CS 4414: Operating Systems

Machine Problem 3: File Management

Michael Eller

22 November 2016

I was successful in implementing all functions on the FAT 16 file system. For clarity the commands that the user can enter into my program are colored in ***orange***, and all functions that are implemented in the underlying C++ code are colored in ***cyan***.

Introduction

The File Allocation Table (FAT) is a computer file system architecture that organizes and manages data clusters. The FAT system also displays the file structure in a human-readable format. The objective of this project was to explore the FAT 16 file architecture and implement a few select functions: ***cd*** (change directory), ***ls*** (list segments/files), ***cpout*** (copy out), and ***cpin*** (copy in).

Implementation

The program operates as a simple shell that can traverse and manipulate a FAT 16 file system. The following functions are implemented in the C++ programming language, and all paths are relative to the user's present working directory. The program has one command-line parameter: the path to a FAT 16 raw data file.

cd <directory>

The ***cd*** command is implemented with the combination of ***change_directory***() and ***follow_path***() functions. The ***change_directory***() function takes in one string as a parameter, and searches the user's current directory (all directory entries of the current cluster) and attempts to match the file name, which is the first eight bytes of each entry, to the string parameter. If a match is found, the function loads the first sector of the matched directory's starting cluster (bytes 26 and 27) into the user's current directory, which is a global variable. Since the ***change_directory***() function can only change directories by one folder at a time, the ***follow_path***() function is used in order to change multiple directories with one function call. It simply takes in a list of strings with all the directories and calls ***change_directory***()

on each of them. All functions implemented in this program use ***follow_path***() in some manner.

ls [directory]

If no argument is given, ***ls*** displays all of the files and folders in the user's present working directory. Otherwise, the function lists all files and folders from the path given by the user. After saving the user's current location in a temporary variable, ***follow_path***() is called in order to load the required directory. In the same manner that ***change_directory***() loops through to find a matching filename, ***list_directory***() loops through the appropriate directory and prints every file name that is not a long file name or the volume ID. The function then loads the starting location back into the user's current working directory.

cpout <src> <dest>

Between ***cpout*** and ***cpin***, I found that ***cpout*** is far easier to implement. The function begins by calling ***follow_path***() to the source file's directory. Then the FAT is consulted to check if the file is larger than a single cluster. As long as the active cluster is not the last cluster in the chain, the function continually reads source file data into a cluster-sized buffer and writes the data to the destination file on the computer's file system. The function also keeps track of the remaining bytes in the source file in order to only write the remaining bytes stored in the last cluster of the chain.

cpin <src> <dest>

The ***cpin***() function is very similar to ***cpout***(); however, instead of reading from the FAT, ***cpin***() finds the first open cluster in the FAT and starts writing source data there. If the file is larger than one cluster, ***cpin***() repeats this process while writing each successive cluster number to the index of the FAT that corresponds to the previous cluster in the chain. Additionally, ***cpin***() builds and writes the directory entry in the path specified by the destination argument.

Problems Encountered

In general I found that I understood the homework conceptually, but was slowed down by little issues. For example, I created a text file on my computer using the *touch* terminal command. Apparently the filename can appear as all uppercase, but the actual name required to open it is lowercase. This was extremely avoidable and easy to fix but slowed down my productivity since it took approximately 30 minutes to find. Besides small issues like that, the hardest part of this project was maintaining context. I used a global variable to hold the user's current location, and I had frequent issues using ***lseek***() and returning to the correct location. The GDB debugger was extremely valuable in chasing down these issues.
NOTE: I assumed that a directory was not more than the size of one cluster, that directories did not need to be in alphabetical order, and that files can exist with same name.

Testing and Analysis

I primarily used a hex editor to verify any changes that I made to the sample disk file. Once I verified single level ***cd*** and ***ls*** I moved on to multiple level testing in both directions (forward vs. '..'). The ***cpout***() function was easy to verify since the result is a file on the user's computer. I used ***cpout***() to to verify ***cpin***() by copying out the same file I copied in. Similarly single and multiple level directory paths were tested as well as single and multiple cluster sized files.

