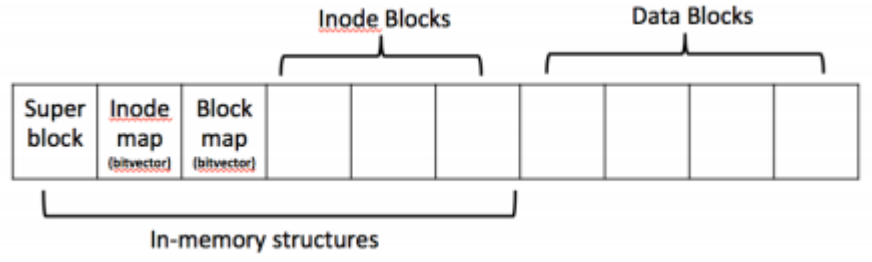
EL MOUAOUINE Mehdi
MAALOUL Amira

Technical Report

In this project, we have implemented a simple UNIX-like file system simulator. We can create and delete files and directories, read and write files. The functionality of the file system is similar to UNIX file systems, but it does not include per-process open file table or handling of concurrent accesses.

This figure shows the file system structure in our simulator. A size of block is 512 byte and the disk has 4096 blocks. Superblock, inode map, block map and inode blocks are loaded into the memory at file system mount time.



The file system will only use direct blocks in the inode to support small size files (up to 5120 bytes). Each file in a file system has an identification number, called inode number, which is unique in a file system. Inode structure (Inode) is defined in “fs.h” file.

Our data structures:

```
//Super Block data structure
typedef struct
{
      int freeBlockCount;
      int freeInodeCount;
      char padding[504];
} SuperBlock;
```

```
//iNode Information
typedef struct
{
      TYPE type;
      struct timeval lastAccess;
      struct timeval created;
            int owner;
            int group;
      int size;
      int blockCount;
      int indirectBlock;
            int directBlock[10];
      char padding[24];
} Inode; // 128 bytes
```

EL MOUAOUINE Mehdi
MAALOUL Amira

```
//Each directory entry
typedef struct
{
      int inode;
      char name[MAX_FILE_NAME];
} DirectoryEntry;
```

```
//Entire directory - One data block
typedef struct
{
      DirectoryEntry dentry[MAX_DIR_ENTRY];
      int numEntry;
      char padding[8];
} Dentry;
```

Our algorithms :

In this project we simulate a Unix file system, so we create a virtual “disk” that we can mount and unmount. When we mount a virtual disk file, we first check if the file already exists and load the superblock, InodeMap, BlockMap and inodes into the memory. If the file does not exist, we create a disk structure which means we initiate the file system superblock inodeMap and blockMap and the root directory.

Files creation :

We create files by taking in the name and the size from user input, so first we verify the inputs (names, and size) if they already exist or size of the file is too big then we initialize the number of blocks (size/block_size) then we verify if there is enough space and inodes to add the file then we get an inode and we fill it then we set the file type, owner and group then we set time for creation and access then we set the size and blockCount then we check if a file was already removed from our disk then we search for all entries for first empty spot then we add a new file into the current directory entry and we get data blocks.

We use practically the same concept for all our functions.

File reading :

This method will read the size passed in from the file offset, so we use two variables fileContents and tempContentsHolder, first we check if there is user input errors then we clear the buffer to avoid a segmentation fault then we proceed by getting the inode of the file then again we check for errors and we clear buffer to avoid segFault then we get the number of blocks to iterate through then we use a for loop through blocks, we get the directBlock number we pull contents helded is disk at that block then we pass it to the temporary holder then we get the file size to print its content.

EL MOUAOUINE Mehdi
MAALOUL Amira

We used utility functions such as get_free_inode or get_free_block that are well commented and detailed in our code.

This is the algorithms that we think that we need to explain them is this doc the rest of the functions are explained is the code.

Source code:

fs_util.c : contains utitlity functions

fs.c : contains all the file system functions

disk.c: contains virtual disk management functions

References:

- Operating systems course
- Code source found on Internet: https://www3.nd.edu/~dthain/courses/cse30341/spring2017/project6/