**FILESYSTEM LAB**

**OBJECTIVE**

Design a file system that has a directory structure, which can store subdirectories and files with the following requirements:

- Directory and filenames have to be a maximum of 8 characters, and filenames can have a 3 characters extension.

- Maximum file size is 32768 bytes

- Block size (allocation units) for the file system must be 512 bytes

- All the directories must be linked in some way.

- Every entry in a directory must have a time stamp indicating the last time the entry was modified, and the time it was created.

- A logical directory that describes files and attributes/meta data

- A physical directory that organizes blocks allocated to each file.

The file system must also have basic input/output control system functions. The input/output control system functions include:

- Create file

- Open file

- Delete file

- Close file

- Read file

- Write file

Lastly, an application must be made that can demonstrate the functionality of the created file system.

**TOOLS USED TO COMPLETE PROJECT**

I used the Project 4 Outline of File System Design Notes.pdf and Laboratory Project 4 Creating a File System.pdf on Blackboard as a guideline when implementing my file system.

I used the following sites for insight on the FAT filesystem

<https://en.wikipedia.org/wiki/File_Allocation_Table>

<https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system>

**LIBARARIES USED**

#include <stdio.h> - for basic C functions

#include <time.h> - for time based functions

#include <string.h> - for string related functions

#include <stdlib.h> - for memory related functions

**MY IMPLEMENTATION**

I implemented the FAT file system on a 3MB virtual disk. This file system will only work for a 3MB disk. Moreover, I viewed this disk with a base 10 perspective, so I see the disk as 3000000 bytes (there will be unused bytes due to using the base 10 perspective). I used the base 10 perspective due to ease of use when building the system, as my computer looks at the virtual file with a base 10 perspective.

This file system will be made up of 5859 blocks. Each block is 512 bytes.

As I stated in the previous documentation for the file system. My file system will be split into four regions.

|  |  |  |  |
| --- | --- | --- | --- |
| INFO BLOCK  REGION  (1 block) | FAT REGION  (22 blocks) | ROOT DIRECTORY REGION  (204 blocks) | DATA REGION  (5632 blocks) |

Info Region – will contain the date the virtual disk was formatted in the form month/day/year.

FAT Region – will be composed of my FAT. Each entry is a 2 byte short, which maps to a block in the data region. The value of each entry represents something different. For example:

* -2 = This entry points to nothing. (Free entry)
* -1 = This entry points to nothing but is allocated to a file.
* Any other value represents the address for another FAT entry.

Root Directory Region – will be composed of my root directory. Each entry in my root directory is 48 bytes. Although I didn’t use a struct directly in my implementation. A root directory entry can be represented by the following struct:

struct root\_directory\_entry {

char flag; (size: 1 bytes)

* this flag will determine if the entry is a file or directory.
* d = directory, f = file, null character = empty entry.

short open; (size: 2 bytes)

* This open flag will tell you if a file is open or closed.
* -1 = closed. 0 = open
* Directories will always be open.

short FAT\_address; (size: 2 bytes)

* this variable will point to starting address in the FAT

char name[8]; (size: 8 bytes)

* this array will hold the name of a file/directory

char ext[3]; (size: 3 bytes)

* this array will hold the extension of a file, if there is one.

int creation\_year; (size: 4 bytes)

* this variable will hold the value of the creation year.

int creation\_month; (size: 4 bytes)

* this variable will hold the value of the creation month.

int creation\_day; (size: 4 bytes)

* this variable will hold the value of the creation day.

int mod\_year; (size: 4 bytes)

* this variable will hold the value of the mod year.

int mod\_month; (size: 4 bytes)

* this variable will hold the value of the mod month.

int mod\_day; (size: 4 bytes)

* this variable will hold the value of the mod day.

int parent; (size: 4 bytes)

* Holds the index of the parent directory

int size; size: 4 bytes

* Represents number of bytes allocated to a file.

};

Data Region – will contain the actual information of a file. Mapped to entries in the FAT. If a byte in this region is empty then it will be equal to the null character.

I used offset arithmetic to navigate the virtual disk and write to it. For the root directory entries, I used the sizes listed in the struct above as offsets when navigating a root directory entry.

When navigating the FAT, I moved in offsets of 2, or via the links of the FAT. When I wanted to go to the data region block mapped to the FAT entry I used the following formula:

(FAT ADDRESS-BLOCK\_SIZE)/2;

FAT ADDRESS is the address of the FAT entry.

BLOCK-SIZE is 512. I subtracted the FAT ADDRESS by 512 because the info block at the beginning of the file system, shifts the index of the FAT entries by 512.

I divide by 2, since each entry is made up of 2 bytes. So every 2 bytes in the FAT region map to a block in the data region.

USAGE

When providing path to functions it must be in this form /ROOT/subdirectoryA/subdirectoryB/…/

When providing a filename to functions it must be in the form name.ext for files. Where name can only be a maximum of 8 characters, and ext a maximum of 3 characters.

When providing a directory name it must not contain an extension. It’s name must also be a maximum of 8 characters long.