Conclusion

My implementation of the FAT 16 file manager is fully functional as specified by the problem description. I would like to acknowledge that I used the data structures provided by Professor Grimshaw and a significant amount of information gathered from this wiki.

Code

makefile

```
## Michael Eller mbe9a
## OS Machine Problem 3
## 22 November 2016
## makefile

OBJS = file_manager.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

fat: $(OBJS)
        $(CC) $(LFLAGS) $(OBJS) -o fat

file_manager.o: file_manager.h file_manager.cpp
        $(CC) $(CFLAGS) file_manager.cpp

clean:
        \rm *.o *~ fat

tar:
        tar cfv mbe9a.tar fat file_manager.h file_manager.cpp makefile mbe9a.pdf
```

file_manager.h

```
// Michael Eller mbe9a
// OS Machine Problem 3
// 22 November 2016
// file_manager.h (structs and function protos)

#ifndef FILE_MANAGER_H
#define FILE_MANAGER_H

// entry attribute definitions
#define READ_ONLY        0x01
#define HIDDEN                  0x02
#define SYSTEM                  0x04
#define VOLUME_ID         0x08
#define DIRECTORY         0x10
#define ARCHIVE                 0x20
#define LFN                         0x0F
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string.h>
#include <list>
#include <errno.h>
using namespace std;

// struct to hold the FAT16 boot sector
typedef struct {
    unsigned char jmp[3];
    char oem[8];
    unsigned short sector_size;
    unsigned char sectors_per_cluster;
    unsigned short reserved_sectors;
    unsigned char number_of_fats;
    unsigned short root_dir_entries;
```

```
    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;

 // struct to hold a FAT16 entry
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;

// function to retrieve the FAT value of the active cluster
bool get_fat_value(unsigned short* value);

// function to return the offset of the first data sector of a given cluster
unsigned int first_sector_of_cluster(unsigned int cluster);

// read dir entries in single cluster (directory)
void list_directory(string path);

// change_directory helper function
void next_entry(unsigned int starting_sector, unsigned short offset);

// method to change the PWD string, another helper function for 'cd'
void update_pwd(unsigned int direction);

// implementation of cd command
Fat16Entry change_directory(string path, bool ls);

// helper function for main, takes in a char* and outputs a list of arguments
list<char*> build_arguments(char* token);

// helper function for main, takes in a list<char*>* and call change_directory
Fat16Entry follow_path(list<char*>* arguments, bool ls);

// helper function to test if we are in the root directory
void check_and_set_root();

// copy out function implementation
void copy_out(char* dest, list<char*>* arguments);

// helper function for copy in, returns the offset of the dir entry written by this function
// very similar to 'ls'
unsigned int write_dir_entry(Fat16Entry dir_entry, list<char*>* arguments);

// copy in function implementation, file modify data and time are neglected, attributes are assumed 0x00
void copy_in(char* name, unsigned int* src_file, list<char*>* arguments);

#endif
```

