University of Cergy-Pontoise

REPORT

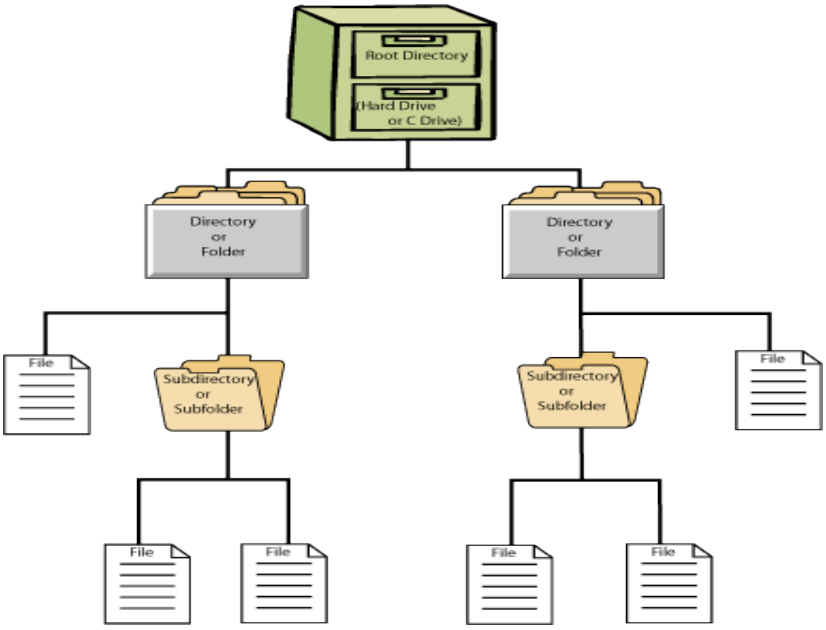
for the project of operating system
Bachelor of Computer Science third year

on the subject

file system manager

written by

AYAD Ishak and ABDELMOUMENE Djahid



March 2019

Table des matières

1 Introduction

In this project, we created a simplified version of the UNIX extended file system. To be mounted on an another UNIX file which was used as a virtual partition.

Our version of the file system followed a lot of the EXT standards with exception of the usage of block groups and the block group descriptors which were not used. The main file system API contains UNIX system call style functions such as open and write etc. However we did add an underscore at the end of all of these functions to avoid conflicts with the actual system.

2 Architecture

For the implementation of the file system we chose to divide different parts of the FS (file system) into five levels of abstraction :

The first one (in disk.c) separating the interaction with the UNIX file -used as a partition- from the rest of the FS. And for that a structure was created to facilitate the transportation of some useful information about the file to the rest of of the FS. This structure serves as the virtual partition and is called **fs_filesyst** and it holds the file descriptor to the partition file and the size. Which is used to simulate how a real partition has an immutable size. And lastly the number of blocks which can also be deduced from the size, these blocks have a fixed size of 4096 (FS_BLOCK_SIZE).

```
struct fs_filesyst{
  uint32_t fd;       /**< file descriptor */
  uint32_t tot_size;/**< total size of our file (partition) */
  uint32_t nblocks; /**< number of blocks in disk image*/
};
```

Listing 1 – virtual filesystem structure

The second level of abstraction handles the formatting of the main parts of the FS. such as the superblock which is always in the first block and holds information about the FS in general as well as the offsets and sizes of other sections of the FS. the structure holding the superblock is as follows :

```
struct fs_super_block  {
  uint32_t magic;         /**< the filesystem magic number */

  uint32_t data_bitmap_loc;  /**< data bitmap location in block num */
  uint32_t data_bitmap_size; /**< data bitmap size in blocks */
  uint32_t inode_bitmap_loc; /**< inode bitmap location in block num */
  uint32_t inode_bitmap_size;/**< inode bitmap size in blocks */

  uint32_t inode_loc;       /**< location of inodes in block num */
  uint32_t inode_count;       /**< no of inodes in blocks */
  uint32_t data_loc;        /**< location of the data in blocks */
  uint32_t data_count;      /**< no of data blocks */

  uint32_t free_inode_count; /**< no of free inodes */
  uint32_t free_data_count;  /**< no of free blocks */

  uint32_t nreads;          /**< number of reads performed */
  uint32_t nwrites;       /**< number of writes performed*/
  uint32_t mounts;          /**< number of mounts*/

  uint32_t mtime;         /**< time of mount of the filesystem */
  uint32_t wtime;         /**< last write time */
};
```

Listing 2 – super block structure

To get the offset and sizes of the other sections, we set the ratio of the inode table as constant (1% of the total blocks), from this we can also get the bitmap size and we can leave the rest to the data blocks and its bitmap. From all of this we can get the offsets of all the sections. So at the end our file system will look like this :

