_file function iterates through the file's clusters, continuing while the value of the next cluster is less than 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;
```

For each cluster, the amount of data to read is computed as SectorsPerCluster * BytesPerSector, or the remaining bytes if on the final cluster. The read data is sent directly to stdout for display, and the next cluster in the chain is determined by reading 2 bytes at the position calculated with the next formula:

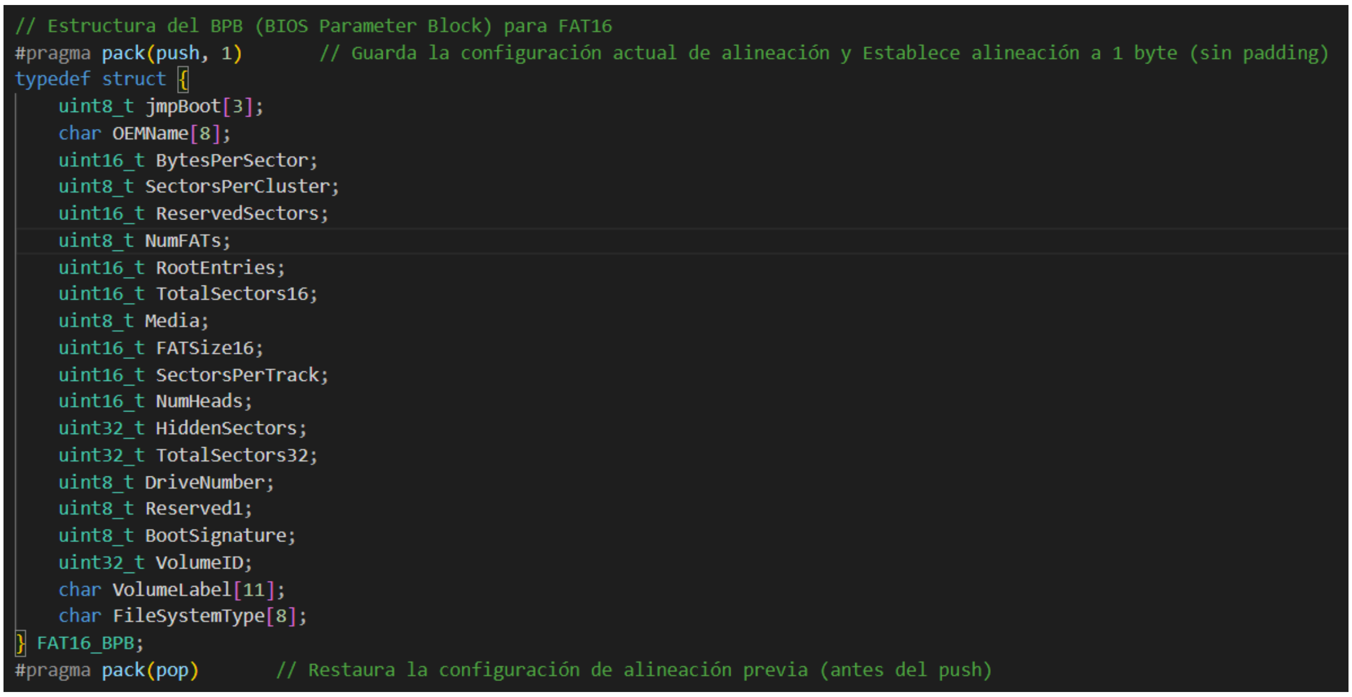
```
off_t fat_off = bpb->ReservedSectors * bpb->BytesPerSector + cur * 2;
```

2. Data structures used and justification

2.1 Identification and collection of file system information:

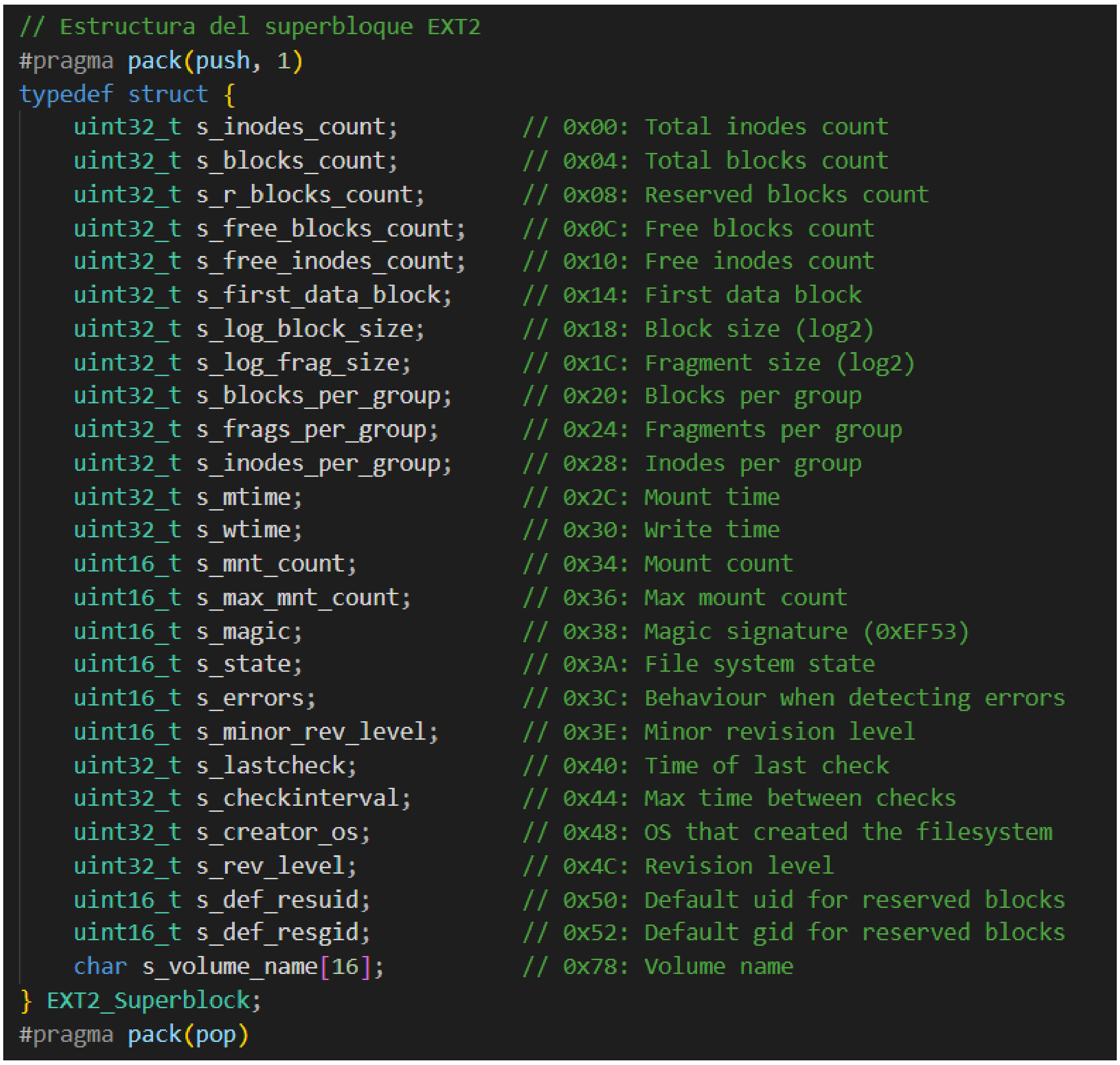
For the first phase, two data structures were used: one to read the BPB (BIOS Parameter Block) of FAT16 file systems, and another to read the superblock of EXT2 file systems. Each field in these structures was assigned the size specified in the documentation.

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)
```

Additionally, note the use of #pragma before and after declaring each structure:

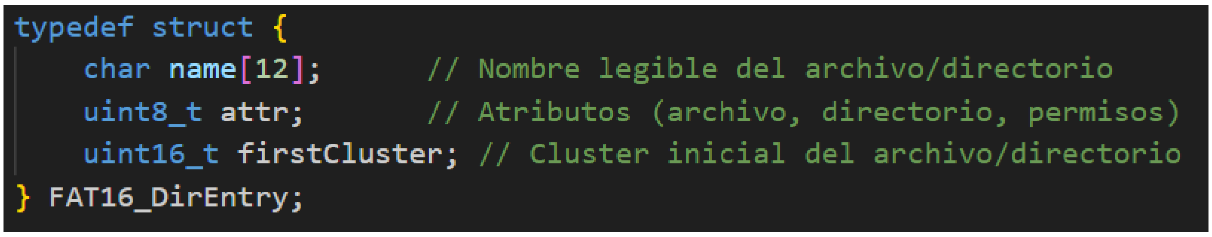
- #pragma pack(push, 1): sets 1-byte alignment (no padding) and pushes the current alignment configuration onto the stack.
- #pragma pack(pop): pops and restores the previous alignment configuration.