file_manager.cpp

```
// Michael Eller mbe9a
// OS Machine Problem 3
// 22 November 2016
// file_manager.cpp (main)

#include "file_manager.h"

// some globals
// holds the boot sector of the disk
Fat16BootSector BS;

// track state/position of file system
Fat16Entry CURRENT;

// maintain the current path to display for user
string PWD;

// entire volume
unsigned int DISK;

// where data starts on the disk
unsigned int first_data_sector;

// first root directory sector, root sector is treated differently
unsigned int first_root_dir_sector;

// are you in the root directory?
bool root;

// function to retrieve the FAT value at the active cluster (is it the last cluster in the chain?)
bool get_fat_value(unsigned short* value, bool first_cluster, unsigned short active_cluster)
{
        // save current location
        unsigned int location = lseek(DISK, 0, SEEK_CUR);

        unsigned char FAT_table[BS.sector_size];
        unsigned short fat_offset;
        if (first_cluster) fat_offset = CURRENT.starting_cluster * 2;
        else fat_offset = active_cluster * 2;
        unsigned short fat_sector = BS.reserved_sectors + (fat_offset / BS.sector_size);
        unsigned short ent_offset = fat_offset % BS.sector_size;

        // put the fat_sector into FAT_table
        lseek(DISK, fat_sector * BS.sector_size, SEEK_SET);
        read(DISK, &FAT_table, sizeof(FAT_table));
        *value = *(unsigned short*)&FAT_table[ent_offset];

        // return to original location
        lseek(DISK, location, SEEK_SET);

        // check if there are chained clusters
        if (*value >= 0xFFF8) return false;

        // not accounting for bad clusters here
        return true;
}
```

```
// function to return the offset of the first data sector of a given cluster
unsigned int first_sector_of_cluster(unsigned int cluster, bool file)
{
        // don't calculate if the user is in the root directory
        if (root && !file) return first_root_dir_sector;
        // else calculate from the given cluster
        return ((cluster - 2) * BS.sectors_per_cluster) + first_data_sector;
}

// helper function to get file/directory name
void get_filename(char name[])
{
        // copy the filename into the char array
        memcpy(name, &CURRENT.filename, sizeof(CURRENT.filename));

        // append the file extension if it's not a directory
        if (!(CURRENT.attributes & DIRECTORY))
        {
                for (unsigned int i = 0; i < sizeof(CURRENT.filename); i++) {
                        if (name[i] == ' ') //|| (i == sizeof(CURRENT.filename) - 1)
                        {
                                // if we are at the end of the filename, start appending at the next position
                                //if ((i == sizeof(CURRENT.filename) - 1) && (name[i] != ' ')) i++;

                                // add the dot
                                name[i] = '.';

                                // add the rest of the extension
                                for (int j = 1; j <= 3; j++) name[i + j] = CURRENT.ext[j - 1];

                                // add the null terminator
                                name[i + 4] = '\0';
                                return;
                        }
                }
        }
        // get rid of extra spaces in directory names
        else for (unsigned int i = 0; i < sizeof(CURRENT.filename); i++) if (name[i] == ' ') name[i] = '\0';
}
```

```
// read dir entries in single cluster (directory)
void list_directory()
{
        // start by reading in and saving the first sector of the active cluster
        unsigned short counter = 0;
        unsigned int first_sector = first_sector_of_cluster(CURRENT.starting_cluster, false);
        lseek(DISK, first_sector * BS.sector_size, SEEK_SET);
        read(DISK, &CURRENT, sizeof(Fat16Entry));

        // keep listing as long as the first byte != 0
        while (1)
        {
                // don't print it if it's a long file name or if it's the volume ID
                if ((CURRENT.attributes & LFN) || (CURRENT.attributes & VOLUME_ID))
                {
                        // read next entry
                        counter++;
                        lseek(DISK, (first_sector * BS.sector_size) + (counter * sizeof(Fat16Entry)), SEEK_SET);
                        read(DISK, &CURRENT, sizeof(Fat16Entry));
                        continue;
                }
                else
                {
                        // get the filename
                        // initialize with max possible size
                        char name[sizeof(CURRENT.filename) + sizeof(CURRENT.ext) + 2];
                        get_filename(name);

                        // display the name
                        cout << name << endl;

                        // read the next entry
                        counter++;
                        lseek(DISK, (first_sector * BS.sector_size) + (counter * sizeof(Fat16Entry)), SEEK_SET);
