2024

File Systems Project

Advanced Operating Systems

Miguel-Nicolás Palau Lessa

Contents

Development

Design & Implementation

Each filesystem has its own library containing the set of public and private functions that enables operations with it. These libraries are used in the main file depending on which type is the filesystem provided by the user.

All functionalities check the type of the given file before actually performing operations. The procedures *EXT2_check* and *FAT16_check* read the filesystem and recognises if it is known. For EXT2, the *s_magic* field of the superblock is read and compared to 0xEF53 which indicates that the filesystem is indeed EXT2. FAT16 requires calculating the number of clusters, which is between 4085 and 65525 in FAT16 filesystems.

Phase 1

The first phase required retrieving the metadata of a given filesystem. After checking the filesystem type, the following is executed:

EXT2

The Superblock is fetched and stored in a struct that can hold the entire Superblock data. This is done by simply reading 204 bytes, the size of the Superblock, on the fixed offset of 1024. Then, the requested data is printed nicely to the screen by accessing the different variables in the struct.

FAT16

The BootSector is fetched and stored in a struct that can hold the entire BootSector data. This is done by simply reading 64 bytes, the size of the BootSector, starting from position 0. Then, the requested data is printed nicely to the screen by accessing the different variables in the struct.

Phase 2

The second phase had to list all files and directories in a given filesystem. After checking the filesystem type, the following is executed:

EXT2

The Superblock is fetched and stored in a struct to be able to later access necessary information to traverse the filesystem. A function that will commence the traversal is called with inode 2, which holds the entries of the root directory. This function receives an inode and will recursively traverse the inodes it finds inside, printing all found files and directories in a hierarchical format.

It works by first accessing the inode's block group descriptor to get the inode table identifier, with it and the inode id, it accesses the inode entry. To optimize the traversal, given that many blocks can be assigned to a single inode, all the inode's block ids are fetched and stored in an array, easing their sequential access later. A recursive logic is employed to allow gathering all blocks stored in an indirect manner inside *i_block*.

The traversal starts by reading the first block and processing the directory entry's details. This involves checking whether the inode id is 0, which means that the entry is unused and should be ignored, whether it is an internal directory we want to ignore such as ".", ".." and "lost+found" and printing the name in a nested format. If the entry's file type is 2 it means it is a directory, thus, the function is called again with the entry's inode id. The directory entry loop is not stopped until the end of the last block associated to the initial inode is reached.

FAT16

The BootSector is also fetched here to make required calculations for the traversal. The recursion starts by reading the Root Directory region which, as the name implies, holds the entries of the data in the filesystem's root directory. Similarly to EXT2, the entries are traversed ignoring unused ones, which have 0xE5 in position 0 of the name, internal directories and entries with special attributes such as volume name. FAT filesystems previously had a name limit and currently handle long names in a special way that is not required to handle in this project, thus, the short name is employed.

Entries indicate if they represent a directory or a file through the attribute field, if a directory is found the function is called recursively with the pointing cluster. The loop is not stopped until no more clusters are associated and pending to read according to the FAT table.

Phase 3 & 4

Phase 3 displays the entire content of a given file from a given filesystem. As the file must be searched by name, Phase 3 is built on top of Phase 2, to recursively traverse directories until the file is found.

EXT2

The same method as Phase 2 is called but with a file name to find. If a directory entry, that does not describe a directory, matching the name provided is found, its inode is fetched and returned to access the file data. The file size is then calculated and its blocks are accessed and printed until the bytes read matches the file size. From block 0 to 11, the direct block id is provided. For the following ones, block ids must be retrieved using a recursive algorithm that depends on the depth of the indirect pointing.

FAT16

The FAT16 implementation also uses the same function used by Phase 2 but with slight changes to return the directory entry whenever the file name given is found and the entry is not a directory.

