Brian Team
dot4qu

CS 4414 Machine Problem 3 – FAT16 Filesystem

Pledge:

On my honor, I have neither given nor received help on this assignment.

Problem:

The main problem statement of this assignment can be taken almost exactly from the assignment: “to familiarize [us] with file system organization”. We learned the use of a FAT and how it translates to the data locations for single and multi-clustered files. We also needed to understand directory structure each entry’s attributes and other information to traverse the filesystem. The goal was to be able to both read and write into a given FAT16 filesystem, utilizing things like the boot sector, File Allocation Table, root directory, and data section. We needed to be able to handle file sizes utilizing both single and multiple clusters within the filesystem. I was able to complete the assignment successfully according to the specification and my own given test cases.

Approach:

The very first step was to read through the FAT16 whitepaper given and other online resources to understand the layout of the system itself. Afterwards, to simplify the reading and manipulation of the data that would be used often, two structs were found online and used. The first was for the boot sector of the filesystem while the second was for a single entry. By utilizing these, the bytes were abstracted into named members for easier user.

The start of the boot sector was found at byte zero, so the input filename was simply opened as a file and read into the Fat16BootSector struct. As can be seen in the fat.h code included, this contained important values like the number of bytes per sector, number of sectors per cluster, root entry count, and more.

For simplification of later file seeking, 3 offsets were calculated. The first was the start of the root directory (root_dir_start) through the following equation: (reserved_sectors + fat_size_sectors * number_of_fats) *bytes_per_sector. This is because the root directory started directly after the boot sector, the rest of the reserved sectors, and both FATs. The second offset

calculated was the start of the FAT (fat_start), found by simply multiplying the bytes_per_sector times the reserved_sectors. The first FAT starts directly after the reserved sectors. The third offset calculated was the actual start of data storage for each file (data_start), found by adding the root_entry_count * 32, the size of each root entry, and the root_dir_start. The data section begins immediately after the root directory.

The 'read_dir_entries' function was then called after seeking to the start of the root directory. This is function is designed to iterate through a given directory and build up an array of type Fat16Entry with every entry contained. It stopped when it encountered an entry with a filename beginning with 0x00, as specified by the whitesheet.

At this point, the program entered its main while-loop, looping forever until it received an input of 'exit' in accordance with the assignment. At every iteration, it called 'parse_input' which took in the user's input string and delimited it by spaces into a vector of strings. This enabled the later code to pull pieces out, such as the command (at the zeroth index), or file paths at later indices. The rest of the main function was devoted to a large if-else statement checking the command string against each the four possible commands.

If an 'ls' command was received, it checked to see if there was a second entry in the input vector; if not, then the current directory needed to be listed (the current directory was saved and updated each iteration within the cwd variable). Two functions were used for this behavior. The first, 'find_dir', was widely used for all of the commands. Beginning with the root directory, it took the entire file path and called 'find_dir_entries' on the current directory. It then searched through all entries returned for the name of the next child directory. Repeating until the last directory of the string, it ended with an array of entries for the directory passed in. For 'ls', the next step was fairly simple, and involved a 'print_directories' function which cycled through the entries and printed them in the correct format.

The 'cd' command was slightly more complicated, as it involved changing the cwd variable, but still mimicked 'ls' code closely. It first checked for entries of '.' and '..', expanding those into the current directory and parent directory respectively. Using the same logic as 'ls', 'find_dir' was then called with the pathname of the directory that the user was trying to transfer to. Once the entries for this directory were found, the cwd variable needed to be set to the new directory. This involved some small logic depending on whether the input directory was an absolute or relative path.

The final two commands, ‘cpin’ and ‘cpout’ involved much more logic to implement. I began with ‘cpout’ as it didn’t involve altering the given filesystem, only the host. Note: for the remainder of this writeup, any file referred to as ‘local’ corresponds to the FAT16 filesystem, and any file referred to as ‘host’ is one on the “real” unified filesystem.

