*CIS 3207, Section 4*

*Tuyen Pham*

*TUID: 915591991*

**PROJECT 4A: DESIGN OF THE SIMPLE FILE SYSTEM**

1. ***Overview***

This purpose of this project is to implement a simple file system on top of a virtual disk and to understand implementation details of library functions that offer a file system. To be specific, the file data and file system will be stored on the virtual disk, which has 16,384 blocks and each block holds 4kB. On the other hand, the virtual disk is a single file which can be stored on system provided by the Linux operating system. In addition, the file system does not have to store more than 256 files, which means out of 16,384 blocks available on a disk, only 8,192 must be reserved as data blocks.

1. ***Design illustration***

The structure I am planning is the File Allocation Structure (FAT). I also plan to use the logical directory of files and folders. FAT will stem from Fat – Root Directory – Data Blocks style filesystem. For the storage, this will take (size of int\*(size of the drive over the size of a block)) bytes in the drive. For the first entry in the FAT system would be initialized -1 for the chain and 0 indicates the block has not begun yet. The root directory starts with block 0, this means nothing can link back to the root directory, making the block chains discourteous and the block x in the system is made FAT[x].

Considering every piece of data will hold some block address, or own a certain percent of the block, every directory and file will have a block address associated with that address. If the block contains a file, or directory listing, the block may or may not depend on the testing hold one or serval data structs, all of which have a defined size. Defining the type of struct may be helpful as well, whether dealing with a file or a directory.

1. ***Data Structure***

The file system or a disk is consisted of 64 blocks, each of which has 16 bytes. 32 blocks are for meta-data and the other are for file data. The type of all data is stored in the file system and each character’s size is one byte, so we can store 16 characters in a block. The metadata of the file system includes the root directory, FAT and open file table. In my design, the root directory occupies 4 blocks, which has 64 bytes in the disk file and it also stored in array DIR [8][8] in memory for operation. Let use i to denote the index of a file so DIR[i][0] denotes the status of a file, DIR [i][1] denotes the number of the first block containing the file data, DIR [i][2] to DIR [i][5] denote the file name with at most four characters, DIR [i][6] denotes the length of the file and DIR [i][7] denotes the corresponding file descriptor if the file is opened. In addition, FAT occupies 4 blocks (64 bytes) in the disk file and it is also stored in array FAT [32][2] in memory. Let’s use i to denote the index of a data block. FAT [i][0] denotes the status of the block, free or used. FAT [i][1] denotes the number of the next block containing a file’s data if the file exists. After required operations, the root directory and file allocation table in the disk file will be updated according to DIR [8][8] and FAT [32][2]. The open file table is only stored in memory and will be reset to original state after is disk is dismounted and closed.

1. ***Details Implementation of Pseudocode***

File system functions are based on the function in disk.c which is already provided, so I design 12 file system functions as the following:

1. make\_fs(char \*disk\_name): Firstly, check whether the disk\_name is legal. If the disk\_name is consisted of no more than 4 characters, it is legal and I use make\_disk() function to make a file system and create a disk file. Then I initialize the three tables in memory: DIR[8][8], FAT[32][2] and OFT[4][3].
2. mount\_fs(char \*disk\_name): Firstly, open the disk file and copy the metadata in the disk file to buffers in memory by using block\_read() function. The metadata is read block by block. Then update the information of DIR[8][8] and FAT[32][2] according to the data structure of the metadata.
3. unmount\_fs(char \*disk\_name): Firstly, copy 16 bytes of the DIR[8][8] or FAT[32][2] every time to a buffer. And then use block\_write() function copy the data in the buffer to the disk file. OFT[4][3] do not need to be copied. At last, close the disk file with close\_disk() function and reset DIR[8][8], FAT[32][2] and OFT[4][3] to the original state.
4. fs\_open(char \*name): Firstly, check whether the file exists, whether the file is already open and whether there is available file descriptor for use. If not, there is a failure in opening the file and return -1. Otherwise, search for an available file descriptor with first fit algorithm. If the file is opened for the first time, search for an available data block with first fit algorithm. Then update the relevant information and return the file descriptor.
5. fs\_close(char \*name): update relevant information in DIR[8][8] and OFT[4][3]. And decrease the number of used file descriptors.
6. fs\_create(char \*name): Firstly, check whether the file with name exists or not and whether the number of files exceeds 8. Then search for an available index in DIR[8][8] with first fit algorithm.
7. fs\_delete(char \*name): Firstly, check whether the file with name exists and whether the file is open or not. Then free the data blocks of the file by set the status of these blocks to 0. At last, remove all the information of the file in DIR[8][8].
8. fs\_read(int fildes, void \*buf, size\_t nbyte): Firstly, copy all the data of the file in the disk file to a buffer in memory. Then copy the required part of the file to buf[] according to the offset and nbyte.
9. fs\_write(int fildes, void \*buf, size\_t nbyte): Consider different situations and handle them one by one. When the file pointer is at the beginning, the end or between the beginning and the end of the file, different solutions are provided.
10. fs\_get\_filesize(int fildes): return the length of file stored in DIR[8][8].
11. fs\_lseek(int fildes, off\_t offset): Update the offset of an open file in OFT[4][3].
12. fs\_truncate(int fildes, off\_t offset): Calculate the blocks the file needs after truncating. And free the blocks the file do not need any more. Then update the information of DIR[8][8] and FAT[32][2].
13. ***Testing***

As for testing purposes, I intend to run the drive with different commands and outputs to make sure the virtual drive was created, make sure to handle these functions above and then try out other functions to support for the file system.