COP4610
Introduction to Operating Systems
Project #3:
Implementing a FAT32 File System

Important Information

Deadline:
Project is due on **April 21st at 11:59pm**. No late submissions.

Grader:
Juan Pablo Conde (jc18ha@my.fsu.edu)
Office hours: Tuesdays & Wednesdays 1:30PM-2:30PM. MCH 102D

Submission:
- Compress your project files into a .tar file and submit through Canvas
    - Include only the folder contents, not the folder itself
    - File name format: project3_lastname1_lastname2_lastname3.tar
- Your project **must include**:
    - All the code files (.c and .h)
    - Makefile
    - README file
    - Git commit log (please use a private repository!)
    - Do not include binaries or executables
- Only submit once per team.
- Report your division of labor in the README file.
- You **must report** all the extra credit tasks developed in the README file in order to get extra credit.

Makefile Output:
Your Makefile must generate an executable file named **project3**.

Grading Policy:
- Total points for implementation: 70
- Total extra points for implementation: 7 (+3 if well documented)
- Total points for documentation: 30 (refer to project syllabus for details)
- Omitting the Makefile will result in a 70-point deduction
- Projects that do not compile will receive no more than 30 points

Teams:

- Must work in teams of **3 members**.
- Submit your team members' information to Canvas (People > "Project3" tab).
- Working alone is possible but must get approval by TA and it is not encouraged.
- Distribute your work evenly. Team members that do not contribute equally to the project will have points deducted.
- Report when you work together (and on what) in cases you do not split workload or work on the same computer.
- Report team issues to TA as soon as possible.

Refer to the Project Syllabus for more details about the projects, grading policy, and submission: https://canvas.fsu.edu/courses/113040/files/7041532/download?wrap=1

Recommended reading:
In order to perform the tasks, students are recommended to read and understand the FAT32 specification (uploaded to Canvas).

Project Overview

The purpose of this project is to familiarize you with basic file-system design and implementation. You will need to understand various aspects of the FAT32 file system such as cluster-based storage, FAT tables, sectors, directory structure, and endianness.

Problem Statement

For this project, you will design and implement a simple, user-space, shell-like utility that is capable of interpreting a FAT32 file system image. The program must understand the basic commands to manipulate the given file system image, must not corrupt the file system image, and should be robust. You may not reuse kernel file system code and you may not copy code from other file system utilities.

Project Tasks

You are tasked with writing a program that supports the following file system commands to a FAT32 image file. For good modular coding design, try to implement each command in one or more separate functions (e.g. for write you may have several shared lookup functions, an update directory entry function, and an update cluster function).
Your program will take the image file path name as an argument and will read and write to it according to the different commands. You can check the validity of the image file by mounting it with the loop back option and using tools like Hexedit.
You will also need to handle various errors. When you encounter an error, you should print a descriptive message (e.g. when cd'ing to a nonexistent file you can do something like "Directory not found: foo"). Further, your program must continue running and the state of the system should remain unchanged as if the command were never called (i.e. don't corrupt the file system with invalid data).

`1. exit [2]`
Safely close the program and free up any allocated resources.

`2. info [4]`
Parse the boot sector. Print the field name and corresponding values for each entry, one per line (e.g. Bytes per Sector: 512).

- The fields you need to print out are
  - *bytes per sector*

- *sectors per cluster*
- *reseverd sector count*
- *number of FATs*
- *total sectors*
- *FATsize*
- *root cluster*

3. size FILENAME [3]
Prints the size of the file `FILENAME` in the current working directory in bytes.
- Print an error if `FILENAME` does not exist.

4. ls DIRNAME [5]
Print the name field for the directories within the contents of `DIRNAME` including the "." and ".." directories.
- For simplicity, you may print each of the directory entries on separate lines.
- Print an error if `DIRNAME` does not exist or is not a directory.

5. cd DIRNAME [4]
Changes the current working directory to `DIRNAME`.
- Your code will need to maintain the current working directory state.
- Print an error if `DIRNAME` does not exist or is not a directory.

6. creat FILENAME [5]
Creates a file in the current working directory with a size of 0 bytes and with a name of `FILENAME`.
- Print an error if a file with that name already exists.