The ‘cpout’ command began by separating the local filename from the file path itself through small substringing and helper functions such as ‘remove_ext’. After calling ‘find_dir’ on the file path to fill the entries array, the filename was compared with each entry until found. The ‘copy_file_out’ function was then executed to perform the task of copying the actual file data onto the host filesystem. Within a do-while loop, ‘fseek’ was called to position the file pointer at the start of the file’s data by computing the equation “data_start + (fat_offset - 2) * sectors_per_cluster * bytes_per_sector”, where the fat_offset is the location of that file’s entry in the FAT. There is a subtraction of two because the FAT has two reserved entries at its beginning while the data section starts cluster two right at the beginning. An if-statement checked the remaining size of the file left to be copied and, if greater than 4096, the full cluster amount of 4096 was copied into a buffer, and then written onto the host file system. If there was less than 4096 bytes of data remaining to be copied than the smaller amount of remaining_filesize was buffered and written to the host. Before the loop ended, an ‘fseek’ to the current FAT entry was executed and the value was saved into fat_offset. If this equaled 0xFFFF, the loop exited because that code corresponds to no more data clusters for a file. If not, the loop reiterated with the new fat_offset used to find the next data cluster with the above equation. The reason a do-while loop is used is for single cluster file, there needs to be a single execution and copy before it exits with a FAT entry of 0xFFFF. Once the ‘copy_file_out’ function exits, the desired file has been copied from local to host.

The fourth and final command, ‘cpout’, was the most difficult to implement. Multiple separate functions were used outside of the main method, which was only responsible for parsing the local and host file paths and passing them to the ‘copy_file_in’ function. This function first read the file from the host filesystem to determine the filesize. Using the same logic as ‘cpin’, it then separated the filename and file path, calling ‘find_dir’ on the filepath to build the array of entries. When the ‘find_dir’ function returned, the file pointer remained pointed directly after the final entry of the directory, since they had all just been read. Utilizing this fact, a new Fat16Entry was created, populated with the ‘build_entry’ function, and then written into the end of the

directory at the file pointers location. The ‘build_entry’ function should not be overlooked however. It was responsible for parsing and filling in every field of the Fat16Entry struct, including things like the filename, extension, attributes, and especially the starting cluster. Another separate function was written to find the first open FAT entry, which was the FAT index closes to zero holding a value of 0x0000. This function pointed the file pointer at the start of the FAT and iterated over every entry until the first empty match. This function was used to assign the first open FAT index to the starting_cluster value of the new entry. The filesize was the final entry member saved, and the function returned back to ‘copy_file_in’. The final piece of this function was to write the actual data from the host file into the local FAT16 filesystem. The ‘write_data’ function took as parameters the newly created entry, the local disk file to write to, and the host filename to copy. The meat of the function was an if-else statement which checked the file’s size. If less than 4096, than a single cluster copy could be used. If greater than 4096, than multiple clusters were needed. For a single cluster copy, the file data was written to the data section at the cluster value held in the entry’s starting_cluster variable. Then the 0xFFFF value signifying no more clusters was written into that index of the FAT. For a multi-cluster copy, a while-loop iterated, decreasing the remaining_filesize by 4096 each time. Every iteration, it copied 4096 bytes into an open cluster, found the next open FAT entry, rewrote the previous FAT entry with the location of the new entry, and wrote 0xFFFF to the new entry. This ensured that for every data cluster that was being written, it would have a corresponding FAT entry that held the next FAT index to access for the next cluster of data. Finally, when the remaining_filesize fell below 4096, the remaining data was copied to the last cluster, the FAT entry was set to 0xFFFF, and the function exited.

Results:

Upon completion of the program, it ran successfully for all test cases given. These included:

- ls with no directory specified
- ls of absolute and relative paths
- cd of absolute and relative paths
- cd of ‘..’ translated to parent directory
- cd of ‘.’ translated to current directory
- cpout single cluster file with absolute and relative paths

- cpout multi cluster file with absolute and relative paths
- cpin single cluster file with absolute and relative paths
- cpin multi cluster file with absolute and relative paths

Each file that was copied in and then copied out was checked for consistency by diff-ing the two files. During testing a hex editor was also used to view the raw bytes of the FAT and the data section to ensure correct values and locations were being written. It was given that no error checking was necessary for valid file paths or commands, so this functionality was omitted.

Analysis:

There was not a large amount of runtime analysis that needed to be run on this program, as the only significant loops that were executing were the main loop repeatedly executing user commands, the loops for copying files larger than 4096 bytes, and the loop that checked the FAT for the next open entry. Worst case scenario for the copying loops is that they would run many times for a single massive file that took up an entire empty filesystem. Worst case for the FAT lookup is that it would iterate through the entire table for a single open entry at the last index. This could be mitigated by saving the last known open spot and beginning the search from there each time, looping to the beginning of the FAT if the end was reached.