                        read(DISK, &CURRENT, sizeof(Fat16Entry));

                        // break if the first byte is 0
                        if (CURRENT.filename[0] == 0x00) break;
                }
        }
        // return to the first sector
        lseek(DISK, first_sector * BS.sector_size, SEEK_SET);
        read(DISK, &CURRENT, sizeof(Fat16Entry));
}
```

```
// change_directory helper function
void next_entry(unsigned int starting_sector, unsigned short offset)
{
        // load sector at given offset into CURRENT
        lseek(DISK, (starting_sector * BS.sector_size) + (offset * sizeof(Fat16Entry)), SEEK_SET);
        read(DISK, &CURRENT, sizeof(Fat16Entry));
}

// method to change the PWD string, another helper function for 'cd'
void update_pwd(unsigned int direction)
{
        // take one level off of PWD if the user did 'cd ..'
        if (direction == 0)
        {
                // used to index the PWD string from the end, for figuring out where to stop deleting
                unsigned short index = 0;

                // get rid of all the end characters until first '/' is reached
                for (string::reverse_iterator str_riter = PWD.rbegin() + 1; str_riter < PWD.rend(); ++str_riter)
                {
                        index++;
                        if (*str_riter == '/') break;
                }
                // delete the last directory
                PWD = PWD.substr(0, strlen(PWD.c_str()) - index);
        }
        // add name of the current directory to PWD
        else
        {
                // get directory name
                char* name = strtok((char*)CURRENT.filename, " ");

                // append it to PWD
                PWD += name;

                // formatting
                PWD += '/';
        }
}
```

```
// implementation of cd command
Fat16Entry change_directory(string path, bool ls)
{
        // get the cluster of pwd, before we start searching
        unsigned int starting_sector = first_sector_of_cluster(CURRENT.starting_cluster, false);

        // stay in current directory
        if (!strcmp(path.c_str(), ".")) next_entry(starting_sector, 0);

        // go back one directory
        else if (!strcmp(path.c_str(), ".."))
        {
                next_entry(starting_sector, 1);
                // update the PWD string if we are actually cd'ing
                if (!ls) update_pwd(0);
                // check and see if we are now looking at the root directory
                check_and_set_root();
        }
        // search for next directory
        else
        {
                // offset to iterate through directory
                unsigned int offset = 2;
                if (root) offset = 1;
                char name[sizeof(CURRENT.filename) + sizeof(CURRENT.ext) + 2];
                while (1)
                {
                        // start at the third entry in directory
                        next_entry(starting_sector, offset);
                        // stop if the name wasn't found and it's the end of the cluster
                        if (CURRENT.filename[0] == 0x00)
                        {
                                cout << "No such directory." << endl;
                                break;
                        }
                        // ignore long file names
                        if (CURRENT.attributes & LFN)
                        {
                                offset++;
                                continue;
                        }
                        // get file/directory name
                        get_filename(name);

                          // is it a directory and does it match?
                          if ((CURRENT.attributes & DIRECTORY) && (!strcmp(path.c_str(), name)))
                          {
                                  // go to the first sector of the starting cluster and update the path
                                  root = false;
                                // update the path if this wasn't used in an 'ls' command
                                  if(!ls) update_pwd(1);
                                  // continue to the first sector of the entry's starting cluster
                                  next_entry(first_sector_of_cluster(CURRENT.starting_cluster, false), 0);
                                  return CURRENT;
                          }
                          // if it's a file
                          else if ((!strcmp(path.c_str(), name)))
                          {
                                  // continue to the first sector of the entry's starting cluster
                                  return CURRENT;
                          }

                          // try the next entry
                          offset++;
                  }
        }
        return CURRENT;
}
```

```
// helper function for main, takes in a char* and outputs a list of arguments
list<char*> build_arguments(char* token)
{
        // parse first argument