This is done to disable automatic structure padding, because by default the compiler may insert padding to align fields according to the target architecture and optimize memory access. Such alignment introduces internal gaps and can change the offsets of each field within the structure, which would invalidate the documented offsets.

2.2 Display directory tree:

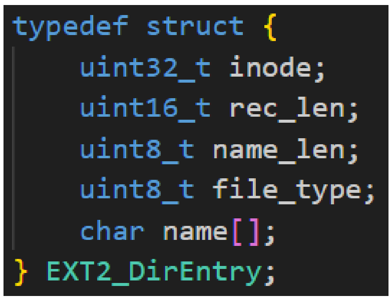
For this phase, additional structures were introduced to store, in a simpler and more organized way, the information of the elements involved in the recursive traversal of the directory tree. This allows storing directory entries for both FAT16 and EXT2, along with inode information and group descriptors in EXT2.

FAT16:

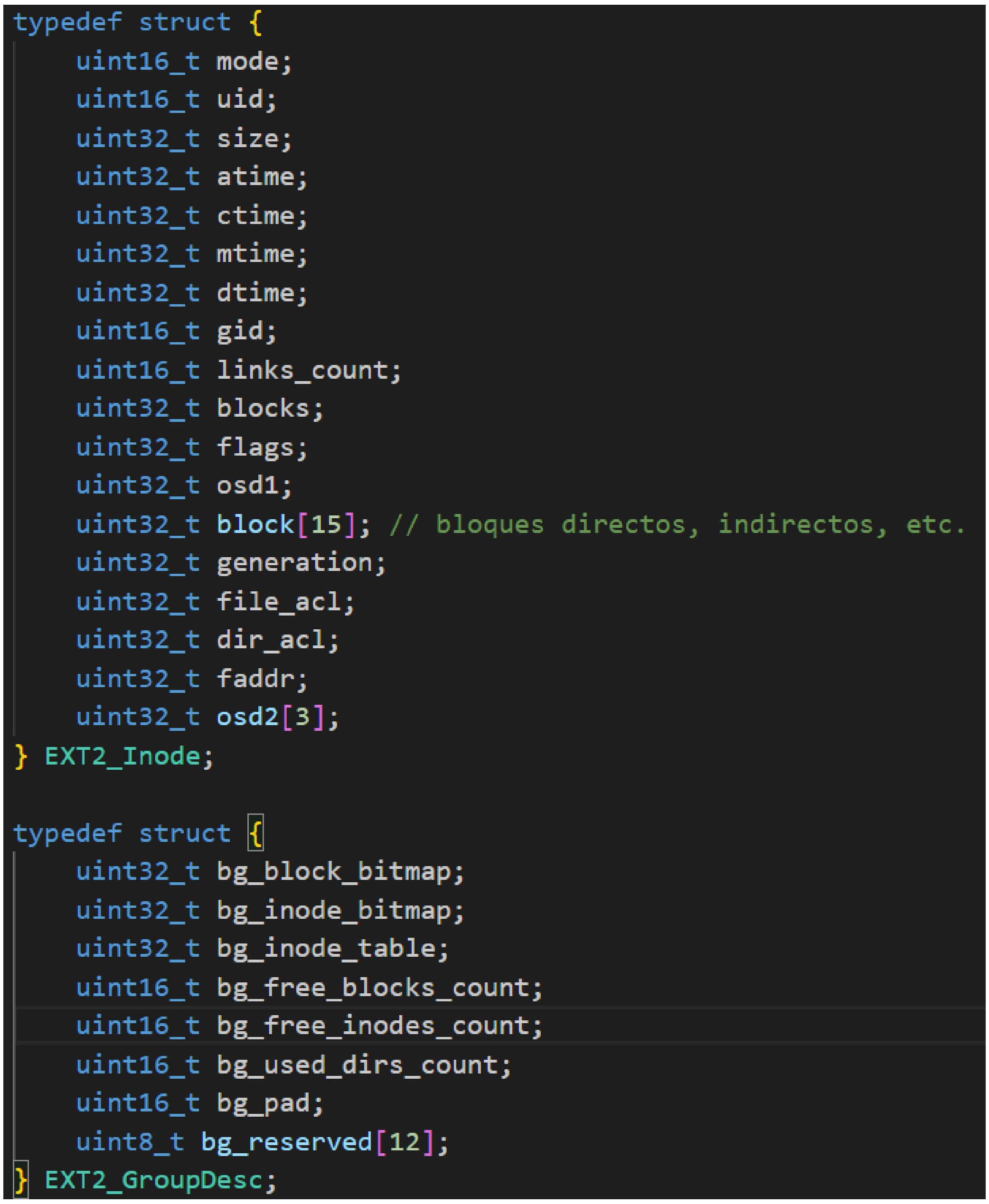


```c
typedef struct {
    char name[12];      // Nombre legible del archivo/directorio
    uint8_t attr;       // Atributos (archivo, directorio, permisos)
    uint16_t firstCluster; // Cluster inicial del archivo/directorio
} FAT16_DirEntry;
```

EXT2:



```c
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;
```

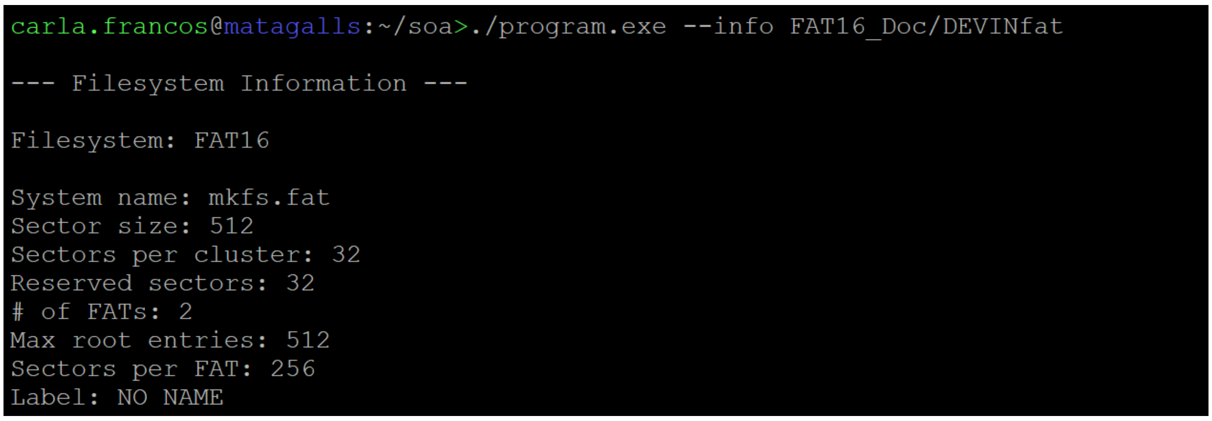
3. Different test results for both file system types

Below are the results of the tests performed with file systems of various sizes, both FAT16 and EXT2, using the options offered by the program:

3.1 Identification and collection of file system information:

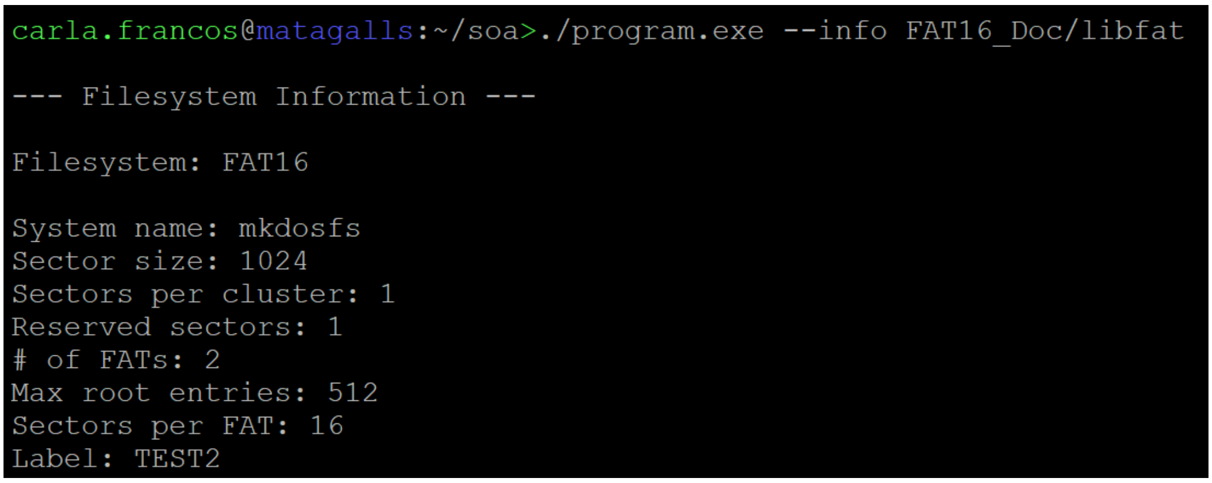
FAT16

DEVINfat:



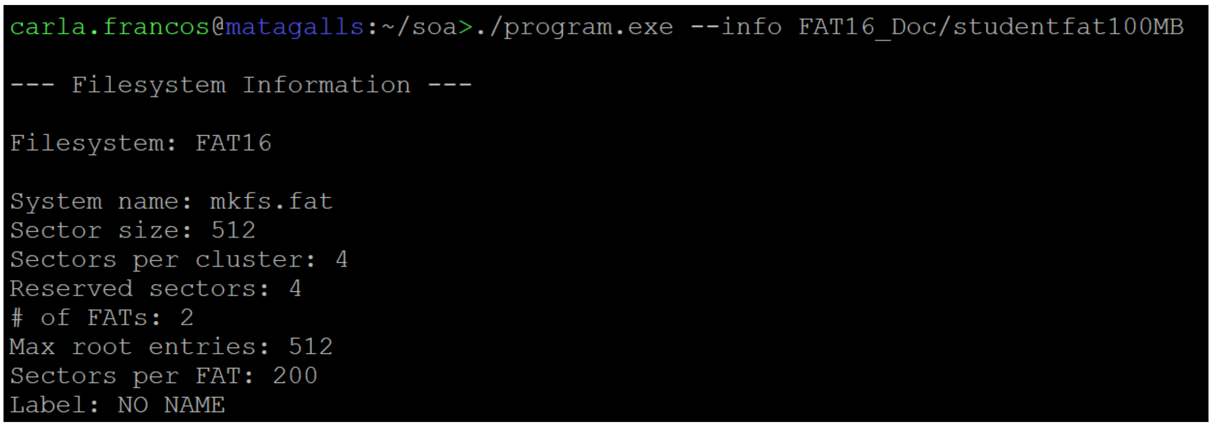
```
carla.francos@matagalls:~/soa>./program.exe --info FAT16_Doc/DEVINfat

--- 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
```

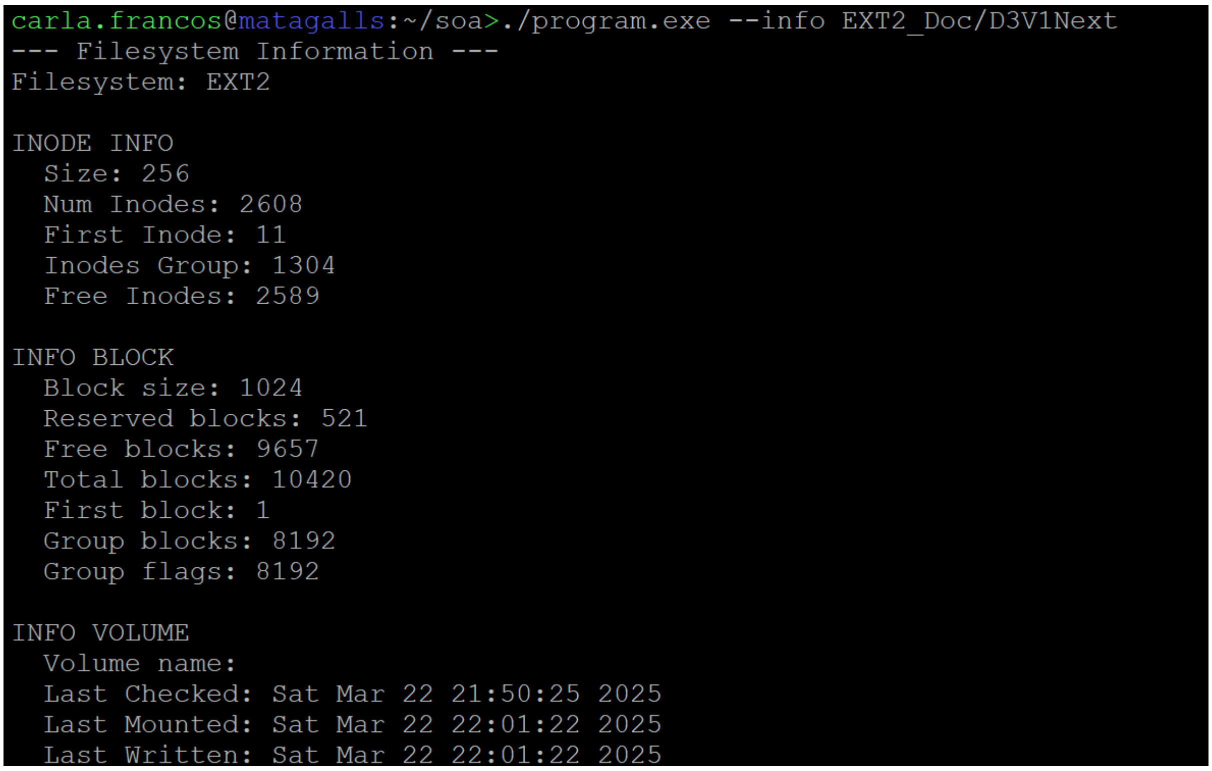
studentfat100MB:



```
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
```

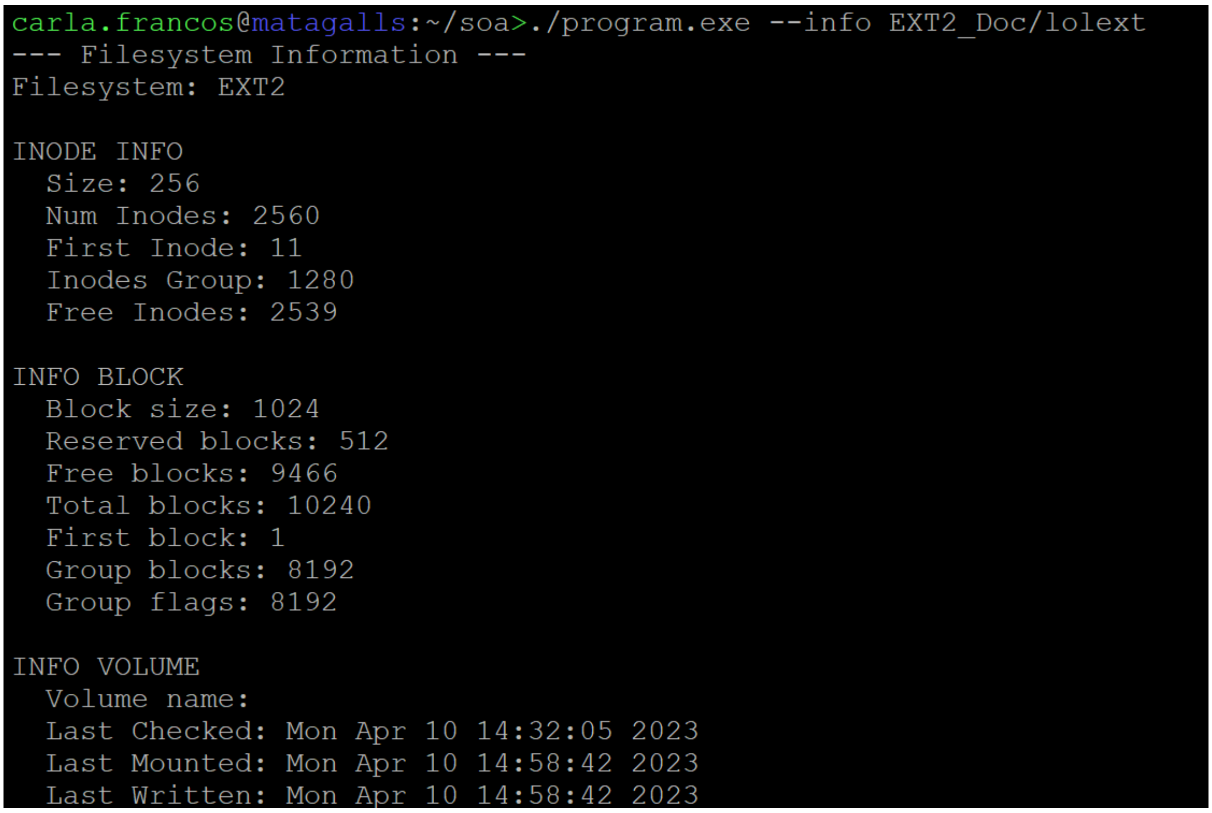
EXT2

D3VINext:



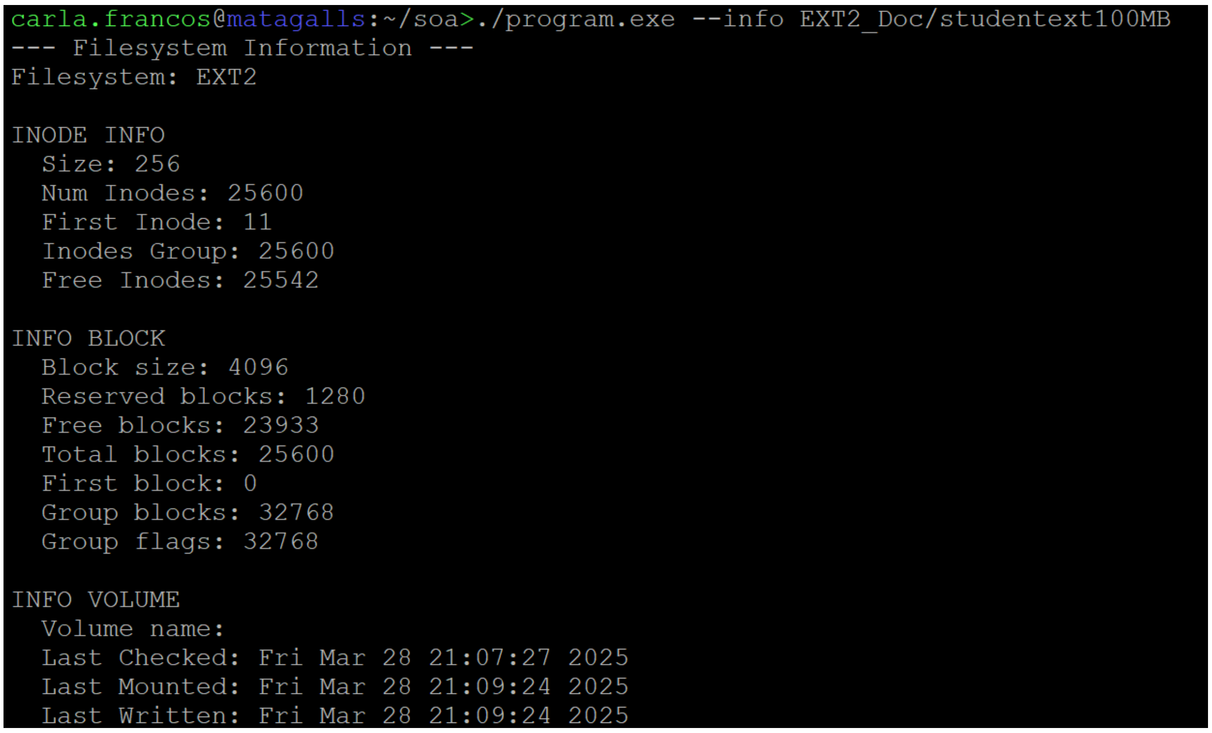
```
carla.francos@matagalls:~/soa>./program.exe --info EXT2_Doc/D3V1Next
--- Filesystem Information ---
Filesystem: EXT2

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:

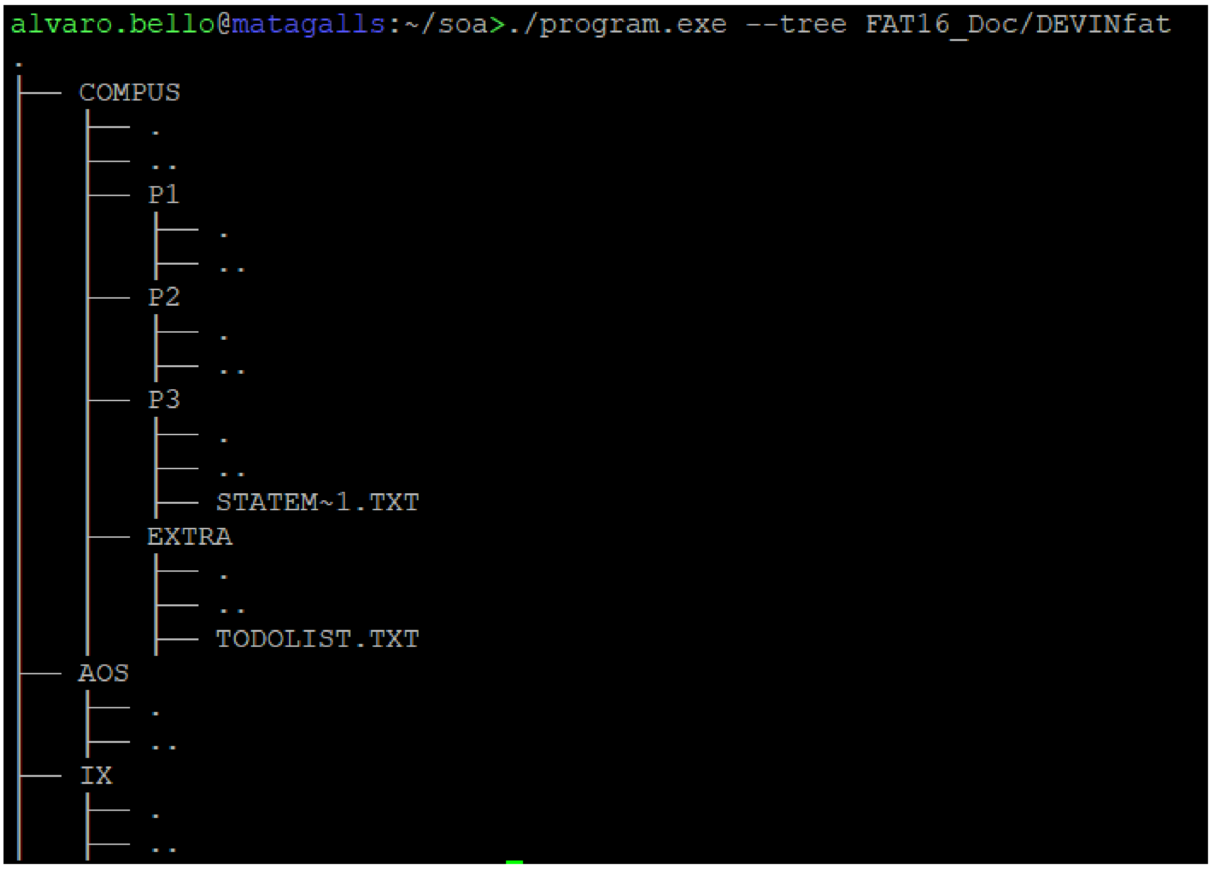


```
carla.francos@matagalls:~/soa>./program.exe --info EXT2_Doc/studentext100MB
--- Filesystem Information ---
Filesystem: EXT2

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 Display directory tree:

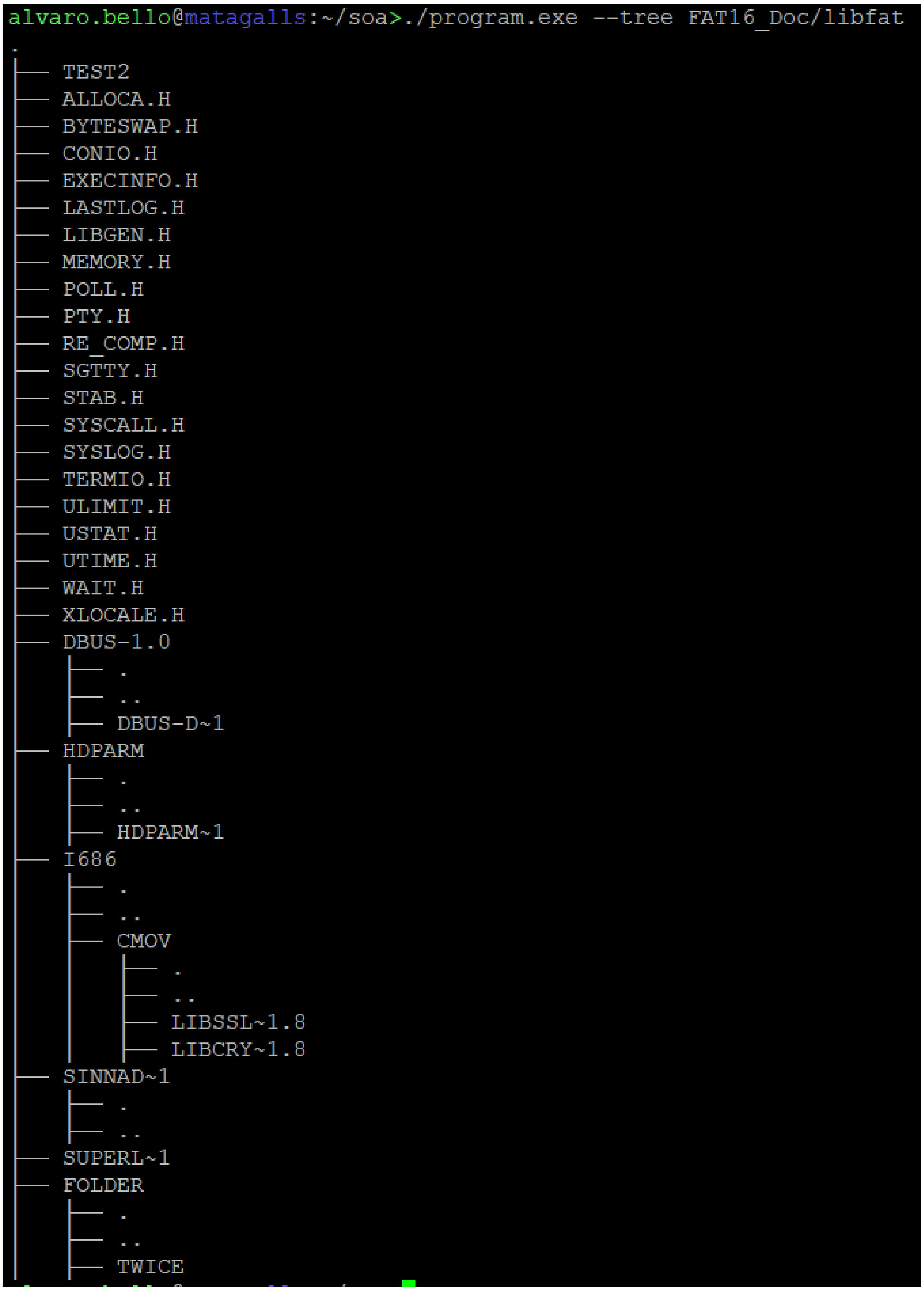
FAT16

DEVINfat:



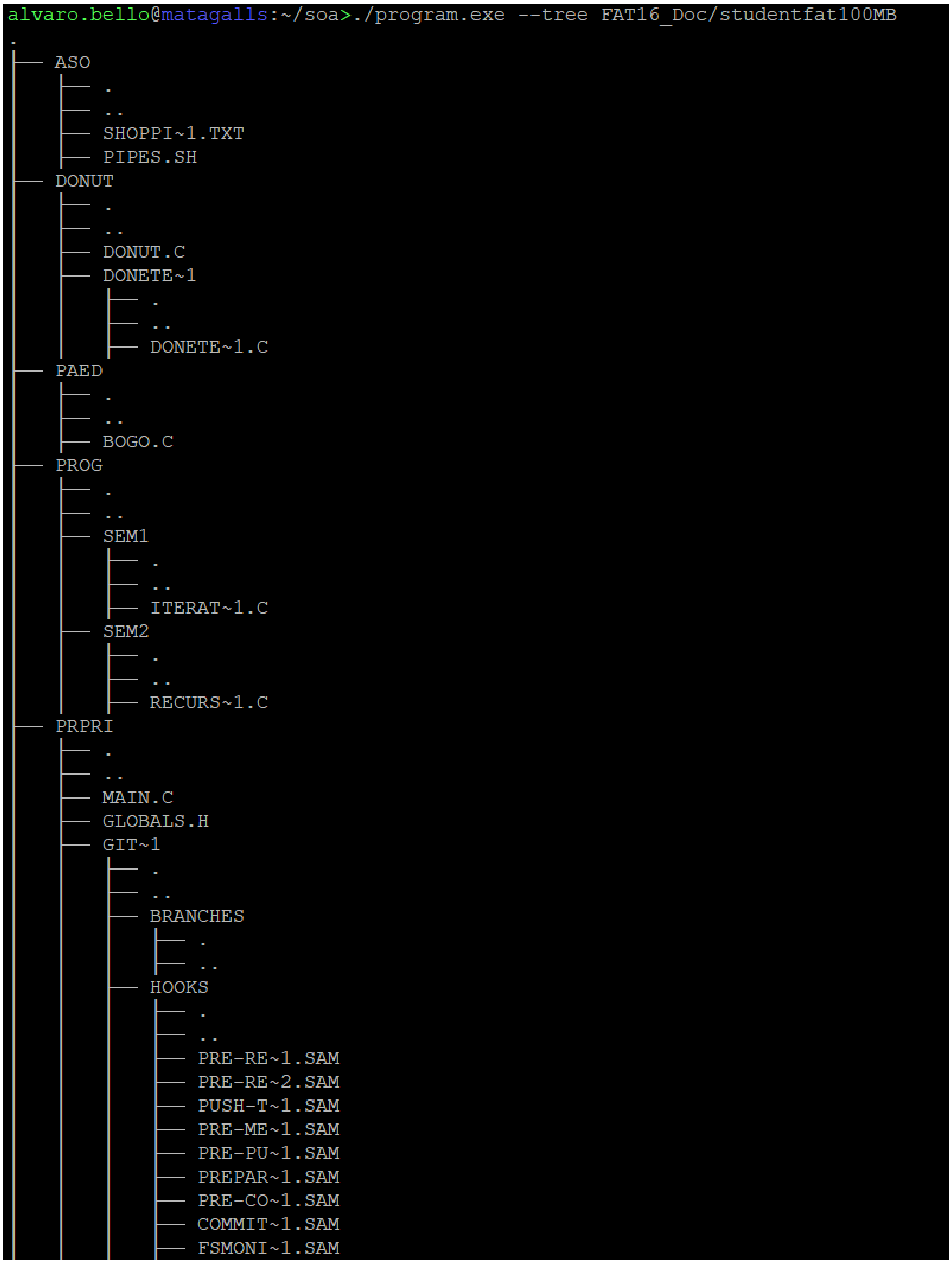
```
alvaro.bello@matagalls:~/soa>./program.exe --tree FAT16_Doc/DEVINfat
.
├── COMPUS
│   ├── .
│   ├── ..
│   ├── P1
│   │   ├── .
│   │   ├── ..
│   ├── P2
│   │   ├── .
│   │   ├── ..
│   ├── P3
│   │   ├── .
│   │   ├── ..
│   │   ├── STATEM~1.TXT
│   ├── EXTRA
│   │   ├── .
│   │   ├── ..
│   │   ├── TODOLIST.TXT
├── AOS
│   ├── .
│   ├── ..
├── IX
│   ├── .
│   ├── ..
```

libfat:

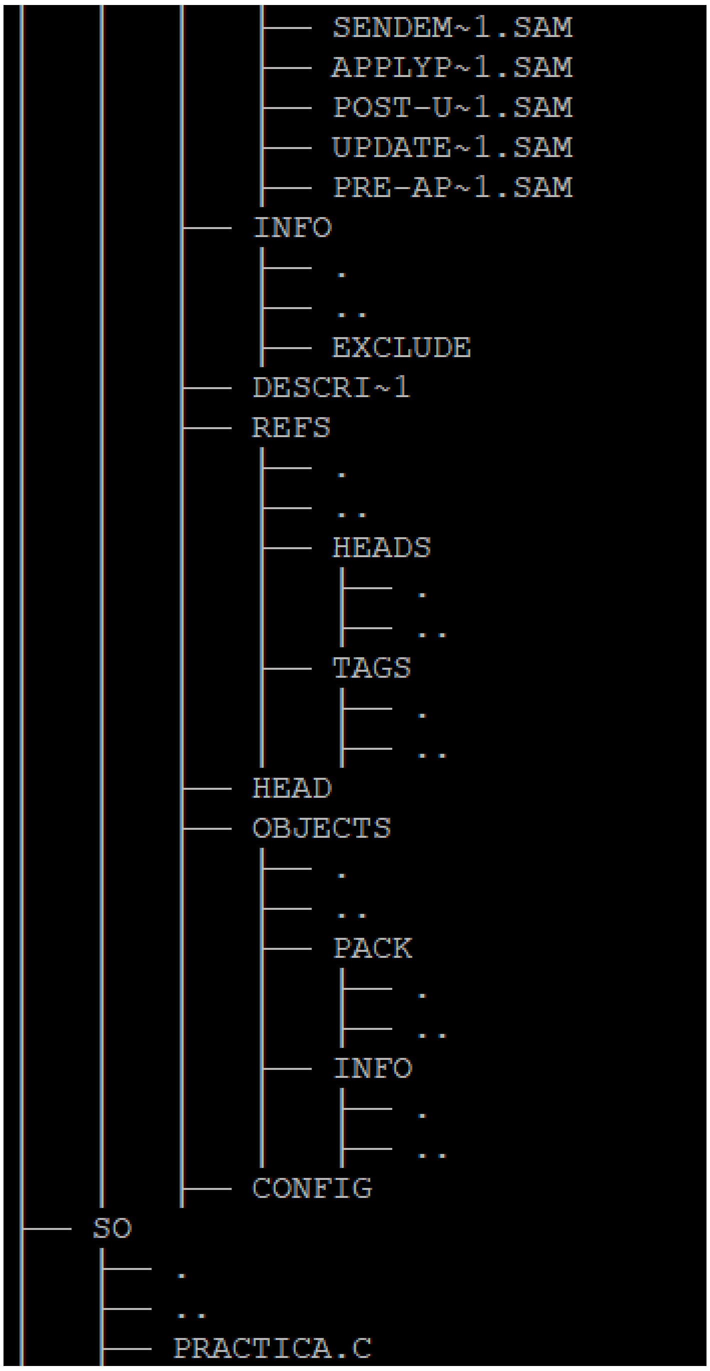


```
alvaro.bello@matagalls:~/soa>./program.exe --tree FAT16_Doc/libfat
.
├── TEST2
├── ALLOCA.H
├── BYTESWAP.H
├── CONIO.H
├── EXECINFO.H
├── LASTLOG.H
├── LIBGEN.H
├── MEMORY.H
├── POLL.H
├── PTY.H
├── RE_COMP.H
├── SGTTY.H
├── STAB.H
├── SYSCALL.H
├── SYSLOG.H
├── TERMIO.H
├── ULIMIT.H
├── USTAT.H
├── 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
```

