VMH FILESYSTEM

Basic OS project
Fall 2022
First year engineering program

BY

Villon Chen
Moaad Maaroufi
Hamza Jad Al Aoun

SUPERVISORS

Ludovic Apvrille
Sophie Coudert

EURECOM Engineering School
Sophia Antipolis, France

Thursday 1st December, 2022

Contents

I Introduction

Usually, when a file system is full, the user gets a "no space left on device" error message: it is not possible to add a new file or extend a file.

The objective of the project is to program a new file system that can handle a "no space left on device" error: this new file system we are going to implement assumes that, when the file system is full, the file system deletes the oldest files until it can store the new file.

The file system to program is simply an array of data stored as a regular file on the computer. The implementation of files, directories, inodes... and the intuition behind it are going to be discussed thoroughly in the first section.

Six main functions has been implemented in the file system (create, write, read, remove, ls, size) with the objective of reserving as much space as possible for file data. The structure of each one of these functions will be explained in the following sections.

II File system structure

As we explained in the introduction, the main characteristic of this file system is the FIFO principle (first in first out) in writing files. So instead of using standard static data blocks, why not use a dynamic queue (FIFO) container which will model perfectly our file system. The cherry on top is that the files in our file system will be always sorted by date (the oldest element is always going to be the first element in the array).

In this manner, each time we want to modify the file system we have to put on the RAM, make the necessary modifications and then save it back to the disk. In our case, the size of the file system is just 10 MB, so its not going to affect the RAM. With larger file Systems this approach is obsolete.

To do so, we first use a function *get_FS* that reads a file system stored on the disk to the memory. After the necessary changes are made to the file system, we use the function *put_FS* that rewrites the file system back into the disk.

FILESYSTEM
SuperBlock sb
File * file_array
Directory * directory_array

Table II.1: FileSystem structure

In order to facilitate the manipulation of the objects that we are going to use in the file system we chose for our file system to be a *struct* that contains a **superblock**, a **directory array** and a **file array**:

- **The SuperBlock**: which is in itself a *struct* that contains some useful information about the file system(number of files, number of directories, size of the directory array, the current size of the file system, and the maximum size of the file system which is in our case 10 MB).

SUPERBLOCK
long int file_number
long int directory_number
long int directory_array_size
long int current_size
long int max_size

Table II.2: SuperBlock structure

- **The directory array**: which is an array of structs. The directories are going to be modeled by structs that contains its name and the index of its parent directory in the directory array. We chose this implementation so we can link directories to one another easily. The root is going to have index zero by convention in its parent id. When we need to remove a particular directory we will just put -1 in its parent id instead of removing it from the directory array(this will mess up the parent child relationship because it relies on the index of each directory in the array and thus changing it will cause problems). In all other implementations, this will taken into account, so by convention parent id = -1 means this directory doesn't exist and we overwrite it with brand new directory.

DIRECTORY
char name*[WORD_SIZE]*
long int parent_id

Table II.3: Directory structure

- **File array**: which is an array of structs as well, these structs are going to be of type File which is in itself a struct that contains the contents of the file *char* bytes* and another struct named *Inode* which provides some information about the file such as its name, size and its parent directory index.

FILE
Inode inode
char * bytes

Table II.4: File structure

INODE
long int size
char name*[WORD_SIZE]*
long int parent_id

Table II.5: File structure

III Commands

1 The create command

1.1 How it works

create: takes as an input the size of the file system we want to create.

When calling the create command, memory will be allocated on the heap for both the directory and the file array since they are dynamic, and then a file system structure is filled with the initial data. Eventually the new file system will have:

- max_size will be size*1000*1000 (size is passed to create in MB and its multiplied by 10^6 in order to convert it into bytes).
- the directory and file's number is initialized to 0.
- the current size of the file system will updated to the size of the Superblock plus the size of the directory and file arrays.
- using add_directory function, root directory "/" with index -2 is added to the file system after allocating memory for 1 Directory struct in the directory array.

At the end the FileSystem struct will be stored on the disk using put_FS function and all the allocated memory will be freed up.