Data Structures

EXT2

The data structures used by the EXT2 functionalities represent as structs the different chunks of data available. The structs declared are EXTNest, Superblock, GroupDescriptor, Inode and EXTDirectoryEntry. All are EXT2 data blocks except EXTNest, which represents a current hierarchy nesting state, with the nesting count and a Boolean array indicating if the nested data is the last one in their nest.

```
typedef struct {
    int count;
    int* is_last;
} EXTNest;

typedef struct {
    uint32_t s_inodes_count;
    uint32_t s_blocks_count;
    uint32_t s_r_blocks_count;
    uint32_t s_free_blocks_count;
    uint32_t s_free_inodes_count;
    uint32_t s_first_data_block;
    uint32_t s_log_block_size;
    int32_t  s_log_frag_size;
    uint32_t s_blocks_per_group;
    uint32_t s_frags_per_group;
    uint32_t s_inodes_per_group;
    uint32_t s_mtime;
    uint32_t s_wtime;
    uint16_t s_mnt_count;
    uint16_t s_max_mnt_count;
```

```
    uint16_t s_magic;
    uint16_t s_state;
    uint16_t s_errors;
    uint16_t s_minor_rev_level;
    uint32_t s_lastcheck;
    uint32_t s_checkinterval;
    uint32_t s_creator_os;
    uint32_t s_rev_level;
    uint16_t s_def_resuid;
    uint16_t s_def_resgid;
    uint32_t s_first_ino;
    uint16_t s_inode_size;
    uint16_t s_block_group_nr;
    uint32_t s_feature_compat;
    uint32_t s_feature_incompat;
    uint32_t s_feature_ro_compat;
    uint8_t  s_uuid[16];
    char     s_volume_name[16];
    unsigned char  s_last_mounted[64];
    uint32_t s_algo_bitmap;
} Superblock;

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;
    char bg_reserved[12];
} GroupDescriptor;

typedef struct {
    uint16_t i_mode;
    uint16_t i_uid;
    uint32_t i_size;
    uint32_t i_atime;
    uint32_t i_ctime;
    uint32_t i_mtime;
    uint32_t i_dtime;
    uint16_t i_gid;
    uint16_t i_links_count;
    uint32_t i_blocks;
    uint32_t i_flags;
    uint32_t i_osd1;
    uint32_t i_block[15];
    uint32_t i_generation;
```

```
    uint32_t i_file_acl;
    uint32_t i_dir_acl;
    uint32_t i_faddr;
    unsigned char i_osd2[12];
} Inode;

typedef struct {
  uint32_t inode;
  uint16_t rec_len;
  uint8_t name_len;
  uint8_t file_type;
} EXTDirectoryEntry;
```

FAT16

Similarly to EXT2, the data structures used hold data chunks of the FAT16 filesystem, apart from FATNest. The structs are FATNest, BootSector and FATDirectoryEntry.

```
typedef struct {
    int count;
    int* is_last;
} FATNest;

typedef struct {
    uint8_t BS_jmpBoot[3];
    uint8_t BS_OEMName[8];
    uint16_t BPB_BytsPerSec;
    uint8_t BPB_SecPerClus;
    uint16_t BPB_RsvdSecCnt;
    uint8_t BPB_NumFATs;
    uint16_t BPB_RootEntCnt;
    uint16_t BPB_TotSec16;
    uint8_t BPB_Media;
    uint16_t BPB_FATSz16;
    uint16_t BPB_SecPerTrk;
    uint16_t BPB_NumHeads;
    uint32_t BPB_HiddSec;
    uint32_t BPB_TotSec32;
    uint8_t BS_DrvNum;
    uint8_t BS_Reserved1;
    uint8_t BS_BootSig;
    uint32_t BS_VolID;
    uint8_t BS_VolLab[11];
    uint8_t BS_FilSysType[8];
} BootSector;
```

```c
typedef struct {
    uint8_t DIR_Name[11];
    uint8_t DIR_Attr;
    uint8_t DIR_NTRes;
    uint8_t DIR_CrtTimeTenth;
    uint16_t DIR_CrtTime;
    uint16_t DIR_CrtDate;
    uint16_t DIR_LstAccDate;
    uint16_t DIR_FstClusHI;
    uint16_t DIR_WrtTime;
    uint16_t DIR_WrtDate;
    uint16_t DIR_FstClusLO;
    uint32_t DIR_FileSize;
} FATDirectoryEntry;
```