In terms of space complexity, this program was not extremely efficient since the program functionality requirements were more important than space. There were several large arrays declared, such as the one to hold each entry when looking through a directory or the 4096-byte char arrays for copying data, but all dynamically allocated memory was released when no longer needed.

Conclusion:

The purpose of this assignment was to teach the understanding of the FAT16 layout and the process of reading and writing to it. It was necessary to really read and understand the whitepaper before even beginning to write code. Implementing the ‘ls’ and ‘cd’ commands taught the ability to traverse through directories and file attributes, while the copy commands provided the understanding of the translation between FAT entries and data clusters. These skills are extremely important when dealing with operating systems and their control of data storage.

fat.h

```
/*
Written by Brian Team (dot4qu)
Date: 11/21/16
This header file is responsible for listing the structs,
macros, included files, and function prototypes for fat.cpp
*/

#include <stdlib.h>
#include <stdio.h>
#include <string>
#include <iostream>
#include <vector>
#include <cstring>
#include <cstring>
#include <cerrno>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctime>

using namespace std;

#define BOOT_SECTOR_SIZE 512

typedef struct {
   unsigned char jmp[3];
   char oem[8];
   unsigned short bytes_per_sector;
   unsigned char sectors_per_cluster;
   unsigned short reserved_sectors;
   unsigned char number_of_fats;
   unsigned short root_entry_count;
   unsigned short total_sectors_short; // if zero, later field is used
   unsigned char media_descriptor;
   unsigned short fat_size_sectors;
   unsigned short sectors_per_track;
   unsigned short number_of_heads;
   unsigned int hidden_sectors;
   unsigned int total_sectors_long;

   unsigned char drive_number;
   unsigned char current_head;
   unsigned char boot_signature;
   unsigned int volume_id;
   char volume_label[11];
   char fs_type[8];
   char boot_code[448];
   unsigned short boot_sector_signature;
} __attribute((packed)) Fat16BootSector;
```

```
typedef struct {
   unsigned char filename[8];
   unsigned char ext[3];
   unsigned char attributes;
   unsigned char reserved[10];
   unsigned short modify_time;
   unsigned short modify_date;
   unsigned short starting_cluster;
   unsigned int file_size;
} __attribute((packed)) Fat16Entry;

FILE* read_boot_sector(Fat16BootSector *bs);                                    /* helper
function to read in the bootsector into the Fat16BootSector struct and return a file pointer to the
filesystem .raw file */
void fix_filename(unsigned char *buf, unsigned char *new_buf);          /* strips trailing spaces off of
filenames less than 8 characters */
void fix_extension(unsigned char *buf, unsigned char *new_buf);                 /* strips trailing spaces
off of file extensions less than 3 chars */
int read_dir_entries(string filename, Fat16Entry **entries_arr, FILE *in);/* takes in a dir name and array
of entries and iterates along them building array for searching through */
string parse_input(vector<string> &out);                                        /* reads in users
input string, parses into a vector of terms, and returns the command given */
void print_dir(int num_entries, Fat16Entry **entries);                          /* helper function to
iterate through dir entries printing in the correct format */
int find_dir(string dir, Fat16Entry **entries, FILE *file);             /* takes absolute or relative
filepath and iterates searching through each dir for the next to get the final set of entries */
string remove_ext(string filename);
        /* strips ext of of input filename */
void copy_file_out(Fat16Entry *entry, string file_out, FILE *in);/* major workhorse function to
repeatedly copy filedata out until final cluster reached */
unsigned short find_open_fat_entry(FILE *in);                                   /* iterates
through fat until first open cluster is found and returns index */
void build_entry(Fat16Entry *new_entry, string filename, int file_size, FILE *in);          /* builds up
entry information when cpin'ing a file */
void write_data(Fat16Entry *entry, FILE *in, FILE *source);                     /* writes actual file data
to data section while iterating through fat values */
void copy_file_in(string host_read_str, string local_write_str, string cwd, FILE *in, Fat16Entry
**entries);       /* major workhorse function to handle setting up entry and writing data for a cpin */
```