1.2 Testing

```
$ vmhFS /tmp/tmpFS create 10
FileSystem created
Size: 80 B
Max size: 10000000 B
Files: 0
Directories: 1
```

Code block 1: *create command*

```
$ stat /tmp/tmpFS
  File: /tmp/tmpFS
  Size: 10000000        Blocks: 8          IO Block: 4096   regular file
Device: 10302h/66306d   Inode: 5898250     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/  michel)   Gid: ( 1000/  michel)
Access: 2022-12-01 14:31:22.790783924 +0100
Modify: 2022-12-01 15:13:15.805815135 +0100
Change: 2022-12-01 15:13:15.805815135 +0100
 Birth: 2022-12-01 13:17:22.874251973 +0100
```

Code block 2: *File system on the disk*

2 The ls command

2.1 How it works

ls: takes as an input the directory path and two flags: -r for the recursive version and -d to sort the files by date. (By construction of the file system, they're always sorted.)

- *ls_accumulator(FileSystem fs, long int dir_id, int level, int max_level)*: this function is recursive, it takes the directory id as an input, it uses the *get_dir_children(fs,id)* function to get all the children directories of the given directory and stores them in a *dir_children* structure, if the children number of this directory is greater than zero this means that its not empty and has children directories, in this case we will print the name of the directory children and recursively print all the files found in these directories. The base condition for this recursive function is *dir_children.number*==0 i.e to stop when a given directory doesn't n't have any children directories.

- *Dir_files get_dir_files(FileSystem fs, long int dir_id)*: this function takes as an input the directory id and find all the files found in this directory.

  The ls function always checks whether the path provided is correct or not. If the recursive flag hasn't been indicated, it will print the all files inside the desired directory.

  On the other hand if the recursive flag is present, the ls function prints the files inside the given directory and recursively prints all files contained in its children directories schemed in a tree representation.

2.2 Testing

```
$ vmhFS /tmp/tmpFS create 10
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/dir2/dir3/dir4/dir5/foo.txt
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/foo1.txt
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/foo2.txt
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/foo3.txt
$ vmhFS /tmp/tmpFS ls /dir1
$ vmhFS /tmp/tmpFS ls /dir1 -r
```

Code block 3: *ls command*

```
Write file /tmp/foo.txt to filesystem at: /dir1/dir2/dir3/dir4/dir5/foo.txt
Write file /tmp/foo.txt to filesystem at: /dir1/foo1.txt
Write file /tmp/foo.txt to filesystem at: /dir1/foo2.txt
Write file /tmp/foo.txt to filesystem at: /dir1/foo3.txt
List segment:
dir1
L__foo1.txt [18 B]
L__foo2.txt [18 B]
L__foo3.txt [18 B]
L__dir2
List segment:
dir1
L__foo1.txt [18 B]
L__foo2.txt [18 B]
L__foo3.txt [18 B]
L__dir2
    L__dir3
        L__dir4
            L__dir5
                L__foo.txt [18 B]
```

Code block 4: *Command output*

3 The read command

3.1 How it works

read: return the bytes of the file. It takes in argument the path to the file. First it checks if the path refers to a file and not a directory.

Then, we call the function *file_from_path*. If the return value equals -1, that means the file doesn't exist. Else, we'll use the function *file_id* to get the index of the file in the *file_array*.

Thanks to the File struct, we can access easily to the content of the file with the member *File.bytes*. Finally we print it in the standard output.

