CSC-415 Operating Systems

File System Project

*Group Submission*

Team:

Diligence

GitHub:

<https://github.com/CSC415-2022-Fall/csc415-filesystem-mkim797>

|  |  |
| --- | --- |
| Name | Student ID |
| Mark Kim | 918204214 |
| Peter Truong | 915780793 |
| Zeel Diyora | 920838201 |
| Chengkai Yang | 921572896 |

Table of Contents

[Description 2](#_Toc120204899)

[fsInit.c 2](#_Toc120204900)

[fsFree.c 2](#_Toc120204901)

[fsDir.c 2](#_Toc120204902)

[FsHelpers.c 3](#_Toc120204903)

[Issues Encountered 3](#_Toc120204904)

[Driver Program Operation 4](#_Toc120204905)

[Screenshots 4](#_Toc120204906)

[Commands shown in screenshots 4](#_Toc120204907)

[Compile 5](#_Toc120204908)

[Initialize New Volume; md, touch, pwd, ls -la, cd, cp2fs, rm (directory) 5](#_Toc120204909)

[Load Existing Volume; cd, pwd, cp, mv 6](#_Toc120204910)

[Load Existing Volume; ls, cd, mv, cp, pwd 7](#_Toc120204911)

[Load Existing Volume; cat 8](#_Toc120204912)

# Description

This project is a file system using FAT for free space and volume management. The portion of code that our team completed is split into 5 sections (not including the fsshell.c. file and mfs header). These sections are fsInit, fsFree, fsDir, fsHelpers, and b\_io, which handle file system initialization, free space management, directory management, misc helper functions, and buffered I/O for reading and writing to the file system.

Since we were using FAT to track the free space and the volume, our free space map occupies much more space than a bitmap would have. However, the use of FAT allowed us to dynamically manage non-contiguous allocation, reallocation, and deallocation of free space as we read and write files and directories. Since each block of the free space map only knew its own location and the next location (a linked list), handling the free space only required us to iterate through each file/directory’s maps and the free space map and link up the ends to the beginnings as those requests are made. One disadvantage of this, however, is that one cannot jump directly to a particular block of the file system and to reach any block, we have to iterate through the entire list until we reach the block that we are looking for. From a design perspective, it simplifies some of the code, but also required us to think about how one would traverse a file by the block.

## fsInit.c

This portion of the program simply initializes the file system by creating/loading the volume control block, initializing the free space, and setting the current working directory array and path. The initialization is fairly short as the bulk of the work is contained in the free space management (fsFree.c). This file also contains the logic for freeing any malloc’d memory.

## fsFree.c

This section manages the free space map. We have an init\_free() function that initializes the free space map by filling the FAT with the references to the next block and flagging the final block to 0xFFFFFFFE. All the other functions manage the allocation of free space or retrieving the location of a block on the volume.

## fsDir.c

This file contains the functions that manage directories and directory entries. The init\_dir() function initializes directories. parsePath() is likely one of the most important functions of the file system; many functions depend on this function. It takes a path string, tokenizes it, then navigates through the directory tree using those tokens. If it encounters an invalid entry, it simply returns an error. Otherwise, it returns the directory array associated with the final entry of the path (but does nothing with the final entry). It is the caller’s responsibility to do something with the directory array returned. In addition to the functions mentioned above, this file also includes many other functions meant to validate, navigate or mutate directories.

## FsHelpers.c

Any helper functions that are not really associated with free space or directory management are contained here. We have functions that do such things as setting the current working path, retrieving the final token in a path string, calculating the number of blocks a byte count would occupy, and writing to disk.

# Issues Encountered

This file system was certainly difficult. Although we put in a countless amount of time planning and trying to figure out the structure of the system, the plans were often missing small details that would have giant unforeseen consequences which would require us to either refactor large portions of code, or just complete scrap entire sections.

One such small detail was how we would manage the path for the current working directory. We initially approached it by trying to mutate the path string as directory changes occurred. After many hours of frustration and failure on this, we ended up just scrapping all our work on it in favor of just navigating the entire directory tree backwards down to the root directory and constructing the path that way for every change.

We ran into many issues regarding jumbled data results from our b\_write and b\_read functions. This occurred because it was not readily apparent how to separate the cause of write versus read. We initially just copied files over to the file system then copied them back to Linux, but we weren’t sure if it was being caused by write, read, or both. This took a lot of work to pour over the hexdump, which was extremely time-consuming and tedious. It wasn’t until we realized that we could just read the SampleVolume sans hexdump to compare what was written on the file system with the payload. This was a massive time-saver and would’ve been a useful bit of knowledge that wasn’t readily apparent.

On a similar note, we got the file system working with a buffer size of 200, but once we started switching to buffer sizes greater than the block size, we once again ran into issues. These issues were caused by “part2” of the read and write. At first, we could not figure out what was causing it, but it should’ve been obvious to us that because we were using FAT, we could not simply write consecutive blocks in one big chunk. We switched to writing one block at a time and iterating through the free space incrementally. I am sure we could implement optimized code that would be able to write contiguous blocks, but since this method works, we simply said it is “good enough.”

Another point to note is that we are aware of memory leaks in the system. Although we could hunt those memory leaks down, we again decided that what we have is “good enough.”

# Driver Program Operation

The driver program fsshell.c is largely untouched from the original provided. We implemented randomization for the buffer size just to rigorously test our file system. We also implemented the cmd\_mv() function, but the actual logic of move is contained in the fsDir.c file. Finally, we added a fair number of extra functions to the shell to help us debug the file system. Those extra functions mostly call lower level functions and print out some result from the function call. This helped us tremendously as it provided us with really important data on what was occurring in our file system as we ran specific functions. These test functions include:

1. pp: parsePath() – print out the entire directory that parsePath returns
2. isfile: fs\_isFile()
3. isdir: fs\_isDir()
4. opendir: fs\_opendir()
5. getcwd: fs\_getcwd()
6. readdir: fs\_readdir()
7. getde: print directory entry to console

# Screenshots

## Commands shown in screenshots

ls - Lists the file in a directory

cp - Copies a file - source [dest]

mv - Moves a file - source dest

md - Make a new directory

rm - Removes a file or directory

touch - creates a file

cat - (limited functionality) displays the contents of a file

cp2l - Copies a file from the test file system to the linux file system

cp2fs - Copies a file from the Linux file system to the test file system

cd - Changes directory

pwd - Prints the working directory

history - Prints out the history

help - Prints out help

## Compile

Text

Description automatically generated

## Initialize New Volume; md, touch, pwd, ls -la, cd, cp2fs, rm (directory)

Text

Description automatically generated

## Load Existing Volume; cd, pwd, cp, mv

We also did a diff comparison between files after copying it to and back from the file system

Text

Description automatically generated

## Load Existing Volume; ls, cd, mv, cp, pwd

Text

Description automatically generated

## Load Existing Volume; cat

Text

Description automatically generated

Diagram, schematic

Description automatically generated