fat.cpp

```
/*
Written by Brian Team (dot4qu)
Date: 11/21/16
This C++ source file implements a barebones FAT16 filesystem controller.
It supports the cd, ls, cpin, and cpout commands.
*/

#include "fat.h"

//GLOBALS
int root_dir_start;                                     /* holds the offset within the filesystem
file for the root directory */
int fat_start;                                          /* holds the offset within the filesystem
file for the FAT */
int data_start;                                         /* holds the offset within the filesystem
file for the data section */
unsigned char sectors_per_cluster;              /* the number of sectors per cluster */
unsigned short bytes_per_sector;                /* the number of bytes per sector */

FILE* read_boot_sector(Fat16BootSector *bs, char *data_file) {
        FILE *raw = fopen(data_file, "rb+");
        fread(bs, sizeof(*bs), 1, raw);
        return raw;
}

void fix_filename(unsigned char *buf, unsigned char *new_buf) {
        int i = 0;
        //iterate along filename copying all letters into new buf. Needs index check b/c if filename 8
bytes, keeps going since buf is passed as pointer
        while (buf[i] != ' ' && i < 8) {
                new_buf[i] = buf[i];
                i++;
        }
        //replace first space with null terminator
        new_buf[i] = '\0';
}

void fix_extension(unsigned char *buf, unsigned char *new_buf) {
        int i = 0;
        //iterate along filename copying all letters into new buf. Needs index check b/c if filename 8
bytes, keeps going since buf is passed as pointer
        while (buf[i] != ' ' && i < 3) {
                new_buf[i] = buf[i];
                i++;
        }
        //replace first space with null terminator
        new_buf[i] = '\0';
}
```

```
int read_dir_entries(string filename, Fat16Entry **entries_arr, FILE *in) {
	int i = 0;
	unsigned char name_buf[9];
	int temp;

	if (filename == "/") {
		//root dir case
		fseek(in, root_dir_start, SEEK_SET);
		 temp = ftell(in);
	} else {
		while (entries_arr[i] != NULL) {
			fix_filename(entries_arr[i]->filename, name_buf);
			if (!strcmp(filename.c_str(), (char*) name_buf)) {
				//this is the entry of the dir we're looking at, need to go to it on disk
				fseek(in, data_start + (entries_arr[i]->starting_cluster - 2) *
sectors_per_cluster * bytes_per_sector, SEEK_SET);
				temp = ftell(in);
				break;
			}
			i++;
		}
	}

	i = 0;
	while(true) {
		Fat16Entry *new_entry = new Fat16Entry;
		fread(new_entry, sizeof(Fat16Entry), 1, in);

		//first byte of filename is zero, that means we're done reading files for this directory
		if (new_entry->filename[0] == 0x00) {
			//just read in a blank entry, need to rewind 32 before returning for copy_file_in's
use of filepointer for new entry addition
			fseek(in, -32, SEEK_CUR);
			break;
		}
		//checking for long filename attr. If it is, toss this entry , don't need it
		if (new_entry->attributes == 0x0F)
			continue;
		//set current index of array to address of new_entry (since the variable is a pointer)
		entries_arr[i] = new_entry;
		//increment index in entry array
		i++;
	}

	return i;
}

string parse_input(vector<string> &out) {
	int end = 0;
	int start = 0;
	int arr_idx = 0;
```

```
	string ret_dir;
	string input_line;

	out.clear();

	//read in user input line
	getline(cin, input_line);

	while (input_line[end] != '\0') {
		if (input_line[end] == ' ') {
			out.push_back(input_line.substr(start, end - start));
			end++;
			start = end;
			continue;
		}
		end++;
	}
	//push back last dir b/c while loop kicks out at \0 w/o printing
	out.push_back(input_line.substr(start, end - start));
	end++;

	if (out.size() <= 1) {
		ret_dir = "";
	} else if (out.at(0) == "ls") {
		ret_dir = out.at(1);
	} else if (out.at(0) == "cd") {
		ret_dir = out.at(1);
	} else if (out.at(0) == "cpin") {
		//for copy in, we return the local filepath we want
		return out.at(2);
	} else if (out.at(0) == "cpout") {
		//for copy out, we return the local filepath to copy out
		return out.at(1);
	}

	return ret_dir;
}

void print_dir(int num_entries, Fat16Entry **entries) {
	int i = 0;
	unsigned char name_buf[9];
	unsigned char ext_buf[4];

	for (i = 0; i < num_entries; i++) {
		//strips unneccesary trailing spaces if filename not 8 chars
		fix_filename(entries[i]->filename, name_buf);

		if (entries[i]->attributes & (1 << 4)) {
			//directory
			printf("D %s\n", name_buf);
		} else if (entries[i]->attributes & (1 << 3) || entries[i]->attributes & (1 << 1)) {
```

```
                        //volume label or hidden file, dont print
                } else {
                        //file
                        fix_extension(entries[i]->ext, ext_buf);
                        printf("F %s.%s\n", name_buf, ext_buf);