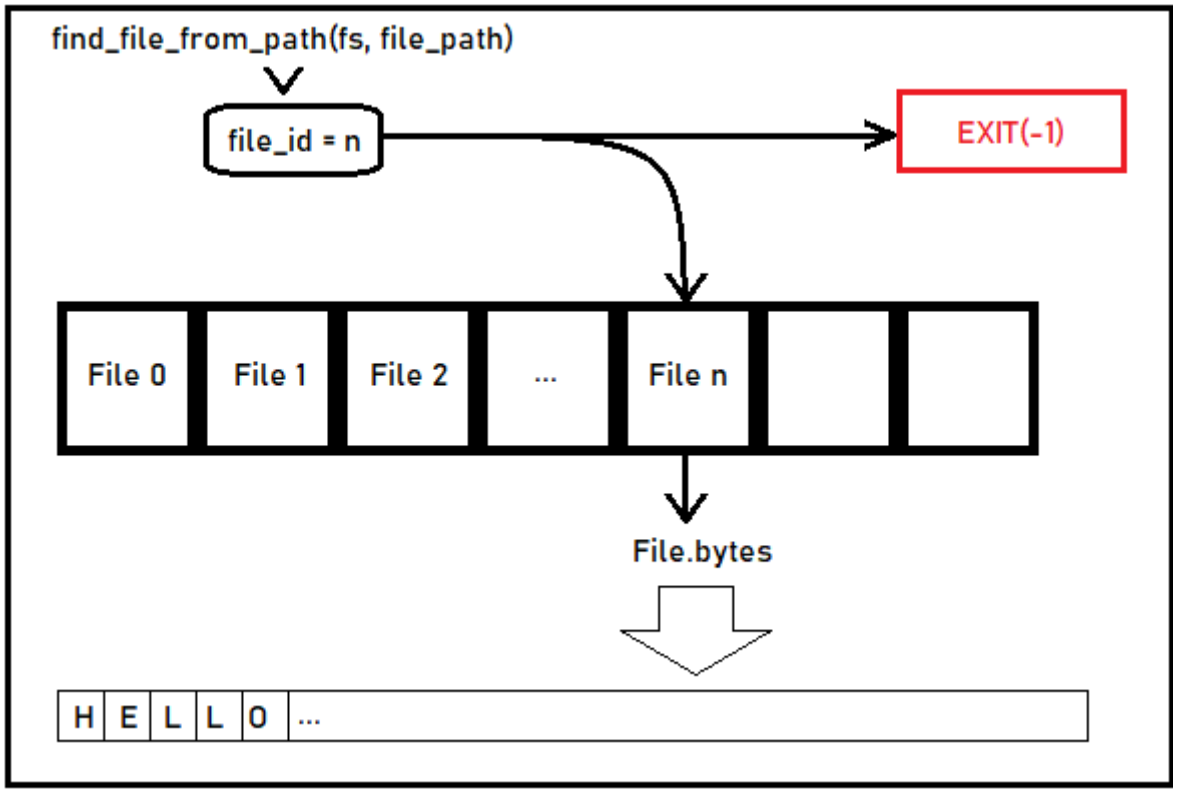

Figure 1: Reading a file (made with MS Paint)

3.2 Testing

```
$ echo "Random string 123" > /tmp/foo.txt
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/dir2/foo.txt
$ vmhFS /tmp/tmpFS read /dir1/dir2/foo.txt
$ vmhFS /tmp/tmpFS read /dir1/dir2/foo2.txt
```

Code block 5: *Read command*

```
Write file /tmp/foo.txt to filesystem at: /dir1/dir2/foo.txt
Random string 123
File doesn't exist
```

Code block 6: *Command output*

4 The remove command

4.1 How it works

remove: remove a file or directory at the provided path.

As we did in the read function we should also check here whether the path is correct or not. To do so, we can call the functions *find_file_from_path* and *find_dir_from_path*.

If the path is valid (i.e. there is something at this path in the file system), then we must get a non -1 value (-1 means the file/dir doesn't exist at this path). Else, if we get return value -1 for both of these functions, the path is not valid.

Now that we know whether the path refers to a file or a directory, we just need to call the adapted function to remove it (either *rm_directory or rm_file*). Those functions are file system functions, because they are used in many commands.

Remove a file is different from removing a directory. On one hand, remove a file consists of removing the file from the *file_array* and then shift every other file to fill the gap. On the other hand, removing a directory consists only of setting up the directory's *parent_id* to -1 (convention) and name to *TRASHED*.

Directories should not be shifted like files, because we need to preserve the parent-child relationship between directories and this information is stored inside in *parent_id* member and their indices in the *directory_array*.