7. mkdir DIRNAME [5]
Creates a new directory in the current working directory with the name `DIRNAME`.
Print an error if a file called `DIRNAME` already exists.

8. mv FROM TO [5]
Moves a file or directory or changes the name of a file or directory.
- If `TO` exists and is a directory, the system will move the `FROM` entry to be inside the `TO` directory.
- If `TO` does not exists, the system will rename the `FROM` entry to the name specified by `TO`.
- If `TO` exists and is a file and `FROM` is a file too, print an error "The name is already being used by another file".
- If `TO` is a file and `FROM` is a directory, print an error "Cannot move directory: invalid destination argument". Must also work with `..` and `.` entries.

9. open FILENAME MODE [8]
Opens a file named `FILENAME` in the current working directory.
- A file can only be read from or written to if it is opened first. You will need to maintain a table of opened files and add `FILENAME` to it when open is called `MODE` is a string and is only valid if it is one of the following:
  - r – read-only
  - w – write-only
  - rw – read and write
  - wr – write and read
- Print an error if the file is already opened, if the file does not exist, or an invalid mode is used.

10. close FILENAME [4]
Closes a file named `FILENAME`.
- Needs to remove the file entry from the open file table.
- Print an error if the file is not opened, or if the file does not exist.

11. `read FILENAME OFFSET SIZE [8]`

Read the data from a file in the current working directory with the name `FILENAME`.

- Start reading from the file at `OFFSET` bytes and stop after reading `SIZE` bytes.
- If the `OFFSET`+`SIZE` is larger than the size of the file, just read size-`OFFSET` bytes starting at `OFFSET`.
- Print an error if `FILENAME` does not exist, if `FILENAME` is a directory, if the file is not opened for reading, or if `OFFSET` is larger than the size of the file.

12. `write FILENAME OFFSET SIZE "STRING" [8]`

Writes to a file in the current working directory with the name `FILENAME`.

- Start writing at `OFFSET` bytes and stop after writing `SIZE` bytes. You will write `STRING` at this position.
- If `OFFSET`+`SIZE` is larger than the size of the file, you will need to extend the length of the file to at least hold the data being written.
- If `STRING` is larger than `SIZE`, write only the first `SIZE` characters of `STRING`. If `STRING` is smaller than `SIZE`, write the remaining characters as ASCII 0 (null characters).
- Print an error if `FILENAME` does not exist, if `FILENAME` is a directory, or if the file is not opened for writing.

13. `rm FILENAME [5]`

Deletes the file named `FILENAME` from the current working directory.

- This means removing the entry in the directory as well as reclaiming the actual file data.
- Print an error if `FILENAME` does not exist or if the file is a directory.

14. `cp FILENAME TO [4]`

Copies the file specified by `FILENAME`.

- If `TO` is a directory, copy the file directly into a folder.
- If `TO` is not a valid entry, create a copy of the file within the current working directory, and assign it the name given by `TO`.
- Print an error if the file specified by `FILENAME` does not exist.

Extra Credit

15. `rmdir DIRNAME [3]`

Removes a directory by the name of `DIRNAME` from the current working directory.

- This command can only be used on an empty directory (no directory entries except `.` and `..`).
- Make sure to remove the entry from the current working directory and to remove the data `DIRNAME` points to.
- Print an error if the `DIRNAME` does not exist, if `DIRNAME` is not a directory, or `DIRNAME` is not an empty directory.

16. `cp -r FROM TO [4]`

Recursively copy the directory indicated by `FROM` (and all its contents) to the directory indicated by `TO`.

- `FROM` must be a valid directory
- If `TO` does not exist, copy the folder (and all its contents) to the current directory and assign the new folder the name given by `TO`.
- Print an error if `FROM` does not exist or is not a directory.

Restrictions and Allowed Assumptions

- Must be implemented in the C programming language
- File and directory names will not contain spaces and no file extensions (ie .txt)
- File and directory names will be names (not paths) within the current working directory.
- You do not need to support long directory names, but these entries may exist in the image; you can safely skip over these entries
- You do not need to update any fields not needed for shell operation and not specified in the project requirements
    - for example, you do not need to update file timestamps
- `.` (current directory) and `..` (parent directory) are valid names
- STRING will always be contained within " characters