                }
        }
}

int find_dir(string dir, Fat16Entry **entries, FILE *file) {
        int start = 0;
        int end = 0;
        int i = 0;
        int num_entries = 0;
        string temp_dir;
        unsigned char name_buf[9];

        while(true) {
                if (dir[end] == '/') {
                        //just pulled in another dirname, need to read its entries into array
                        if (start == end)
                                //case of pulling in first slash of absolute path
                                temp_dir = dir[start];
                        else
                                temp_dir = dir.substr(start, end - start);
                        //update our entries array for this dir
                        num_entries = read_dir_entries(temp_dir, entries, file);
                        end++;
                        start = end;
                } else if (dir[end] == '\0') {
                        if (start != end) {
                                //update our entries array for the final dir (start != end checks case of
only finding root dir) then break out
                                temp_dir = dir.substr(start, end - start);
                                num_entries = read_dir_entries(temp_dir, entries, file);
                        }
                        break;
                } else {
                        //still moving through dirname
                        end++;
                }
        }
        return num_entries;
}

string remove_ext(string filename) {
        int i = filename.length() - 1;
        while (filename[i] != '.') {
                i--;
        }
        return filename.substr(0, i);
```

```
}

void copy_file_out(Fat16Entry *entry, string file_out, FILE *in) {
	//holds two byte value pulled in from current fat location
	unsigned char fat_value[2] = {0, 0};
	const short fat_eof = 0xFFFF;
	//hold the number representation of current fat index used for seeking that much from the fat_start
	short fat_offset = entry->starting_cluster;

	//open file on path within host
	FILE *out = fopen(file_out.c_str(), "wb");
	chmod(file_out.c_str(), 0777);
	//printf("Error: %s\n", strerror(errno));

	unsigned int remaining_filesize = entry->file_size;
	//declare buffer to hold read bytes before we write them
	char *filebuf = new char[4096];
	//seek and read in first value of this file within fat. offset by starting cluster from fat_start
	do {
		//seek to beginning of files actual data
		fseek(in, data_start + (fat_offset - 2) * sectors_per_cluster * bytes_per_sector,
SEEK_SET);
		int temp = ftell(in);
		if (remaining_filesize >= 4096) {
			fread(filebuf, 4096, 1, in);
			fwrite(filebuf, 4096, 1, out);
			remaining_filesize -=4096;
		} else {
			fread(filebuf, remaining_filesize, 1, in);
			fwrite(filebuf, remaining_filesize, 1, out);
			remaining_filesize = 0;
		}

		//reading current entries fat value for end of do-while check
		fseek(in, fat_start + fat_offset * 2, SEEK_SET);
		temp = ftell(in);
		//updates fat_offset with value at current index. either FFFF or will point to next cluster
to read data from for re-iteration of this loop
		fread(&fat_offset, sizeof(fat_offset), 1, in);

		//fat_offset = fat_value;
	} while (fat_offset != fat_eof)/*while (fat_value[0] != fat_eof[0] && fat_value[1] !=
fat_eof[1])*/;

	delete[] filebuf;
	fclose(out);
}

unsigned short find_open_fat_entry(FILE *in) {
	int previous_location = ftell(in);				/* save previous fpointer location */
	unsigned short fat_cluster = 1;					/* holds the index of the fatval
```

```
we're currently reading in */
	short fat_val;                                                          /* hold the
values that we're iterating through in the fat until we find the first open (0x0000) entry */

	//skip over first two reserved fat entries to read first possible, sector 2
	fseek(in, fat_start + 4, SEEK_SET);
	do {
		fat_cluster++;
		fread(&fat_val, sizeof(short), 1, in);
	} while (fat_val != 0x0000);

	//move fpointer back to where it was when we started
	fseek(in, previous_location, SEEK_SET);

	return fat_cluster;
}

void build_entry(Fat16Entry *new_entry, string filename, int file_size, FILE *in) {
		int new_entry_location = ftell(in);                             /* need to save where
were saving entry because we need to seek to the fat to find first open cluster */

	//strip ext
	string filename_no_ext = remove_ext(filename);
	int padding = 0;
	//copies each char of filename into new entry, then pads remaining space for 8 byte array with
spaces if needed
	for (padding = 0; padding < filename_no_ext.length(); padding++) {
		new_entry->filename[padding] = filename_no_ext[padding];
	}