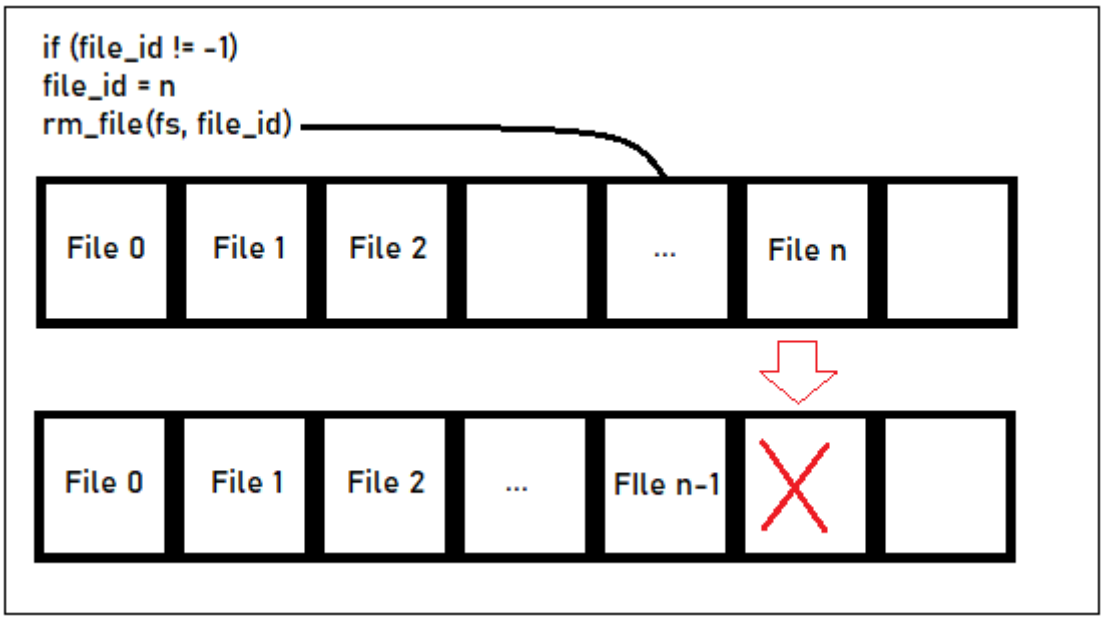

Figure 2: Removing a file (made with MS Paint)

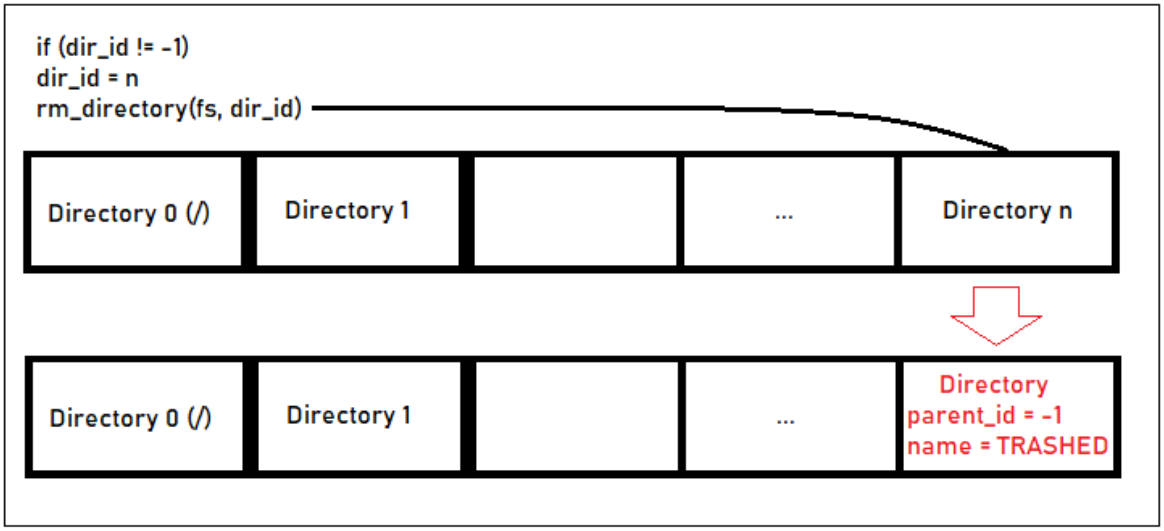

Figure 3: Removing a directory (made with MS Paint)

4.2 Testing

```
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/dir2/foo.txt
$ vmhFS /tmp/tmpFS remove /dir1/dir2/foo.txt
$ vmhFS /tmp/tmpFS remove /dir1/dir2
$ vmhFS /tmp/tmpFS remove /dir1
$ vmhFS /tmp/tmpFS ls / -r
$ vmhFS /tmp/tmpFS remove /
```

Code block 7: *Remove command*

```
Write file /tmp/foo.txt to filesystem at: /dir1/dir2/foo.txt
File /dir1/dir2/foo.txt has been remove from the file system
Directory /dir1/dir2 has been remove from the file system
Directory /dir1 has been remove from the file system
List segment:
/
Root directory cannot be removed
```

Code block 8: *Command output*

5 The write command

5.1 How it works

write: takes as argument a string input file stored in the computer to write at a specified *destination_path*.

We start with some error handling: we check if the given file exists already or not by using the function *find_file_from_path*. This function returns -1 if the file doesn't exist otherwise it returns the index of the file in the file array.

After this step we use the function *create_dir_from_path* to check whether the path exists or not, if it does the function proceeds without changing anything. If it doesn't exist or its missing some directories it creates them.

