CS 4414 Operating Systems – Spring 2015

File System Organization – A Study of the FAT16 File System

Homework 3

Robbie deMuth

To the best of my knowledge, my implementation of the specified primitive shell/file-manager meets all of the requirements presented in the assignment. After carefully expanding its capabilities, it is able to successfully perform each of the five required commands for both relative and absolute paths. Please note, however, that as specified in the latest update of the assignment, the *<dst>* parameter for both the *cpin* and the *cpout* command requires an explicit filename as part of the path. For example, to move a file from the raw file system image to the real unified file system, one would have to specify a new filename in addition to the path to the desired directory instead of relying on the shell to use the filename from the raw file system if none is given.

Problem Description:

The goal of this assignment was to implement a command line tool capable of inspecting, manipulating, and interacting with the FAT16 file system. Specifically, the program was required to act like a very primitive shell/file-manager that bridges the namespaces provided by both the unified file system exposed by the shell that launched program and the FAT16 file-system loaded by the file-manager. In this regard, the shell/file-manager was required to implement the following five commands: *cd <directory>*, *ls <directory>*, *cpin <src> <dst>*, *cpout <src> <dst>*, and *exit*.

Approach:

Before beginning to implement the simpler commands such as *cd* and *ls*, I first embarked on the task of reading and storing important file system information from the raw device block data. This mainly consisted of defining appropriate data structures to access information in the file system's boot block, FATs, root directory, and directory entries. Using information from a variety of provided sources, I created packed structs for both the boot block and the directory entries. By reading the section of the raw file system into the boot block struct, I was able to access and store key information about the file system, such as the FAT offset, the number of FATs, the cluster size, and the number of root directory entries. By storing this and other important information in global variables, I was able to access key information about the file system quickly and effectively within function calls without the need for excessive parameter passing on the stack.

After reading the boot block and its relevant information into memory, I created buffers to hold both a copy of the FAT and the root directory. By using information from the boot block, I was able to compute their required size and offset in the raw file system.

Stored as global variables, these would later become useful for easy access within function calls.

With the proper data stored global memory for easy access, I then made design decisions on how to represent the user's current working directory within the file system. First, I decided to create a current working directory string that would be used to represent the user's absolute path within the file system. Given that a large percentage of file system navigation involves computing cluster numbers, however, I also created a variable to store the first cluster number of the user's current working directory. Since the root directory does not have a starting cluster number, I decided to assign it a value of zero since that value is not used by any other directory or file within the system. This design decision allowed me to differentiate between the root directory and other normal directories when navigating throughout the file system.

At this point, I wrote code to handle user input and parse it into the appropriate commands that would later be implemented as function calls. To begin, I simply added logic to handle the *exit* command so that I could easily break from the shell when testing other commands and inputs. At the same time, I wrote a function call that simply displays the required prompt to *stdout* based on the user's current working directory string.

Then, I began work on the *cd <directory>* command. For reference, note that the function header for the command is *'bool cd(string& cwd, string path, uint16_t& cwdClustNum)'* where the current working directory string and cluster number are passed by reference so that any updates to the current working directory within the function are reflected in the programs execution in *main()*. The function is also designed to return a boolean indicating whether or not the given directory change was successful. First, the function checks if the *cd* command was provided without any arguments, in which case it simply returns true without altering the current working directory. If arguments were provided, however, the program creates an absolute path representing the desired new directory based on the current working directory and the *path* parameter. If the absolute path is equivalent to the root directory, the function simply sets the current working directory string to the empty string to represent the root directory and sets the cluster number to zero for the same reason.

For non-trivial directory changes, the program parses the absolute path into a vector of strings based on the '/' delimiter that is used to separate directories within a path. Then, I programmed a function that recursively removes '..' and '.' entries from the directory vector and computes the total number of necessary directory changes to reach the directory specified by the absolute path. If the sequence of directory changes results in the root directory, the program once again sets the current working directory to the root directory using the method described above.

Next, the program handles the first directory change, which it knows must take place from the root directory. Using the aforementioned directory entry struct, the program loops through the directory entries in the root directory buffer and obtains their

file name and extension. This is done using a function called *getEntryName()* that returns a string representing the name and extension of a given directory entry and handles special cases of long entry names and empty entries by returning sequences of space characters (since they cannot be found at the beginning of a file name in FAT16). By comparing the resulting directory entry file name and extension against the desired next directory in the path, the function either computes the starting cluster number of the next directory or prints an error message before returning false.