*This usage for the client will mainly be used for the create\_file, read\_file, open\_file, close\_file, write\_file and delete\_file functions.*

MACRO FUNCTIONS

# define BLOCK\_SIZE 512 - This macro defines the block size.

# define TOTAL\_BLOCKS 5859 - This macro defines the total number of blocks in the file system.

# define FAT\_REGION\_START 1 - this macro defines the index where the fat region starts. The value is one because index 0 is reserved for the info block region.

# define FAT\_REGION\_END 22 - This macro defines where the fat region ends.

# define ROOT\_START\_INDEX 23 - This macro determines where the Root Directory region starts

# define ROOT\_END\_INDEX 227 -This macro determines where the root directory region ends

# define DATA\_REGION\_START 228 - This macro determines where the data region starts.

#define MAX\_FILE\_SIZE 32768 – This macro determines the max file size.

FUNCTIONS

void Mount() – This function opens the virtual disk

void UnMount() – This function closes the virtual disk

void Format() – This function will format the disk, to make the disk usable for file system operations. This function will delete everything in the disk. Initializes the info region to null character. Initializes the FAT region to -2. Initializes the Root Directory Region and Data Region’s bytes to the null character. This function used fseek/fwrite to traverse through the disk, and fwrite was used to set the default values of each region.

int check\_path(char \*path) – This function will determine if a path is valid. It delimits the argument path with “/” as the delimeter. This is done through strtok. Using the delimited tokens from the path argument, this function checks if the proper parent directory was given for each token. Returns the address of the lowest subdirectory described in path if the path is valid, otherwise it returns -1 on failure.

int search\_Root\_Dir(char \*name, char \*ext) – This function will look for a file with the name and extension provided in the arguments in the Root Directory. Returns the address of the file’s root director entry on success. On failure the function returns -1.

int search\_empty\_Root\_dir() – searches for an empty root directory entry in the Root Directory region. Address of empty root directory entry returned on success. Otherwise, -1 is returned on failure.

int search\_root\_For\_File(int offset) – this function is like the search\_Root\_Dir() function, however a file is searched for using it’s root directory entry offset.

int search\_empty\_FAT\_entry() – this function looks for an empty FAT entry. Returns the address of the empty FAT entry on success, otherwise -1 is returned on failure.

void create\_root\_dir\_entry(int root\_offset, int parent\_value, char \*name, char \*ext, short Fat\_addr) – This function will create a root directory entry with the arguments provided. These parameters are written into the virtual disk using fwrite. The disk is traversed using fseek.

int change\_FAT\_entry(int offset, short value) – This function will change the value of with the short value a FAT entry implied by offset parameter.

int create\_file(char \*path, char \*name) – This function creates either a file or directory. This function will first parse the name parameter separating the extension and name. It will check if the path name and file/directory name is valid. Depending on the extension it will create a file by creating a root directory entry, and a corresponding file entry.

int open\_file(char \*path, char \*name) – This function will change the open flag in a file’s root directory entry to reflect that a file is open. The value for an open flag is 0.

int close\_file(char \*path, char \*name) - This function will change the open flag in a file’s root directory entry to reflect that a file is closed. The value for an open flag is -1.

int is\_File\_open(int root\_offset) – this function will check if a file is open or closed. It used the root\_offset parameter to traverse to the file’s root directory entry.

short get\_starting\_FAT\_addr(int root\_offset) - This function will retrieve the address of the first FAT entry of a file. It returns this address.

int get\_parent(int root\_offset) -This function will retrieve the address of the parent directory of a file. It returns this address.

int get\_size\_of\_file(int root\_offset) – This function will return the size of a file indicated in its root directory entry.

short get\_FAT\_value(short fat\_offset) – This function will return the value of the FAT entry specified by the fat\_offset.

char\* read\_file(char \*path, char \*name) – This function will read the info stored in a file into buffer and return it to the client. It used the file’s first FAT entries to go to the data region, and read the information stored in the file.

int change\_modtime(int root\_offset) – This function will change the medication time variable’s in a file’s root directory entry.

int write\_file(char \*path, char \*name, char \*buf, int buf\_size) – This function will write whatever is in the buf array into a file. The buf-size variable represents how big the buf array is. If the array is too big -1 is returned or if the file is full.

int change\_parent\_value\_to\_root(int offset) – Changed a file’s parent directory to the root directory.

int delete\_file(char \*path, char \*name) – This function deletes a file or directory from the file system. If deleting a directory then all of it’s file’s will have their parent directory changed to the Root directory. The deleted file/directory’s root directory entry’s flag is changed to the null character. Their fat entries are changed to -2, and the corresponding data blocks are changed to the null characters. Returns one on success and -1 on failure.

**TESTING**

To test my file system, I used various print statements to make sure the flow of the program was correct. I also used hex editors to make sure the correct values were being written into the disk.

**KNOWN BUGS**

The filesystem needs a very specific path to be entered for the create, delete, open, close, read, and write functions. It’s usage is explained above.

**CONCLUSION**

Problems I Faced

The biggest problem I faced was the arithmetic needed to traverse the virtual disk. I kept on getting the arithmetic wrong, which led to values being written in the wrong place.

I also experienced a lot of seg faults due to poor memory management.

Learning Outcome

From this project I learned how to implement the FAT file system. I also learned how file’s, and directories work. I also learned the importance of memory management, as I kept getting segmentation faults due to improper usage of memory.