Next, we create a struct of type File where we put the content of the input file and the information related to it thanks to the function *get_file*. Before proceeding we check if the size of the file doesn't surpass 5MB.

Now, to add the File struct that we just created to the file system we need to check first if the number of bytes of the file plus the size of one inode can fit in the current file system. If it is the case we add it directly using the function *add_file*. This function first checks if the file array is empty,if it is we use the the LibC function *malloc* to allocate space for the new file. If there are already some files in the file system, we use *realloc* to allocate more space for the new file. We treated the first case separately to avoid using *realloc* on empty array as it can result in some memory problems.

Moving on to the second case now where there isn't enough space to store the new file. Here we should proceed using the FIFO principle. The first case is where the file array contains only one file, we shall remove it using the function *remove_file* that we discussed earlier in the remove command. Next, we add it to the file system using the function *add_file*. We treat this case separately to avoid using the *realloc* on on empty array. In the second case the file system contains at least two files, we go into a while loop that counts the number of files to be removed from the file system starting starting from the oldest , in order for our new file to fit.

Now we should shift the files to the left in the file array by the number of files that should be removed (that we counted earlier with the while loop). Afterwards we reallocate the file array to fit the current number of files. Eventually we add the new file at end of the file array.

5.2 Testing

```
$ vmhFS /tmp/tmpFS write /tmp/4mb /dir1/dir2/4mb1
$ vmhFS /tmp/tmpFS write /tmp/4mb /dir1/dir2/4mb2
$ vmhFS /tmp/tmpFS write /tmp/4mb /dir1/dir2/4mb3
$ vmhFS /tmp/tmpFS ls / -r
```

Code block 9: *Write command*

```
Write file /tmp/4mb to filesystem at: /dir1/dir2/4mb1
Write file /tmp/4mb to filesystem at: /dir1/dir2/4mb2
Write file /tmp/4mb to filesystem at: /dir1/dir2/4mb3
List segment:
/
L__dir1
    L__dir2
        L__4mb2 [4194304 B]
        L__4mb3 [4194304 B]
```

Code block 10: *Command output*

6 The size command

6.1 How it works

size: computes the size of files inside a given directory. If the -r flag is indicated, size should recursively calculate the size of files inside that directory.

This function takes as an argument the directory path. We use the function *find_dir_from_path* to check if the path is correct, if not it returns -1. If the path exists indeed, it returns the directory index in the directory file.

We also use the function *size_dir_files* that calculates the size of files inside a single directory with a simple for loop over the file array.

For the recursive part, we use the function *size_accumulator* that takes in argument an index of a given directory. The base condition of this recursive function is to stop at a directory that doesn't have any children directories. We compute the indices of a given directory by using the function *get_dir_children* that uses another simple for loop over the directory array. Its algorithm is very similar to the one used in the recursive ls.

6.2 Testing

```
$ vmhFS /tmp/tmpFS write /tmp/foo.txt /dir1/foo1.txt
$ vmhFS /tmp/tmpFS write /tmp/4mb /dir2/4mb1
$ vmhFS /tmp/tmpFS size /dir1 -b -r
$ vmhFS /tmp/tmpFS size /dir2 -k -r
$ vmhFS /tmp/tmpFS size / -b -stat
```

Code block 11: *Size command*

```
Write file /tmp/4mb to filesystem at: /dir1/dir2/4mb1
Write file /tmp/4mb to filesystem at: /dir1/dir2/4mb2
Write file /tmp/4mb to filesystem at: /dir1/dir2/4mb3
List segment:
/
L__dir1
    L__dir2
        L__4mb2 [4194304 B]
        L__4mb3 [4194304 B]
Write file /tmp/foo.txt to filesystem at: /dir1/foo1.txt
Write file /tmp/4mb to filesystem at: /dir2/4mb1
Total size of files in /dir1: 18 B [Recursive]
Total size of files in /dir2: 4194 KB [Recursive]
Current size of the file system 12583532
Total size of the files: 4194322
Ratio Total files/Total filesystem: 0.33332
Total size of files in /: 0 B [Not recursive]
```

Code block 12: *Command output*

IV Conclusion

All the VMH File system commands work as expected and fulfill the initial requirements. The queue structure of the file system simplified the implementation of the other functions. Furthermore, the advantage of working on the RAM is that we make one load and one save instead of repeating this process multiple times for each micro-modification if we were to have a static structure on the disk. But this approach will only work for very small file systems as mentioned before. To counter this problem, a possible solution is to segment the file system and load to the RAM segment by segment.