Testing

During the development of the phases, there has been significant testing to ensure functionalities work as expected, even with edge or uncommon cases. This is the outcome of some testing done:

```
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --info 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 frags: 8192

INFO VOLUME
  Volume Name:
  Last Checked: Mon Apr 10 14:32:05 2023
  Last Mounted: Mon Apr 10 14:57:52 2023
  Last Written: Mon Apr 10 14:58:42 2023
```

```
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --tree libfat
├ 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
├ hdparm
│       └ hdparm
├ i686
│       └ cmov
│               ├ libssl
│               └ libcry
├ sinnad
├ superl
└ folder
        └ twice
```

```
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --cat libfat hdparm
hdparm_is_on_battery() {
    on_ac_power 2>/dev/null
    [ $? -eq 1 ]
}

hdparm_set_option()
{
  local NEW_OPT= NEW_DEF=
  if test -n "$DISC"; then
    for i in $OPTIONS; do
      if test x${i%${i#??}} != x${1%${1#??}}; then
        NEW_OPT="$NEW_OPT $i"
      else
        NEW_OPT=${NEW_OPT%-q}
      fi
    done
    OPTIONS="$NEW_OPT $OPT_QUIET $1"
  else
    for i in $DEFAULT; do
      if test x${i%${i#??}} != x${1%${1#??}}; then
        NEW_DEF="$NEW_DEF $i"
      else
        NEW_DEF=${NEW_DEF%-q}
      fi
    done
    DEFAULT="$NEW_DEF $DEF_QUIET $1"
  fi
}

hdparm_eval_value()
{
  case $1 in
    off|0)
      hdparm set option "$2"0
```

```
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --tree yakofs
├ see
├ lschads
│       ├ pepefolder
│       │       └ pixels
│       ├ marcusfolder
│       ├ arexufolder
│       │       └ java
│       ├ gerifolder
│       └ yubfolder
│               └ jav
├ offsetfolder
│       └ la_vie_en_rose
├ file1
└ osffetfolder
        └ panorama
```

```
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --cat fat1.fs file15
file 15 data
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --cat fat1.fs file16
ERROR: File not found.
```

```
miguelnicolas.palau@matagalls:~/Year3/SOA/src>./fsutils --cat yakofs jav
https://www.youtube.com/watch?v=b-Cr0EWwaTk
```

Time dedication

Phase 2 has been the most time-consuming phase as it is much more complex than Phase 1 and the base of the next phases, thus, a solid implementation was required which involved more time.

The time dedicated to Phase 1 was a single hour, as it was quite straightforward. Phase 2 took approximately a total of 1-2 days. Phase 3 was quite simple on top of Phase 2, it took roughly 2 hours. Phase 4, on the other hand, was a bit more complicated because of the indirect placement of the blocks, requiring a recursive algorithm to access the file data. It took approximately 3-4 hours.

Encountered problems

Some problems encountered during the development of the functionalities include:

- Printing incorrectly the preceding symbol in the tree functionality
  - Solution: For each entry, check if there are more entries. In EXT2, looking for used entries until last block is read. In FAT16, looking for used entries until there are no more related clusters.

- Printing internal directories/files
  - Solution: Ignore ".", "..", "lost+found" and attribute VOLUME_ID.

- Traversing symbolic links allowing infinite loops
  - Solution: Avoid treating symbolic links as directories.

- Memory leaks
  - Solution: Use valgrind to check and resolve any memory leaks.

- Incomplete directories in the tree functionality
  - Solution: Reading the FAT table until there are no more associated clusters.

- Incomplete directories in the tree functionality
  - Solution: Removing the loop stop when an inode id with value 0 is found.

- Incomplete directories in the tree functionality
  - Solution: Reading blocks until total number of blocks is reached, instead of assuming directory entries only consume the first block.

- Printing data for directories
  - Solution: Ignoring directories when comparing file name.

Conclusions

In conclusion, the project provided valuable insights into the internal structure and operation of the EXT2 and FAT16 file systems. EXT2, with its robust structure and efficient file indexing, is ideal for systems requiring high performance and large storage capacities, although its lack of journaling can be a drawback for data recovery. On the other hand, FAT16, despite its limitations in storage capacity and file size, it is quite simple and has broad compatibility, making it suitable for legacy systems and removable media.

The hands-on experience allowed me to differentiate both filesystems acknowledging the importance of choosing the right file system based on specific use cases. It has definitely been useful to expand my knowledge on file systems, as I didn't know how filesystems were structured internally and how operating systems accesses and stores vast amount of data in no time.