After hopefully obtaining the starting cluster number of the next directory in the sequence, the function knows that all other directory changes must take place from normal directories. For each remaining directory change, the function undertakes a process similar to that described above for the root directory, but the code accounts for the fact that a normal directory may in fact be stored over one or more clusters. To deal with this, I wrote a function called *readCluster()* that reads the contents of a given cluster into a specified buffer and then returns the file's next cluster number by accessing the FAT. After looping through each of the remaining directory changes, the function either reports that the specified path could not be found and returns false, or updates the current working directory string and cluster to the new current working directory and returns true.

In order to test the effectiveness of the *cd* command, I then wrote a function that implemented the *ls* command. While this function takes the same parameters as that of *cd()*, the parameters are passed by value to ensure that any directory changes performed within the *ls* function do not affect the user's current working directory. First, the *ls* function attempts to change directories according to its specified parameters. If the call to *cd* returns false as described above, the *ls* function simply returns (note that error message handling is provided by the *cd* function in this case). Otherwise, the *ls* function checks the value of its local copy of the current working directory to determine whether it should list the contents of the root directory, if the cluster number is zero, or a normal directory, otherwise.

If *ls* determines that it must list the contents of the root directory, it simply loops through the directory entries in the root directory buffer in a similar manner to the *cd* function. This time, however, it uses the information obtained from a call to *getEntryName()* to determine whether or not to print the current entry to *stdout*. If the entry is neither empty nor a long entry, a call to a function called *printEntryName()* is made that prints the directory entry output according to the specifications on Collab. For a normal directory, *ls* simply loops through a given directory's clusters via calls to *readCluster()* as described above and then loops through each cluster's contents to determine whether or not to print information to *stdout*. This approach of linearly searching through a file's clusters by reading them into a buffer is commonplace throughout the rest of the program. This does, however, require the additional use of certain boolean flags in some cases to determine if the program should stop searching through a file's clusters if it has found pertinent information within a given cluster.

At this point, I was able to test and debug the execution of the *cd* and *ls* commands by using the provided raw file system image. While I won't go into the details here, several iterations were required to successfully implement both commands, as neither could be easily tested without the completion of the other. For more information, refer to the results and analysis section.

After correcting the *cd* and *ls* commands, I began coding the *cpin* command as described in the requirements and on Collab. For reference, both the *cpin()* and *cpout()* commands have the following function header: *<function name>(string cwd, string fromPath, uint16_t cwdClustNum, string toPath)*. First, the function ensures that the file represented in the *fromPath* variable exists and can be opened by the operating system. If this is not the case, the function simply returns and does not edit the file system. Then, it computed the file size of the input file and computes the number of clusters needed to store that file in the file system. The function then attempts to find the required number of empty clusters using the FAT and stores them in an array. If the *cpin()* function cannot find enough available clusters in the FAT, it once again simply returns without making edits to the file system.

After finding enough clusters to store the input file, the function computes the path of the desired destination and the appropriate file name based on the *toPath* parameter. Recall that the introduction states that the file name must be included in the *toPath* variable. The function then attempts to change into the desired directory and returns without editing the file system if this directory change is unsuccessful. Next, *cpin()* determines whether or not there exists a file in the specified directory with the same file name by making a call to *containsEntryName()*. This function simply loops through the directory entries of a specified directory based on its starting cluster number and searches for an entry with a given file name and extension. If there is not a name conflict, the function then proceeds to create a directory entry for the file in the specified directory.

Next, the function searches through the appropriate directory for empty directory entries in which it could create an entry for the given file. This process is similar for both the root directory and normal directories, but normal directories need to be read cluster by cluster as described extensively above. If *cpin()* encounters an empty directory entry, it creates a c-string representation of the file name and extension that matches the FAT16 specification, and copies its contents into the directory entry. Similarly, it then sets the starting cluster of the file based on the first free cluster in the array that was created earlier in the function. If an empty entry indicates that the rest of the directory is empty, the program simply creates the directory entry in the current location and then writes the empty directory property into the next directory entry. Based on whatever directory and cluster number in which the entry is created, the program then writes either the root directory or the appropriate cluster back to the raw file system. However, if *cpin()* is unable to find an empty directory entry, it returns without making any changes.