        list<char*> arguments;

        // add all to a list and return it
        while (token != NULL)
                {
                        arguments.push_back(token);
                        token = strtok(NULL," /");
                }
        return arguments;
}

// helper function for main, takes in a list<char*>* and call change_directory
Fat16Entry follow_path(list<char*>* arguments, bool ls)
{
        Fat16Entry file_params;

        // follow the path by calling change_directory for each directory in path
        while (!(arguments->empty()))
        {
                file_params = change_directory(arguments->front(), ls);
                arguments->pop_front();
        }
        return file_params;
}

// helper function to test if we are in the root directory
void check_and_set_root()
{
        if (CURRENT.starting_cluster == 0) root = true;
        else root = false;
}
```

```
// copy out function implementation
void copy_out(char* dest, list<char*>* arguments)
{
        // create the file locally
        unsigned int output_file = open(dest, O_RDWR | O_CREAT | O_TRUNC, S_IRWXU);

        // find the fat entry
        Fat16Entry file_entry = follow_path(arguments, true);

        // save the file size
        unsigned int file_size = file_entry.file_size;

        // buffer to read in clusters
        unsigned int cluster_size = BS.sectors_per_cluster * BS.sector_size;
        char cluster_buffer[cluster_size];

        // go to the first cluster and read in the data
        lseek(DISK, first_sector_of_cluster(file_entry.starting_cluster, true) * BS.sector_size, SEEK_SET);
        read(DISK, &cluster_buffer, cluster_size);

        // check if there are more clusters
        unsigned short fat_value = 0;
        if (get_fat_value(&fat_value, true, 0))
        {
                // write to the output file and update the remaining file size
                write(output_file, &cluster_buffer, cluster_size);
                file_size -= cluster_size;

                // move to the next cluster
                lseek(DISK, first_sector_of_cluster(fat_value, true) * BS.sector_size, SEEK_SET);

                // pass by value and reference at once could cause issues
                unsigned short active_cluster = fat_value;
                while (get_fat_value(&fat_value, false, active_cluster))
                {
                        read(DISK, &cluster_buffer, cluster_size);

                        // write to the output file and update the remaining file size
                        write(output_file, &cluster_buffer, cluster_size);
                        file_size -= cluster_size;

                        // check if there is another cluster
                        if (file_size <= cluster_size) break;

                        // else, keep going
                        active_cluster = fat_value;

                        // move to the next cluster
                        lseek(DISK, first_sector_of_cluster(fat_value, true) * BS.sector_size, SEEK_SET);
                }
                // read one last time
                read(DISK, &cluster_buffer, cluster_size);
        }
        // write the remaining bytes to the output file
        write(output_file, &cluster_buffer, file_size);

        // close the file
        close(output_file);
}
```

```
// helper function for copy in, returns the offset of the dir entry written by this function
// very similar to 'ls'
unsigned int write_dir_entry(Fat16Entry dir_entry, list<char*>* arguments)
{
        // go to the directory if not there already
        if (!arguments->empty()) follow_path(arguments, true);

        // get the offset of the the first sector in this directory
        unsigned int first_sector = first_sector_of_cluster(CURRENT.starting_cluster, false);

        // read until there isn't an entry
        unsigned short counter = 0;
        while(1)
        {
                // keep reading if it's a long file name or if it's the volume ID
                if ((CURRENT.filename[0] == 0x00) && !(CURRENT.attributes & LFN)) break;

                // read next entry
                counter++;
                lseek(DISK, (first_sector * BS.sector_size) + (counter * sizeof(Fat16Entry)), SEEK_SET);
                read(DISK, &CURRENT, sizeof(Fat16Entry));
        }
        // write the dir entry here
        lseek(DISK, (first_sector * BS.sector_size) + (counter * sizeof(Fat16Entry)), SEEK_SET);
        unsigned int dir_entry_offset = lseek(DISK, 0, SEEK_CUR);
        write(DISK, &dir_entry, sizeof(Fat16Entry));

        // return to the offset at the beginning of the method
        read(DISK, &CURRENT, sizeof(Fat16Entry));
        return dir_entry_offset;
}
```

```
// copy in function implementation, file modify data and time are neglected, attributes are assumed 0x00
void copy_in(char* name, unsigned int* src_file, list<char*>* arguments)
{
        // get the cluster of pwd, before we start searching
        unsigned int starting_offset = lseek(DISK, 0, SEEK_CUR);