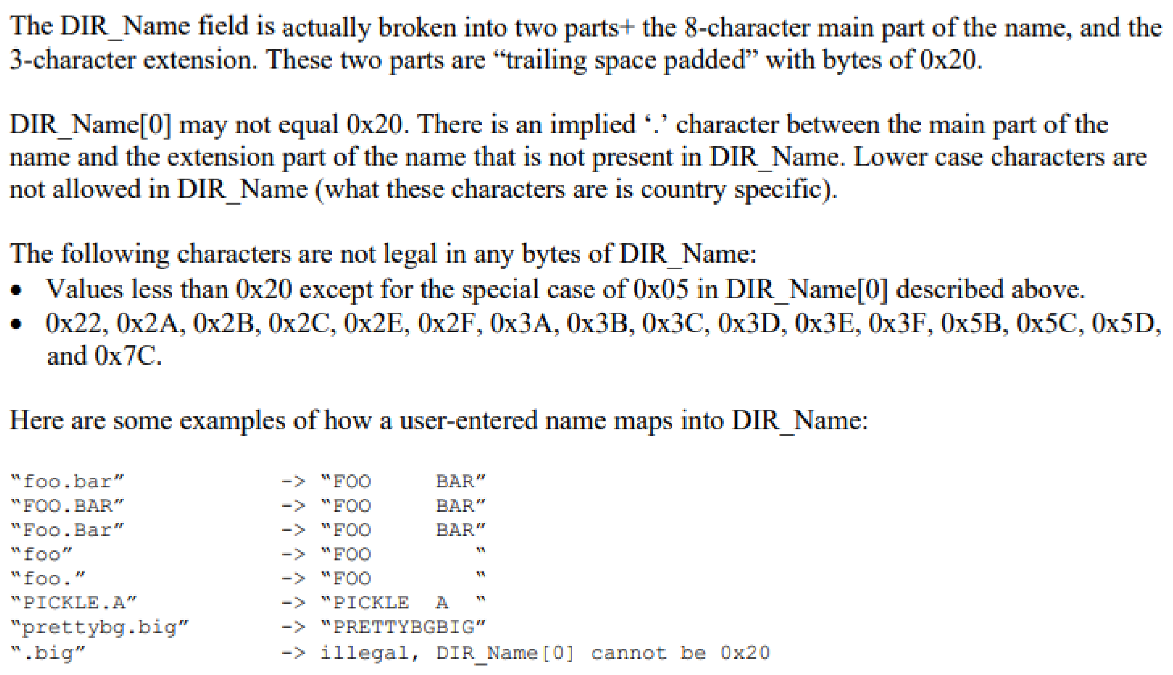


```
│   │   │   ├── SENDEM~1.SAM
│   │   │   ├── APPLYP~1.SAM
│   │   │   ├── POST-U~1.SAM
│   │   │   ├── UPDATE~1.SAM
│   │   │   ├── PRE-AP~1.SAM
│   │   ├── INFO
│   │   │   ├── .
│   │   │   ├── ..
│   │   │   ├── EXCLUDE
│   │   ├── DESCRI~1
│   │   ├── REFS
│   │   │   ├── .
│   │   │   ├── ..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

Uppercase

When running the program, the system's files are displayed correctly but all in uppercase. As noted earlier in section 1.2 Display directory tree, this is because, as indicated in the "FAT 32-Byte Directory Entry Structure -> DIR_Name" section of the document, entries only accept uppercase characters; therefore, if attempting to create a file with lowercase characters in a FAT16 file system, they will be converted to uppercase.

When running the program, the system's files are displayed correctly but all in uppercase. As noted earlier in section 1.2 Display directory tree, this is because, as indicated in the "FAT 32-Byte Directory Entry Structure -> DIR_Name" section of the document, entries only accept uppercase characters; therefore, if attempting to create a file with lowercase characters in a FAT16 file system, they will be converted to uppercase.



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:

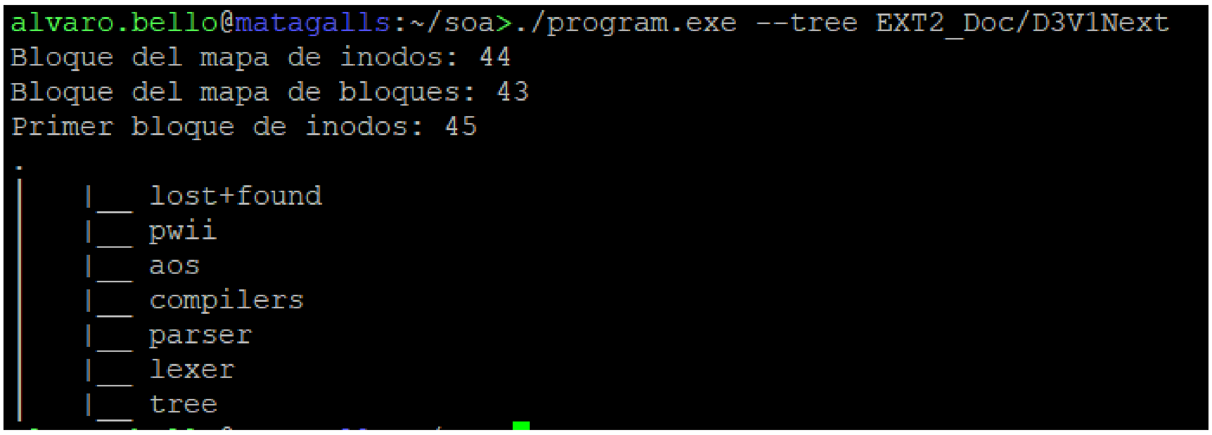
- Values less than 0x20 except for the special case of 0x05 in DIR_Name[0] described above.
- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C.

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
```

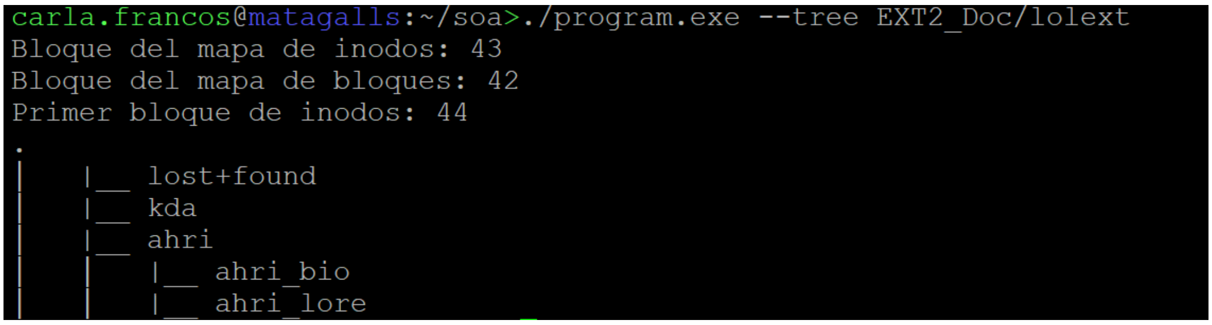
EXT 2

D3VlNext:



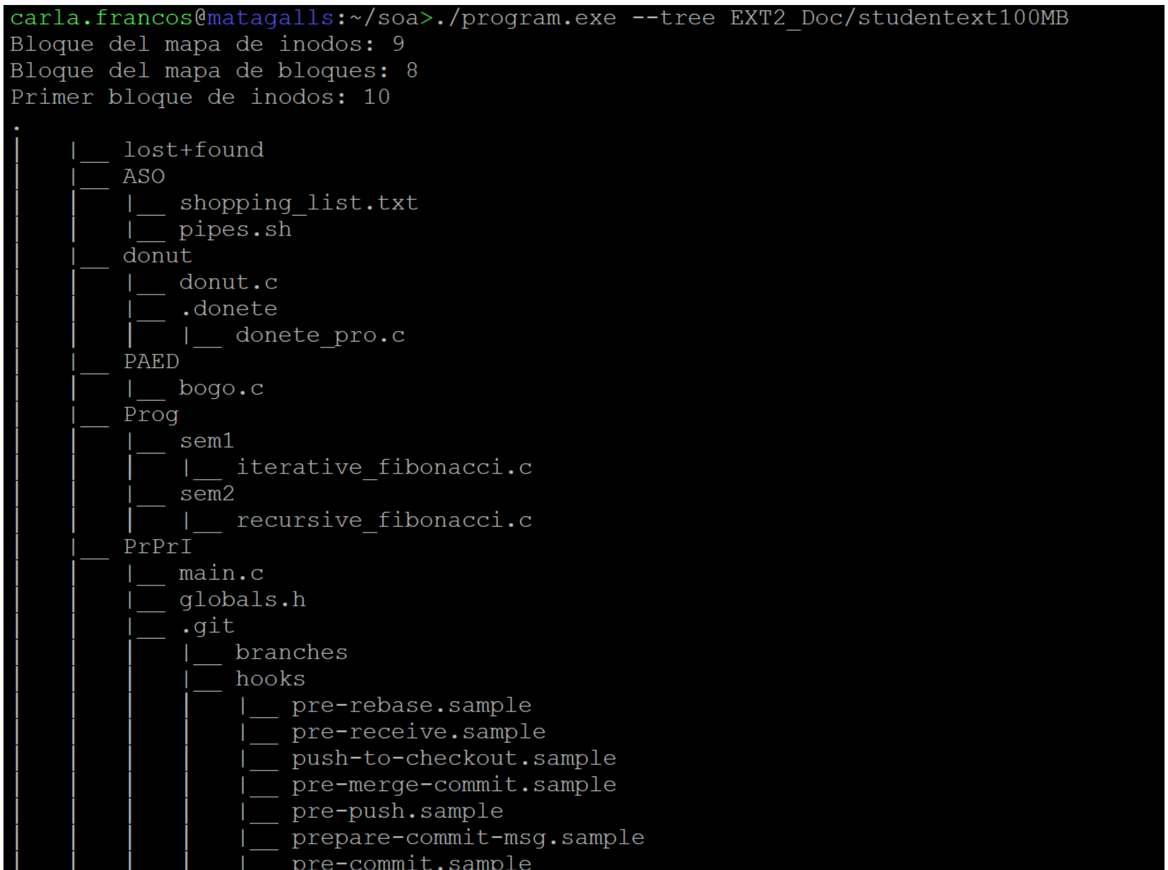
```
alvaro.bello@matagalls:~/soa>./program.exe --tree EXT2_Doc/D3V1Next
Bloque del mapa de inodos: 44
Bloque del mapa de bloques: 43
Primer bloque de inodos: 45
.
|    |__ lost+found
|    |__ pwii
|    |__ aos
|    |__ compilers
|    |__ parser
|    |__ lexer
|    |__ tree
```

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:

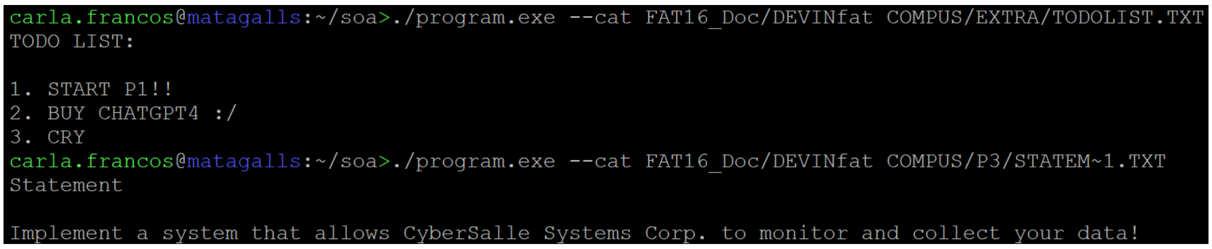


```
carla.francos@matagalls:~/soa>./program.exe --tree EXT2_Doc/studentext100MB
Bloque del mapa de inodos: 9
Bloque del mapa de bloques: 8
Primer bloque de inodos: 10
.
|    |__ lost+found
|    |__ ASO
|    |    |__ shopping_list.txt
|    |    |__ pipes.sh
|    |__ donut
|    |    |__ donut.c
|    |    |__ .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
```

3.3 Display file contents:

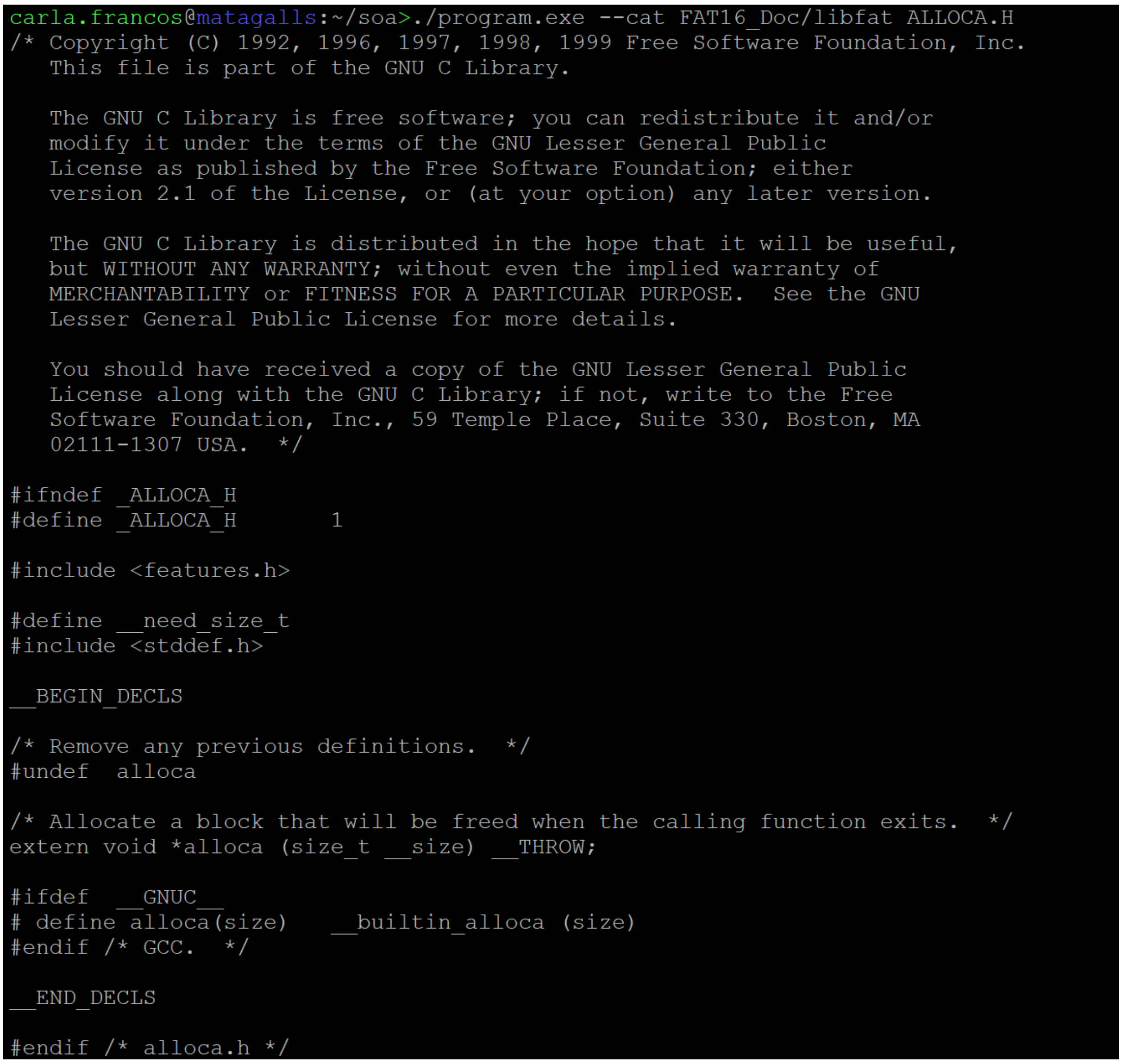
FAT16

DEVINfat:

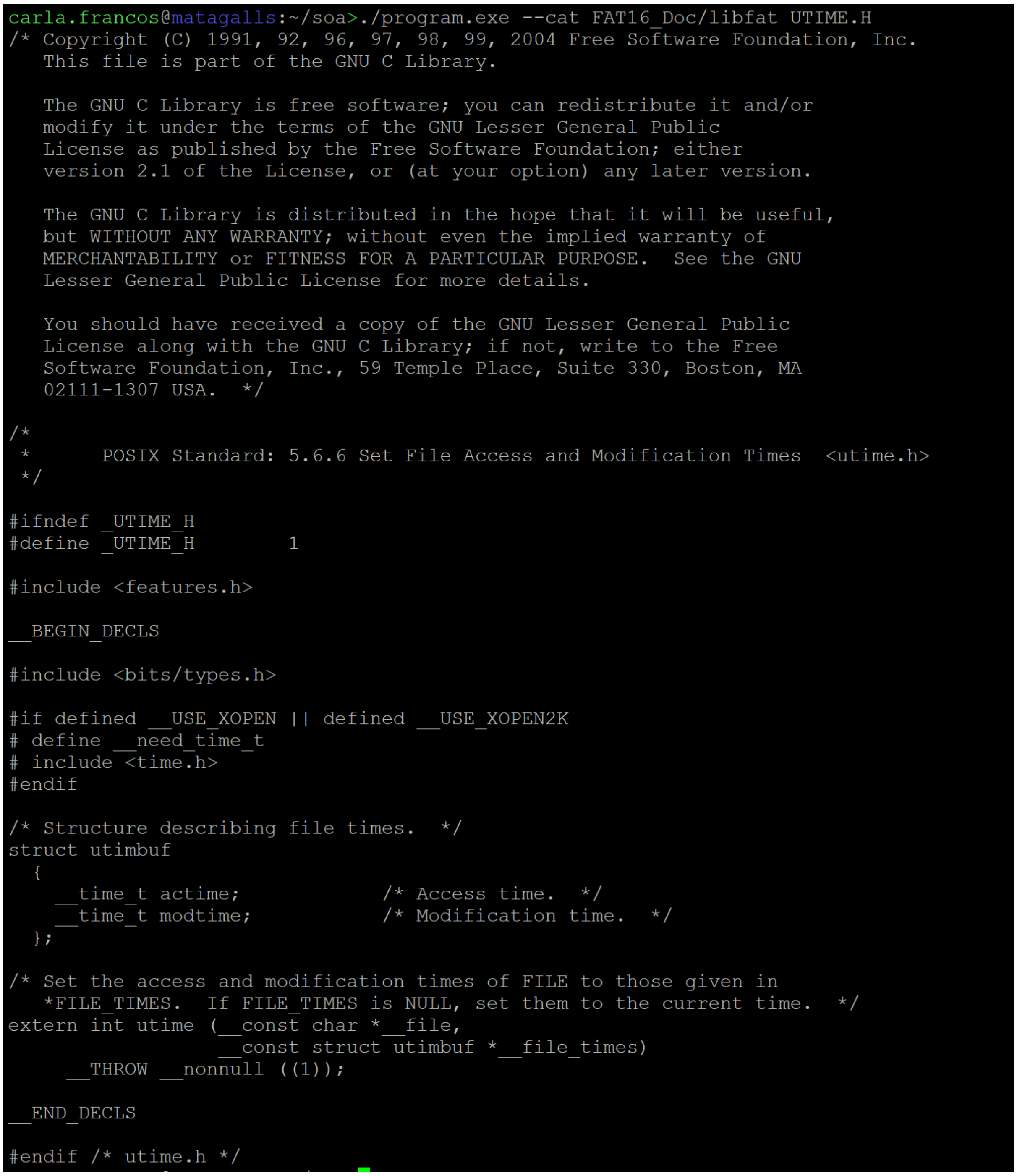


```
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/DEVINfat COMPUS/EXTRA/TODOLIST.TXT
TODO LIST:

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:

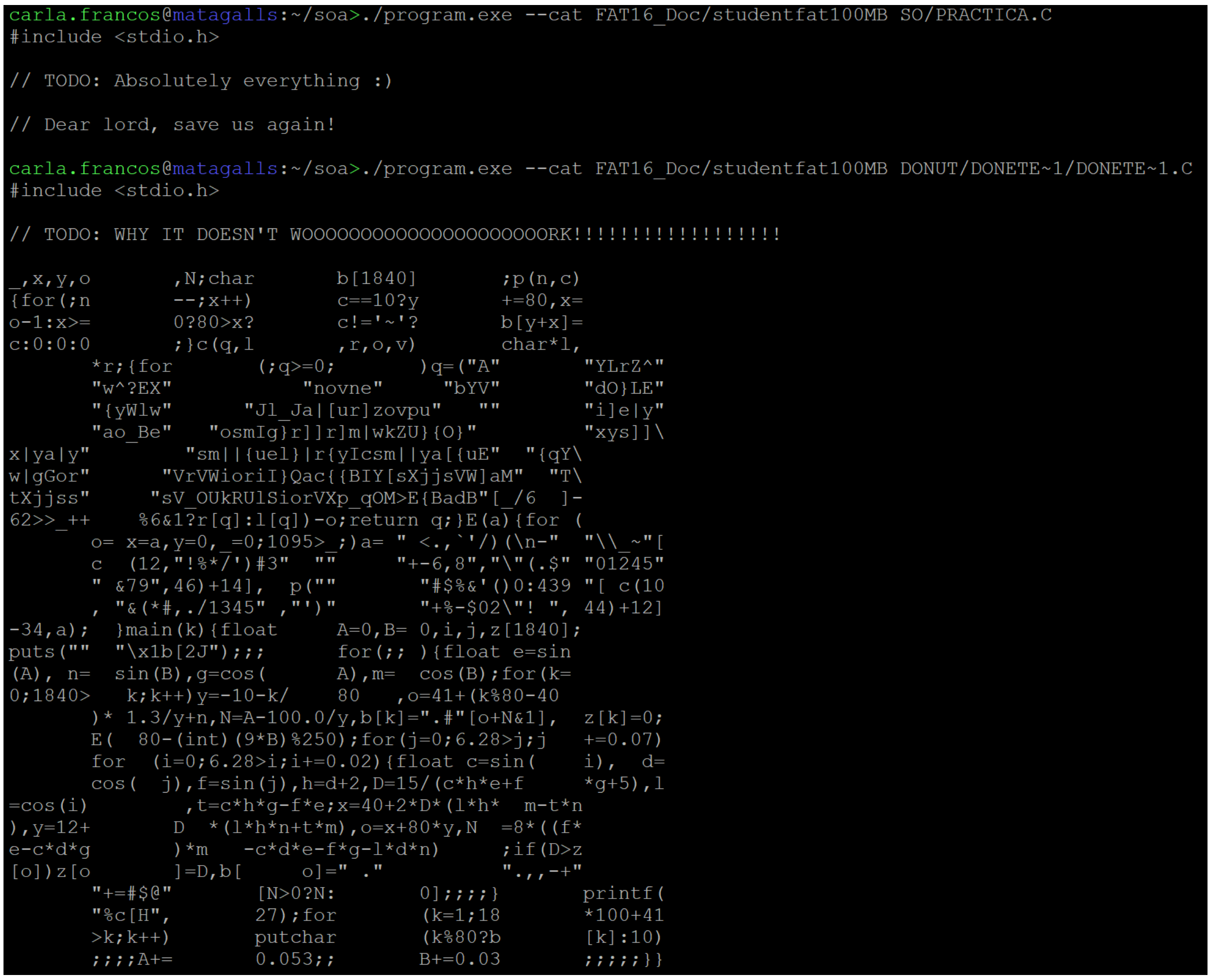


```
carla.francos@matagalls:~/soa>./program.exe --cat FAT16_Doc/libfat ALLOCA.H
/* Copyright (C) 1992, 1996, 1997, 1998, 1999 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   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 WOOOOOOOOOOOOOOOOOOOOORK!!!!!!!!!!!!!!!!!

_,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. Estimated time dedicated to each project phase

Research (reading and understanding the documentation)	10 hours
Identification and collection of file system information (FAT16)	4 hours
Identification and collection of file system information (EXT2)	2 hours
Display directory tree (FAT16)	8 hours
Display directory tree (EXT2)	20 hours
Display file contents (FAT16)	5 hours
Testing	6 hours
Report	8 hours
TOTAL	**63 hours**

In this project, extensive research was conducted on how both file systems work and how their data is organized, including the ability to read and understand two technical papers on the subject. It is also evident that more hours were required for the code related to EXT2 than for FAT16, since the EXT2 file system organizes data in a somewhat more complex way.

Once everything was operational, exhaustive testing was performed with multiple trials to ensure the project functioned correctly.

6. Conclusions

During the development of this assignment, in‑depth knowledge was gained about how two widely used file systems—FAT16 and EXT2—work and how they are internally structured. Across the different phases, various technical challenges were faced that helped develop practical skills in reading official specifications, manipulating data structures specific to each file system, and understanding advanced operating system concepts such as recursive directory‑tree traversal and the management of indirect pointers.

It was also possible to identify the difference in complexity between the two systems. While FAT16 presents a simpler structure based on chains of clusters linked through the FAT, the EXT2 system incorporates more sophisticated and complex mechanisms such as direct, indirect, and triple‑indirect blocks, which posed an additional challenge when implementing directory‑tree traversal. The importance of correctly sizing data structures was also learned, as small offset errors can cause incorrect reads of the entire file system or incomplete traversals.

Finally, the testing and validation phase made it possible to detect and resolve issues related to file system detection and the correct interpretation of metadata, further consolidating the understanding of theoretical concepts in a real environment. As a result, a program was produced that can identify, traverse, and accurately display the directory structure of both file systems, reflecting both the effort invested and the learning achieved during this assignment.

7. Bibliography

All information was obtained from the following documents included within the project folder:

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