At this point, *cpin()* knows that there are enough allocated clusters to hold the file and that the file has a corresponding entry in the appropriate directory. Therefore, the function alters the FAT buffer to link together the clusters of the file and then writes the FAT or FATs back to the raw file system. Lastly, for each of the file's clusters, the function reads an appropriate amount of bytes from the input file and then writes them to the appropriate cluster in the raw file system via a call to *writeCluster()*. Please note that the specific execution of *cpin()* ensures that no data is written back to the raw file system until the function has determined that there are enough available free clusters and that there is an available entry in the specified directory.

Next, I began implementing the *cpout* command. Compared to *cpin* this was relatively straightforward because of the lack of error handling it required in the event that there was not enough free space available. When it begins execution, *cpout()* first determines whether or not a file with the given path and name already exist in the real unified file system. If there is a conflict, it simply returns after printing an error message. Next, it computes the file name and path of the file to be copied and performs a *cd()* to the appropriate directory. If this directory change is unsuccessful, the function simply returns without writing to the real unified file system. By looping through the appropriate directory entries (as described multiple times above), the function computes the starting cluster of the given file, or simply returns if a file with the given name does not exist in the specified directory.

After computing the starting cluster number, *cpout()* attempts to open a file for writing in the real unified file system and returns after printing an error message if it is unable to do so. Finally, given the starting cluster number of the file to read, the function simply loops through the clusters via calls to *readCluster()* and writes the contents of the file from the buffer into which they were read into the real unified file system.

Just like the *cd* and *ls* commands, *cpin* and *cpout* initially required a decent amount of testing and debugging. This at first proved difficult because the testing of the two functions is closely related. While I won't go into vast detail here, more information can be found in the results and analysis section.

Results and Analysis:

Initially, the most difficult part of the assignment revolved around gaining an understanding of the different structures and their layouts within the FAT16 file system. However, even as that became clearer with research and practice, actually implementing many of desired commands was surprisingly difficult. While I was able to print the contents of the boot block to *stdout* to ensure that I had saved the correct information to memory, I ran into issues about how to implement the *cd* command. The first problem I encountered was devising a method to keep track of the user's current directory. While implementing the current working directory in a string was fairly straightforward, I ran into issues keeping track of where I should be reading from in the raw file system. After much thought

and some practice, I decided to also keep track of the starting cluster number of the user's current working directory. This provided me an easy way to differentiate between the root and normal directories and to loop through the directory entries for a given directory.

As this method became clear, however, I had issues testing its effectiveness because of the interdependence of the *cd* and *ls* commands. As hinted at above, it was difficult to initially test these commands because testing one required the proper implementation of the other. For example, it is difficult to initially test the effectiveness of a directory change since I had not yet completed the *ls* command required to display the contents of a directory. Similarly, I could not properly test the *ls* command to ensure that it functioned correctly on both the root and normal directories until I was able to change between directories. This interdependence encouraged iterative development and additional testing and debugging via output to *stdout* to ensure that each function performed the expected operation even as I had not yet completely finished coding the logic.

This same issue arose during the implementation of the *cpin* and *cpout* commands. While I was able to effectively test *cpout* by simply moving a file from the raw file system to my local drive and viewing its contents, testing *cpin* proved more difficult. Specifically, I was unable to effectively test my *cpin()* function until I had completed *cpout()*. This is because testing *cpin* required moving the copied file back to my local drive to ensure that its contents remained unchanged during the process of both copies. With more iterative development, however, I was able to get both commands to function properly. Eventually, I was able to test the implementation of both commands by copying various sized files to and from the raw file system to ensure that memory restrictions were met and that the contents of the files remained unchanged.

As I have brought up earlier, both the *cpin* and the *cpout* command require that the user include a specific file name in the destination path. Initially, I coded these commands such that they simply reused the filename from the source file. However, this prevented the user from being able to specify a new file name during the copying process. From a recent assignment update provided on Collab, an answer states that the instructions "suggest that the file name will be given". In the end, I decided to implement the required naming method instead since I was unable to implement both options. I ran into difficulty determining whether or not a file name was included in the path since file names are not required to have extensions. This caused issues differentiating between file names and folder names when searching through directories. I made my decision on which method to implement based on the functionality of both options. Essentially, I decided that required file naming encapsulated the optional version since one could simply retype the file name into the destination path if they wanted the file's name to remain unchanged.