	for ( ; padding < 8; padding++) {
		new_entry->filename[padding] = ' ';
	}

	//iterate from end until hit period, then strip only the following text for ext
	int ext_idx = filename.length() - 1;
	while (filename[ext_idx] != '.') {
		ext_idx--;
	}
	string ext = filename.substr(ext_idx + 1);
	for (padding = 0; padding < ext.length(); padding++) {
		new_entry->ext[padding] = ext[padding];
	}
	for ( ; padding < 3; padding++) {
		new_entry->ext[padding] = ' ';
	}

	//hardcoded file (archive) attrs
	new_entry->attributes = 0x20;

	unsigned char reserved_bytes[10];
	*new_entry->reserved = *reserved_bytes;
```

```
	time_t curr_time = time(0);
	//hardcoded, not necessary
	new_entry->modify_time = 0;
	new_entry->modify_date = 0;

	//returns index of next available fat entry
	new_entry->starting_cluster = find_open_fat_entry(in);

	//rewind fpointer to where we're going to save this entry
	fseek(in, new_entry_location, SEEK_SET);

	new_entry->file_size = file_size;
}

void write_data(Fat16Entry *entry, FILE *in, FILE *source) {
	const short fat_eoc = 0xFFFF;
	//temporary filebuffer to read data into
	char filebuf[entry->file_size];
	fseek(in, fat_start + entry->starting_cluster * 2, SEEK_SET);
	fwrite(&fat_eoc, sizeof(short), 1, in);

	//seek to beginning of files actual data
	fseek(in, data_start + (entry->starting_cluster - 2) * sectors_per_cluster * bytes_per_sector,
SEEK_SET);
	int data_location = ftell(in);				/* saves location of data for multicluster
writes when we seek to fat to write in current clusters value at previous clusters location */

	if (entry->file_size <= 4096) {
		//single cluster data write
		//pulling in file data from host disk
		fread(filebuf, entry->file_size, 1, source);
		fwrite(filebuf, entry->file_size, 1, in);
	} else {
		//multi cluster data write
		int remaining_filesize = entry->file_size;
		unsigned short previous_fat_cluster = entry->starting_cluster;
		unsigned short current_fat_cluster;

		while(remaining_filesize > 0) {
			if (remaining_filesize <= 4096) {
				//copy only remaining filesize
				//pulling in file data from host disk
				fread(filebuf, remaining_filesize, 1, source);
				fwrite(filebuf, remaining_filesize, 1, in);
				//write EOC value into last fat slot
				fseek(in, fat_start + previous_fat_cluster * 2, SEEK_SET);
				fwrite(&fat_eoc, sizeof(short), 1, in);
				remaining_filesize = 0;
			} else {
				//pulling in file data from host disk
```

```
                                        fread(filebuf, 4096, 1, source);
                                        //copy full allotment of 4096
                                        fwrite(filebuf, 4096, 1, in);
                                        data_location = ftell(in);
                                        //find next open fat spot and save as 'current'
                                        current_fat_cluster = find_open_fat_entry(in);
                                        //seek to location of previous fat spot
                                        fseek(in, fat_start + previous_fat_cluster * 2, SEEK_SET);
                                        //write value of newly saved current into previous' slot for the next hop
                                        fwrite(&current_fat_cluster, sizeof(short), 1, in);
                                        //now seek to current and write EOC value so our next iterations call to
find_open_fat_entry doesnt grab the same one
                                        fseek(in, fat_start + current_fat_cluster * 2, SEEK_SET);
                                        fwrite(&fat_eoc, sizeof(short), 1, in);
                                        //set current as the previous cluster for next iteration
                                        previous_fat_cluster = current_fat_cluster;
                                        //seek back to proper data writing location
                                        fseek(in, data_location, SEEK_SET);
                                        //reduce remaining filesize
                                        remaining_filesize -= 4096;
                                }
                        }
                }
}

void copy_file_in(string host_read_str, string local_write_str, string cwd, FILE *in, Fat16Entry **entries)
{
        FILE *host_read = fopen(host_read_str.c_str(), "rb");
        int file_size;
        int num_entries = 0;
        int entries_end_addr;

        //seek to file end to calculate filesize
        fseek(host_read, 0, SEEK_END);
        file_size = ftell(host_read);
        //seek back to beginnging for following fread of data
        fseek(host_read, 0, SEEK_SET);

        int index = local_write_str.length() - 1;
        string filename, filepath;
