FAT12 Design Document

Hanah Leo & Stephen Brikiatis
CSI-385
Champlain College

September 7th, 2016

Not much has changed since the mid evaluation, sorry.

1 Shell Description

The shell will be a simple while loop that will take user input(commands and arguments) which are defined in the project spec PDF. Commands used to interact and navigate directories and files are used in reference to the FAT12 allocation system and executed with the execvp() function. The provided floppy images will be used for testing commands. The user will be able to quit the program by entering ”exit” or ”logout.”

2 Functions

Functions that are used in the implementation and their details. Prototype, parameters, and return value are summarized.

OTHER FUNCTIONS PROTOTYPED IN COMMANDS- WILL BE DEFINED SOON

2.1 bool checkFile(char* x)

Checks whether the file is empty

PARAMETERS:
x: path of file

RETURN: True if file contains data, False is empty

2.2 bool checkRange(int x, int y)

Validates the sector range and outputs an error message if out of range

PARAMETERS:
x: start sector to show entries
y: ending sector

RETURN: true if range is valid, false if invalid

2.3 unsigned int get_fat_entry(int fat_entry_number, unsigned char* fat)

Gets the specified entry from the given FAT

PARAMETERS:
fat_entry_number: The number of the FAT entry to get (0, 1, 2, ...)
fat: The fat table from which to get the specified entry

RETURN: the value at the specified entry of the given FAT

2.4 bool isFile(struct FileData entry)

Checks file entry attributes with bitmasks to see if the entry is a subdirectory or a file
PARAMETERS:
entry: FileData object holding file entry data

RETURN: return true if entry is a file, else return false

2.5 void printBootSector()

Displays the boot sector information saved in a BootSector structure

PARAMETERS:
None

RETURN: *None*

2.6 int readBootSector()

Reads the boot Sector and assigns values to a BootSector structure.

PARAMETERS:

None

RETURN: the number of bytes read, or -1if the read fails.

2.7 int readFile(pointer file_loc, int bytes_in_file, char* buffer)

Starts at the first cluster of data and go through each subsequent and load all the data into a buffer PARAMETERS:
file_loc: the pointer to the first cluster in the file storage bytes_in_file: how large the file is buffer: where the data will be stored after being resized to bytes_in_file RETURN: status of success of reading

2.8 int read_sector(int sector_number, unsigned char* buffer)

Reads the specified sector from the file system and store that sector in the given buffer

PARAMETERS:
sector_number: The number of the sector to read (0, 1, 2, ...)
buffer: The array whose contents are to be read

RETURN: the number of bytes read, or -1 if the read fails.

2.9 bool removeFile

Displays the boot sector information saved in a BootSector structure

PARAMETERS:
None

RETURN: *None*

2.10 void set_fat_entry(int fat_entry_number, int value, unsigned char* fat)

Set the specified entry in the given FAT to the given value

PARAMETERS:
fat_entry_number: The number of the FAT entry to set (0, 1, 2, ...) value: The given value to place in the FAT entry fat: The fat table in which to set the given value at the specified entry

RETURN: *None*

2.11 int write_sector(int sector_number, unsigned char* buffer)

Writes the contents of the given buffer to the filesystem at the specified sector

PARAMETERS:

sector_number: The number of the sector to write (0, 1, 2, ...)
buffer: The array whose contents are to be written

RETURN: the number of bytes written, or -1 if the read fails.

3 Commands

The sequence of operations to be performed and which function will perform the operations.

3.1 pbs

The print boot sector command displays information on the boot sector of the FAT12 system. Since the boot sector is not in 12 bit entries, it can be read more easily. The command ignores any arguments handed to it and uses 2 functions , readBootSector() and printBootSector() to set the values to a BootSector structure which can then be displayed to the console.

The program uses a char buffer which is parsed according to the boot sector attribute length and in the case that the range is greater than 1, uses the bitwise operators OR and LEFTSHIFT in order to combine the int values. For the last few values which are strings, for loops are used in order to read in the values so they can be printed using %s. The BYTES_PER_SECTOR which is modified to only read the necessary boot sector range is reset to the bytesPerSector value for future use.

- This command ignores any arguments the user may append.

3.2 pfe

Print FAT entry takes two arguments-a starting sector and an ending sector-and displays the values in the FAT table in the range specified. The arguments are validated using the checkRange() function which will display an error message should it return false. The readBootSector() function will be used in order to read in data into a buffer and assign appropriate values to the global Boot Sector object which is used to print out the formatted values in printBootSector().