Conclusion:

While this assignment certainly required a lot of work and entailed a somewhat steep learning curve, it greatly furthered my understanding of file systems. Specifically, I

have a much better understanding of how file systems are formatted as raw files and how one can use the information in a file systems boot block and FAT in order to traverse the file system and copy data back and forth from a raw file system to a local drive. This entailed learning how one can allocate memory to a file within the file system and how files are typically kept in non-contiguous clusters on disk. Overall, while this assignment was certainly daunting and difficult, it complemented my knowledge of file systems from course readings and lectures and also helped me better understand how they are implemented at the software level.

"On my honor, as a student, I have neither given nor received aid on this assignment."

Robert A. deMuth

```
/*
 * File System Organization - A Study of the FAT16 File System
 *
 * CS 4414 Operating Systems Homework 3
 * March 31, 2015
 *
 * Robbie deMuth
 *
 * fileSystem.cpp
 *
 * The following code implements a command line tool that can inspect and manipulate
 * FAT16 file system data. By reading information about the file system from the
 * boot block, the program acts as a primitive shell/file-manager that allows the
 * user to perform basic operations. These operations include simple tasks such as
 * 'cd <directory>', 'ls [directory]', 'cpin <src> <dst>', and 'cpout <src> <dst>'.
 *
 *      COMPILE:     make
 *      OBJECTS:     fileSystem.o
 *      HEADERS:     algorithm fcntl.h iostream sstream stdbool.h stdint.h stdio.h
 *                   stdlib.h string.h unistd.h vector
 *
 *      MODIFICATIONS:
 *          Mar 19 - Began work on reading information from the boot block and
 *                   storing relavant file system information in global varialbes
 *                   where appropriate
 *          Mar 22 - Began work on the 'cd' and 'ls' commands to navigate and
 *                   display information in the file system
 *          Mar 24 - Refined logic to read files and directories with multiple
 *                   clusters
 *          Mar 25 - Began work on the 'cpin' and 'cpout' commands
 *          Mar 29 - Fixed logic relating to errors when changing to non-existent
 *                   directories and attempting to copy in or out to non-existent
 *                   files and locations
 *          Mar 30 - Corrected output to match the new specifications and permormed
 *                   final tests on the new sample images
 */

#include <algorithm>
#include <fcntl.h>
#include <iostream>
#include <sstream>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <vector>
using namespace std;

// Boot sector struct
struct __attribute__((__packed__)) mbr_bpbFAT16 {
  uint8_t  BS_jmpBoot[3];     /* jump instruction to boot code */
  uint8_t  BS_OEMName[8];     /* name string */
  uint16_t BPB_BytesPerSec;   /* number of bytes per sector (almost always 512) */
  uint8_t  BPB_SecPerClust;   /* number of sectors per allocation unit (cluster) */
  uint16_t BPB_RsvdSecCnt;    /* number or reserved sectors */
  uint8_t  BPB_NumFATs;       /* number of FAT data structures */
  uint16_t BPB_RootEntCnt;    /* number of root directory entries (32 bytes each) */
  uint16_t BPB_TotSec16;      /* total number of sectors in the entire disk */
  uint8_t  BPB_Media;         /* media descriptor */
  uint16_t BPB_FATSz16;       /* number of sectors occupied by one FAT */
  uint16_t BPB_SecPerTrk;     /* number of sectors per track */
  uint16_t BPB_NumHeads;      /* number of heads */
  uint32_t BPB_HiddSec;       /* number of hidden sectors */
  uint32_t BPB_TotSec32;      /* total number of sectors in the entire disk */
  uint8_t  BS_DrvNum;         /* physical drive number */
  uint8_t  BS_Reserved1;      /* Reserved */
  uint8_t  BS_BootSig;        /* extended boot record signature */
  uint8_t  BS_VolID[4];       /* volume serial number */
  uint8_t  BS_VolLab[11];     /* volume label */
```

```
  uint8_t  BS_FilSysType[8];  /* file system identifier */
};

// Directory entry struct
struct __attribute__((__packed__)) dirEntry {
  uint8_t  DIR_Name[8];       /* name of the file */
  uint8_t  DIR_NameExt[3];    /* file name extension */
  uint8_t  DIR_Attr;          /* information about file permissions */
  uint8_t  DIR_NTRes;         /* Reserved */
  uint8_t  DIR_CrtTimeTenth;  /* Millisecond stamp in tenths of a second (0-199) */