        // build the dir entry
        Fat16Entry dir_entry;
        dir_entry.attributes = 0;
        unsigned int file_size = lseek(*src_file, 0, SEEK_END);
        lseek(*src_file, 0, SEEK_SET);
        dir_entry.file_size = file_size;

        // build filename
        unsigned char filename[8];
        unsigned int i;
        for (i = 0; i < strlen(name) - 4; i++) filename[i] = name[i];

        // add spaces for any left over space
        while (i < 8)
        {
                filename[i] = ' ';
                i++;
        }

        // copy into the struct
        memcpy(&dir_entry.filename, &filename, 8);

        // file extension
        unsigned short index = 2;
        for (i = strlen(name) - 1; i > strlen(name) - 4; i--)
        {
                dir_entry.ext[index] = name[i];
                index--;
        }

        // read the entire FAT table
        unsigned char FAT_table[BS.fat_size_sectors * BS.sector_size];
        lseek(DISK, BS.reserved_sectors * BS.sector_size, SEEK_SET);
        read(DISK, &FAT_table, sizeof(FAT_table));

        // find clusters for the file data
        // start table index at 2 because 0 and 1 are reserved
        index = 2;
        unsigned int table_entry_offset;
        unsigned int prev_table_entry_index = 0xFFFF;
        bool found = false;
        unsigned char src_buffer[BS.sectors_per_cluster * BS.sector_size];
        while (1)
        {
                // get the value from the table
                unsigned short table_value = *(unsigned short*)&FAT_table[index * sizeof(short)];
```

```
                if (table_value == 0)
                {
                        // calculate the offset of this table entry
                        table_entry_offset = (index * sizeof(short)) + (BS.reserved_sectors * BS.sector_size);

                        // fill the fat table with the old index
                        if (found)
                        {
                                lseek(DISK,(prev_table_entry_index * sizeof(short)) +
                                        (BS.reserved_sectors * BS.sector_size), SEEK_SET);
                                write(DISK, &index, sizeof(short));
                        }
                        // first cluster found
                        else
                        {
                                // write FFFF in case this is the only cluster
                                dir_entry.starting_cluster = index;
                                found = true;
                                lseek(DISK, table_entry_offset, SEEK_SET);
                                write(DISK, &prev_table_entry_index, sizeof(short));
                        }
                        prev_table_entry_index = index;

                        // start writing data to the cluster
                        if (file_size > (BS.sectors_per_cluster * BS.sector_size))
                        {
                                read(*src_file, &src_buffer, BS.sectors_per_cluster * BS.sector_size);
                                lseek(DISK, first_sector_of_cluster(index, true) * BS.sector_size, SEEK_SET);
                                write(DISK, &src_buffer, BS.sectors_per_cluster * BS.sector_size);

                                // decrement remaining bytes
                                file_size -= BS.sectors_per_cluster * BS.sector_size;
                        }
                        else
                        {
                                // write remaining file data to the cluster
                                read(*src_file, &src_buffer, file_size);
                                lseek(DISK, first_sector_of_cluster(index, true) * BS.sector_size, SEEK_SET);
                                write(DISK, &src_buffer, file_size);

                                // write FFFF to the last cluster
                                lseek(DISK, table_entry_offset, SEEK_SET);
                                unsigned short last_entry = 0xFFFF;
                                write(DISK, &last_entry, sizeof(short));
                                break;
                        }
                }
                index++;
        }
        // find a spot for the file entry and write it
        lseek(DISK, starting_offset, SEEK_SET);
        write_dir_entry(dir_entry, arguments);

        // return to the original position
        lseek(DISK, starting_offset, SEEK_SET);
        read(DISK, &CURRENT, sizeof(Fat16Entry));
}
```

```
// main function
int main(int argc, char *argv[])
{
        // open the sample FAT16 volume
        DISK = open(argv[1], O_RDWR);

        // read boot sector into struct
        read(DISK, &BS, sizeof(Fat16BootSector));

        // compute the size of the root directory
        unsigned int root_dir_sectors = ((BS.root_dir_entries * 32) + (BS.sector_size - 1)) / BS.sector_size;

        // compute the offset of the first data sector
        first_data_sector = BS.reserved_sectors +