        //move index backwards from end until it hits space or /.
        //everything from [0] to index is filepath, everything from index to end is filename
        while (local_write_str[index] != '/' && index != 0) {
                index--;
        }
        if (index != 0) {
                //on a slash somewhere in the filename most likely
                filename = local_write_str.substr(index + 1);
                filepath = local_write_str.substr(0, index);
        } else {
                //were at zero index
```

```
                if (local_write_str[index] == '/') {
                        //if were on a slash, substr from index + 1
                        filename = local_write_str.substr(index + 1);
                } else {
                        //not on a slas hbut still at index 0, so single filename of cwd given
                        filename = local_write_str.substr(index);
                }
                filepath = cwd;
        }

        num_entries = find_dir(filepath, entries, in);
        //find dir calls read_dir_entries, which will end up reading the entries for our final dir (end of dir
string)
        //this leaves the file pointer point at the null bytes directly after the last entry has been read
        entries_end_addr = ftell(in);

        Fat16Entry *new_entry = new Fat16Entry();
        build_entry(new_entry, filename, file_size, in);
        //writes new entry directly following final entry read since the fpointer is still sitting there from
'find_dir'
        fwrite(new_entry, sizeof(Fat16Entry), 1, in);

        //handles writing data to correct clusters and fat entries
        write_data(new_entry, in, host_read);
        fflush(in);

        entries_end_addr = ftell(in);
        delete[] new_entry;
}

int main(int argc, char **argv) {
        int curr_file_pos = 0;
        int num_entries = 0;
        int i = 0;
        string cwd, temp_cwd, local_copy, host_copy;
        vector<string> input;

        //pull in data_file name as only arg given (arg[0]) holds program filanme
        char *data_file = argv[1];

        Fat16BootSector *bs = new Fat16BootSector;
        FILE *in;

        //opens file and reads in bootsector bytes located at beginning of disk
        in = read_boot_sector(bs, data_file);

        //saving global values for use in other functions w/o passing bs reference
        sectors_per_cluster = bs->sectors_per_cluster;
        bytes_per_sector = bs->bytes_per_sector;

        //calculating global values used for seeking to offsets throughout fs
```

```
	root_dir_start = BOOT_SECTOR_SIZE + (bs->reserved_sectors - 1 + bs->fat_size_sectors * bs->number_of_fats) * bs->bytes_per_sector;
	fat_start = bs->bytes_per_sector * bs->reserved_sectors;
	data_start = root_dir_start + (bs->root_entry_count * 32);
	//Pointer is at end of boot sector hence the minus one for reserved sectors
	//This seeks past FATs and to the beginning of the root directory
	fseek(in, root_dir_start, SEEK_SET);

	//allocate array for root dir entries
	Fat16Entry *entries[bs->root_entry_count];
	//sets initial directory to root
	cwd = "/";
	//reads and moves file pointer through entire root directory
	num_entries = read_dir_entries(cwd, entries, in);

	//reserve spots for 4 strings
	input.reserve(4);

	while (true) {
		printf(":%s>", &cwd[0]);
		//reads in the user input string and delimits by spaces and places each string in the input vector
		temp_cwd = parse_input(input);

		if (input.at(0) == "ls") {

			if (temp_cwd != "") {
				//only get temp_cwd entries if its not an empty string
				if (temp_cwd == ".") {
					//if '.', just reset it to cwd
					temp_cwd = cwd;
				}
				num_entries = find_dir(temp_cwd, entries, in);
			} else {
				//temp_cwd was empty so we just get it for ourselves (single ls command w/ no path)
				num_entries = find_dir(cwd, entries, in);
			}
			print_dir(num_entries, entries);

		} else if (input.at(0) == "cd") {

			if (temp_cwd != "") {
				//only get temp_cwd entries if its not an empty string
				if (temp_cwd == ".") {
					//if '.', just reset it to cwd
					temp_cwd = cwd;
				} else if (temp_cwd == "..") {
					//need to substring cwd to second to last dir
					int index = cwd.length() - 1;
					//move index backwards from end until it hits space or /.
```

```
                                        //everything from [0] to index is new cwd
                                        while (cwd[index] != '/' && index != 0) {
                                                //c--;
                                                index--;
                                        }
                                        if (index == 0)
                                                temp_cwd = "/";