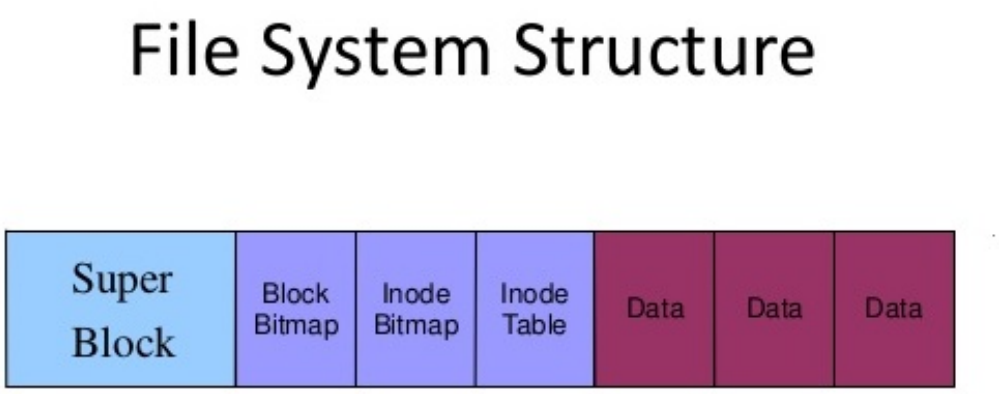


FIGURE 1 – Simplified filesystem structure

in this level we also have functions to allocate, free, read and write data and inode blocks. For the inodes this structure was used :

```
struct fs_inode {
  uint16_t mode;                  /**< file type and permissions */
  uint16_t uid;                   /**< id of owner */
  uint16_t gid;                   /**< group id of owners */
  uint32_t atime;                 /**< last access time in seconds since the epoch */
  uint32_t mtime;                 /**< last modification time in seconds since the epoch*/
  uint32_t size;                  /**< size of the file in bytes */
  uint32_t hcount;                /**< the number of hard links pointing to this inode*/
  uint32_t direct[FS_DIRECT_POINTERS_PER_INODE];/**< direct data blocks */
  uint32_t indirect;                /**< indirect data blocks */
};
```

Listing 3 – inode structure

And there is also the structure that holds the different representations of a data blocks :

```
union fs_block {
  struct fs_super_block super;        /**< super block */
  struct fs_inode inodes[FS_INODES_PER_BLOCK];/**< array of inodes */
  uint32_t pointers[FS_POINTERS_PER_BLOCK];   /**< array of pointers */
  uint8_t data[FS_BLOCK_SIZE];        /**< array of data bytes */
};
```

Listing 4 – union of a block structure

Note that a single block can hold up to 64 inodes and 1024 pointers to other blocks (used in indirect blocks).

For the third level of abstraction we no longer need to worry about the allocation of blocks and formatting, so it handles the actual input/output operations on the inodes. as well as the allocation of file descriptors. For the latter we used the following structures :

```
struct io_filedesc {
  int is_allocated;
  uint32_t offset;
  uint32_t mode;
  uint32_t inodenum;
};

struct io_filedesc_table {
  struct io_filedesc fds[IO_MAX_FILEDESC];
};
```

Listing 5 – files descriptors structure

The file descriptors store the information needed to do the I/O operations such as the inode numbers as well as their current offsets -the offset from the start to write/read to/from-

For the fourth level of abstraction, we handle the directories and operations on them. And for that we used the same approach as UNIX, that is with directory entries, meaning that directories themselves are files containing 'entries' of inode numbers and filename pairs. For this we used these structures :

```
struct dirent {
    uint32_t d_ino;         /** the inode number */
    int d_type;                         /* File type  */
    char d_name[256];                   /* File name  */
};
```

Listing 6 – dirent structure

```
typedef struct {
  int fd;               /** the directory file descripto r*/
  int size;             /** the number of the directory entries stored */
  int idx;              /** the current index in the reading (used in readdir) */
  struct dirent* files; /** the directory entries  */
} DIR_;
```

Listing 7 – dir structure

the first one is a single directory entry containing a fixed size filename of 256 and the type of the file, and the inode number. As for the second structure, it holds information about the directory as a file (its file descriptor etc).

For the fifth and last level of abstractions, we used the previous two level to construct the main API's functions which are the following :

```
int initfs(char* filename, size_t size, int format);  // to initialize the file system
int open_(const char* filename, int creat, uint16_t perms); // open a file
int close_(int fd); // close a file
int write_(int fd, void* data, int size); // write data to an fd
int read_(int fd, void* data, int size);  // read data from an fd
int lseek_(int fd, uint32_t newoff); // change the current offset
int rm_(const char* filename); // remove a file
DIR_* opendir_(const char* dirname, int closedir_(DIR_* dir); // close a directory
int creat, uint16_t perms); // open a directory
struct dirent readdir_(DIR_* dir); // read an entry from the directory
int rmdir_(const char* filename, int recursive); // remove a directory
int ln_(const char* src, const char* dest); // create a hard link
int cp_(const char* src, const char* dest); // copy a file
int mv_(const char* src, const char* dest); // move or rename
void closefs(); // close the file system
```

Listing 8 – main API's functions

3 Implementation

As for the implementation the main algorithms are those of the data blocks and inode allocations, and the reading and writing to files as well as the handling of the directory entries.

3.1 block allocation

for the allocation of data blocks and inodes, the algorithm simply parses the respective bitmaps of the wanted block type, and finds the first null bit, which meant the use of bit wise operations because C doesn't allow for the direct access of bits. after this is done the bit is set to 1 and then the necessary calculations are done to get the offsets.

3.2 I/O lazy allocation

When writing to an inode (file) we have direct and indirect blocks to write to, which are stored in the in the inode itself. When implementing this we optimised the writing so that it takes less space, that is by implementing a lazy allocation system. What it does is that it delays the allocation of a block until the very last moment it's needed (when the file is written to), meaning that if we don't write to a block it won't get allocated, even if the block is between two other allocated blocks.

An example of when this optimisation saves space is when a user writes a byte of data at the start of the file and then changes the offset to 100000 and then writes another byte of data. The idea is that only the first and last blocks are allocated. whereas the others are always not allocated.

To achieve this we have a function that figures out the blocks that need to be allocated when writing - called **io_lazy_allocation**- which takes the the start offset and the size to allocated and calculates the blocks that need to be allocated and then allocates them.

3.3 Directory entries

To store the directory entries, we used a sorted list to store the entries. the sorting is done in order of the file name, to insert a file we simply put it at the end of the directory and then we 'bubble' it into its sorted place. for the search we used a standard binary search. and for the deletion we just find the entry and then delete it, and then adjust the entries that follow.