                (BS.number_of_fats * BS.fat_size_sectors) + root_dir_sectors;

        // compute the offset of the first root directory sector
        first_root_dir_sector = first_data_sector - root_dir_sectors;

        // set the current directory to the ROOT directory
        lseek(DISK, first_root_dir_sector * BS.sector_size, SEEK_SET);
        read(DISK, &CURRENT, sizeof(Fat16Entry));

        // put the root dir in path
        PWD.append("/");

        // char pointers to parse user input
        char *buf = (char*)malloc(sizeof(char)*1024);
        char *token;
        char *src;
        char *dest;

        // list to hold inputs for commands
        list<char*> arguments;

        while (1)
        {
                //root = true;
                check_and_set_root();

                // print the current directory
                cout << ":" << PWD << "> ";

                // read input from stdin
                fgets(buf, 1024, stdin);
                for (int x = 0; (unsigned)x < strlen(buf); x++) if ( buf[x] == '\n' || buf[x] == '\r' ) buf[x] = '\0';

                // exit program in user inputs 'exit'
                if (!strcmp(buf, "exit")) break;
```

```
// parse input
else
{
        // get command
        token = strtok(buf, " ");
        if (!strcmp(buf, "ls"))
        {
                // parse argument
                token = strtok(NULL, " /");

                // list present directory
                if (token == NULL) list_directory();

                // list directory path
                else
                {
                        // get the cluster of pwd, before we start searching
                        unsigned int starting_sector = first_sector_of_cluster(CURRENT.starting_cluster,false);

                        // put the path into a list
                        arguments = build_arguments(token);

                        // follow the path by calling change_directory for each directory in path
                        follow_path(&arguments, true);

                        // list that directory
                        list_directory();

                        // return to the original sector
                        lseek(DISK, starting_sector * BS.sector_size, SEEK_SET);
                        read(DISK, &CURRENT, sizeof(Fat16Entry));
                }
        }
        // change directory
        else if (!strcmp(buf, "cd"))
        {
                // parse argument, similar to 'ls'
                token = strtok(NULL, " /");

                // put the path into a list
                arguments = build_arguments(token);

                // change directory
                follow_path(&arguments, false);
        }
        // copy out
        else if (!strcmp(buf, "cpout"))
        {
                // get the cluster of pwd, before we start searching
                unsigned int starting_sector = first_sector_of_cluster(CURRENT.starting_cluster, false);

                // parse src and dest
                src = strtok(NULL, " ");
                dest = strtok(NULL, " ");

                // put the path into a list
                arguments = build_arguments(strtok(src, " /"));

                // copy the file to the destination on computer
                copy_out(dest, &arguments);

                // return to the original sector
                lseek(DISK, starting_sector * BS.sector_size, SEEK_SET);
                read(DISK, &CURRENT, sizeof(Fat16Entry));
        }
```

```
                        // copy in
                        else if (!strcmp(buf, "cpin"))
                        {
                                // get the cluster of pwd, before we start searching
                                unsigned int starting_sector = first_sector_of_cluster(CURRENT.starting_cluster, false);

                                // parse src and dest
                                src = strtok(NULL, " ");
                                dest = strtok(NULL, " ");

                                // put the path into a list
                                arguments = build_arguments(strtok(dest, " /"));

                                // open the file specified by src on unified system
                                unsigned int src_file = open(src, O_RDWR);

                                // pop off the file name
                                char* name = arguments.back();
                                arguments.pop_back();

                                // copy the file into dest path in the fat16 volume
                                copy_in(name, &src_file, &arguments);

                                close(src_file);

                                // return to the original position
                                lseek(DISK, starting_sector * BS.sector_size, SEEK_SET);
                                read(DISK, &CURRENT, sizeof(Fat16Entry));
                        }
                }
        }
        close(DISK);
        return 0;
}
```