Project 3 Specification
FAT32 File System

Due: Friday April 27th 2018, 11:59pm
Absolute Deadline!

Purpose
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 an nonexistent file you can do something like "Error, no such file or directory: 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).

For understanding the layout of FAT32, there will be a provided specification, but the best place to start is the FAT32 wikipedia page https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system. This will teach you how to find FAT entries, clusters, parse the boot sector, parse directory entries, etc.

Part 1: `exit`
Safely close the program and free up any allocated resources.

Part 2: `info`
Parse the boot sector (the FAT32 Extended BIOS Parameter Block in the wiki link; some entries maybe entire tables from previous versions, make sure to get these as well) and print the values of each of the entries in order, one per line (e.g. Bytes per Sector: 512).

Use your discretion on how to print the values (e.g. strings of decimal digits, hexidecimal digits, or ASCII characters), and on what to label each of the values. You are free to skip over any entry that is more than 50 bytes long.

Part 3: `ls` ***DIRNAME***

Print the contents of `DIRNAME` including the "." and ".." directories. For simplicity, just print each of the directory entries on separate lines (similar to the way `ls -l` does in Linux shells)

Print an error if `DIRNAME` does not exist or is not a directory.

Part 4: `cd` ***DIRNAME***

Changes the current working directory to `DIRNAME`. Your code will need to maintain the current working directory state in order to do pathname resolution for the various files.

Print an error if `DIRNAME` does not exist or is not a directory.

Part 5: `size` ***FILENAME***

Prints the size of the file `FILENAME` in the current working directory in bytes.

Print an error if `FILENAME` does not exist.

Part 6: `creat` ***FILENAME***

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.

Part 7: `mkdir` ***DIRNAME***

Creates a new directory in the current working directory with the name `DIRNAME`.

Print an error if a file called `DIRNAME` already exists.

Part 8: `rm` ***FILENAME***

Deletes the file named `FILENAME` from the current working directory. This needs remove 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.

Part 9: `rmdir` ***DIRNAME***

Removes a directory by the name of `DIRNAME` from the current working directory. 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 or if `DIRNAME` is not a directory.

Part 10: `open` ***FILENAME MODE***

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.

Part 11: `close` ***FILENAME***

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.

Part 12: `read` ***FILENAME OFFSET SIZE***

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.

Part 13: `write` ***FILENAME OFFSET SIZE STRING***

Writes to a file in the current working directory with the name FILENAME. Start writing at OFFSET bytes and stop after writing SIZE bytes. 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.

You will write STRING at this position. 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.

Restrictions

- Must be implemented in the C Programming Language

Allowed Assumptions

- File and directory names will not contain spaces
- File and directory names will be names (not paths) within the current working directory
  - “.” (current directory) and “..” (parent directory) are valid names
- STRING will always be contained within “ characters

Project Submission Procedure

Submit a tar archive of your code (no binaries or executables), Makefile, and README to the Canvas as is detailed in the recitation syllabus. Only submit once per team.