3.3 df

Update this later
The disk free command will display any free logical sectors which will be kept track of through the program. Keep in mind LOGICAL, so the boot sector and the root directory are not counted.

1. Create bitmap(pg 303) mapped to each logical sector in main
2. Set bitmap to reflect file allocation in floppy image when program runs
3. Modify bitmap value whenever a file is allocated or de-allocated
4. Read bitmap and output values needed(see project spec, I'll add it in later) when command is used

3.4 cat *x*

Displays the contents of a specified file x(relative or absolute path) which is given as a single argument. An error message will be displayed in the following circumstances:

- There are 0 or more than 1 argument
- The file specified does not exist
- The path provided is a directory

1. Argument number validation and parsing
2. Argument path validation
3. Traverse through sectors(directories) and search for matching file
4. If file is empty, free memory
5. Otherwise print out contents of file [void readFile(char* x)]

3.5 cd *x*

Changes directory to argument specified. If there are no arguments, the user is brought to the home directory. This function will update the global PATH variable

1. Argument path validation
2. Check if entered path starts with the root
3. If so start at root
4. If not, start at current directory
5. Traverse global PATH variable until
6.
7. If the command runs without error, change the current path variable to the path you navigated to

3.6 ls *x*

Lists the files and directories in the current working directory. If x is the name of a file, the file name, file type, FLC and size will be displayed along with the current and parent directory entries. If x is a directory, list all entries inside along with the current and parent directory entries.

1. Get the current working directory
2. Search fat table for all sectors connected to that directory
3. Display name, type, size and other info for each item in those sectors

3.7 mkdir *x*

Makes a directory. If x already exists, print a message. x may be either an absolute or relative path. An error message will display if there is not exactly one argument.

1. Find free entry in the target directory
2. If the directory is full and space exists on the disk, allocate another sector to the target directory.
3. If there is insufficient space on disk, print an appropriate message and quit the operation
4. Look for space on the disk for the file.
5. If there is insufficient space, quit the operation with an appropriate error message
6. If the right number of free sectors is found, allocate them to the directory and update the required data structures
7. In the target directory, add the entry for x along with its name and FLC

3.8 pwd

Prints the current working directory.

3.9 rm *x*

Command used to remove files. Implementation consists of removal, parent directory optimization, freeing the space used by the removed file, and data structure update. This command does not handle the following:

- 0 arguments
- The removal of directories
- Removal of batch files

1. Validate argument
2. Traverse through sectors and find file location through FAT
3. Remove file [void removeFile(??)]s

3.10 rmdir *x*

Removes the specified directory. Will have to remove all the files in the directory first. Runs RM for each file in directory. Does not handle the following:

- The removal of files
- 0 arguments

- Removal of batch files

1. Validate argument
2. Traverse through sectors and find directory through FAT
3. Check if empty [bool checkFile(??)]
4. If empty remove directory [removeDirectory()]

3.11 touch *x*

Creates a file x if it does not already exist. The argument can be a relative or absolute path name, and if the value already exists a message indicating such will be displayed.

1. Validate argument
2. Find free entry in the target directory
3. If the directory is full and space exists on the disk, allocate another sector to the target directory.
4. If there is insufficient space on disk, print an appropriate message and quit the operation
5. Look for space on the disk for the file.
6. If there is insufficient space, quit the operation with an appropriate error message
7. If the right number of free sectors is found, allocate them to the file and update the required data structures
8. In the target directory, add the entry for x along with its file name and extension, type, size and the FLC

4 Test Plan Overview

Actual Implementation still needed

A general idea of how to test the implementation of the system using bash script. Testing will be implemented incrementally so as to avoid any waterfall+log debugging. The following outlines the steps we will be taking to ensure that the shell is functioning as specified.

- Note: user input is handled by argc, **argv in respective c files
- Use switch statements in main shell which calls the c programs that handles each respective command [?]
- Use floppy images provided in code for testing
- Follow project milestones in syllabus
- Create bash script with basic commands

- Test each command with different number of arguments & incorrect commands
- Incorrect command should error in general
- Test each command with both caps and lowercase
- Run and make sure it runs as each command is implemented
- If errors are thrown, attempt to fix each command before additions are added