                                        else
                                                temp_cwd = cwd.substr(0, index);
                                }

                                num_entries = find_dir(temp_cwd, entries, in);
                        } else {
                                //temp_cwd was empty so we just get it for ourselves (single ls command
w/ no path)
                                num_entries = find_dir(cwd, entries, in);
                        }

                        if (temp_cwd[0] != '/') {
                                //relative path, append to cwd
                                if (cwd == "/") {
                                        cwd += temp_cwd;
                                } else {
                                        cwd += "/" + temp_cwd;
                                }
                        } else {
                                //absolute path, replace it
                                cwd = temp_cwd;
                        }

                } else if (input.at(0) == "cpin") {
                        //string holding path of file within open fs
                        local_copy = input.at(2);
                        //string holding desired copy-to path in host fs
                        host_copy = input.at(1);

                        //function handler for all single or multi cluster data in copying
                        copy_file_in(host_copy, local_copy, cwd, in, entries);

                } else if (input.at(0) == "cpout") {
                        if (input.size() < 3) {
                                exit(-1);
                        }
                        //string holding path of file within open fs
                        local_copy = input.at(1);
                        //string holding desired copy-to path in host fs
                        host_copy = input.at(2);
                        //temp_cwd holding path to local file to copy out
                        //need to substring temp_cwd to seperate filename from path to dir holding file
                        int index = local_copy.length() - 1;
                        string filename, filepath;
```

```
                //move index backwards from end until it hits space or /.
                //everything from [0] to index is filepath, everything from index to end is
filename
                while (local_copy[index] != '/' && index != 0) {
                        index--;
                }

                if (index != 0) {
                        //on a slash somewhere in the filename
                        filename = local_copy.substr(index + 1);
                        filepath = local_copy.substr(0, index);
                } else {
                        //were at zero index
                        if (local_copy[index] == '/') {
                                //if were on a slash, substr from index + 1
                                filename = local_copy.substr(index + 1);
                        } else {
                                //not on a slash but still at index 0, so single filename of cwd
given
                                filename = local_copy.substr(index);
                        }
                        filepath = cwd;
                }

                //strip ext b/c comparision with file in entries only does filename w/o ext
                string filename_no_ext = remove_ext(filename);
                //need to call find_dir on filepath to load in entries array with file info
                num_entries = find_dir(filepath, entries, in);

                unsigned char name_buf[9];
                int i = 0;
                int temp;

                while (i < num_entries) {
                        fix_filename(entries[i]->filename, name_buf);
                        if (!strcmp(filename_no_ext.c_str(), (char*) name_buf)) {
                                break;
                        }
                        i++;
                }
                //copies file while checking and moving fat entries until FFFF is reached
                copy_file_out(entries[i], host_copy, in);

        } else if (input.at(0) == "exit") {
                if (in != NULL) {
                        fclose(in);
                }
                return 0;
        } else {
                //not a recognized command, don't print anything
                //printf("Unrecognized command\n");
```

```
                }

        }

        return 0;
}
```

makefile

```
# Written by Brian Team (dot4qu)
# Date: 11/21/16
# This makefile is responsible for compiling and linking HW3, the FAT16 filesystem

CC=g++
DEPS=fat.cpp fat.h
OBJS=fat.o

%.o: %.cpp
	$(CC) -c -o $@ $<

fat: $(OBJS)
	$(CC) -o $@ $^

clean:
	@rm -f *.o fat
```