  uint16_t DIR_CrtTime;       /* Time file was created */
  uint16_t DIR_CrtDate;       /* Date file was created */
  uint16_t DIR_LstAccDate;    /* Last access date */
  uint16_t DIR_FstClusHI;     /* High word of this entry's first cluster number (0 for FAT12 and
FAT16) */
  uint16_t DIR_WrtTime;       /* Time of last write */
  uint16_t DIR_WrtDate;       /* Date of last write */
  uint16_t DIR_FstClusLO;     /* Low word of this entry's first cluster number */
  uint32_t DIR_FileSize;      /* 32-bit word holding the file size in bytes */
};

/* Global definitions for file system data */
/*****************************************/

struct mbr_bpbFAT16 *bpb;                           /* Pointer to the boot sector struct */
struct dirEntry *RootDirBuffer;                     /* Pointer to the root directory buffer */
int RootDirOffset, RootDirSize, numRootDirEntries;  /* Information about the root directory */
uint16_t* FATBuffer;                                /* Pointer to the FAT buffer */
int FATOffset, FATSize, numFatEntries, numFATs;     /* Infomation about the FAT */
int bytesPerCluster, dirEntPerCluster;              /* Information about cluster size */
int fd;                                             /* File descriptor for the raw file system */
int dataOffset;                                     /* Byte offset of the data region */


/* Returns the number of occurences of the '/' character in a string */
int getNumDirChanges(string path) {
  int result = 0;
  // Loop through the characters in the string
  for (int i = 0; i < path.length(); i++) {
    if (path[i] == '/')
      result++;
  }
  return result;
}

/* Creates an absolute path from either an absolute or relative path */
string createAbsPath(string cwd, string path) {
  if (path.length() == 0) {
    // Path is empty
    return "";
  } else if (path.length() == 1 && path[0] == '/') {
    // Absolute path is the root directory
    return "";
  } else if (path[0] == '/') {
    // Path is already an absolute path
    // Remove the trailing '/' character if present
    if (path[path.length()-1] == '/')
      path.erase(path.length()-1, 1);
    return path;
  } else {
    // Create an absolute path from the relative path
    path = cwd + "/" + path;
    // Remove the trailing '/' character if present
    if (path[path.length()-1] == '/')
      path.erase(path.length()-1, 1);
    return path;
  }
}
```

```
/* Parses an absolute path into the specified vector */
void dirsToVector(vector<string> &dirVector, string path) {
  // Compute the number of directory changes in the path
  int numDirs = getNumDirChanges(path);
  // Resize the vector appropriately
  dirVector.resize(numDirs);
  // Parse the input string on the '/' character
  stringstream dirStream(path);
  getline(dirStream, dirVector[0], '/');
  for (int i = 0; i < numDirs; i++) {
    getline(dirStream, dirVector[i], '/');
  }
}

/* Removes '..' and '.' entries from the directory vector and returns the new size */
int simplifyPath(vector<string> &dirVector, int i) {
  if (i == dirVector.size()) {
    // Return the size of the vector
    return i;
  } else if (i == 0 && dirVector[0].compare("..") == 0) {
    // Remove '..' entries at the beginning of the vector
    dirVector.erase(dirVector.begin()+0);
    return simplifyPath(dirVector, 0);
  } else if (dirVector[i].compare("..") == 0) {
    // Remove '..' entries and the parent directory
    dirVector.erase(dirVector.begin()+i);
    dirVector.erase(dirVector.begin()+i-1);
    return simplifyPath(dirVector, i-1);
  } else if (dirVector[i].compare(".") == 0) {
    // Remove '.' entries
    dirVector.erase(dirVector.begin()+i);
    return simplifyPath(dirVector, i);
  } else {
    // Proceed to the next entry
    return simplifyPath(dirVector, i+1);
  }
}

/* Returns a file name from the directory entry pointer */
string getEntryName(uint8_t* data) {
  // Directory entry is a long directory entry name
  if ((data[11] ^ 0x0F) == 0) {
    return "";
  }
  string name = "";
  // Handle the first character in the file name
  if (data[0] == 0x00) {
    // Entire directory empty - return "  "
    return "  ";
  } else if (data[0] == 0xE5) {
    // Directory entry does not exist - return " "
    return " ";
  } else if (data[0] == 0x05) {
    // The file name character is 0xE5
    name += 0xE5;
  } else {
    // There is a normal file name character
    name += data[0];
  }
  // Read the remainder of the file name
  for (int j = 1; j < 8; j++) {
    if (data[j] == ' ')
      // No more characters to read
      break;
    name += data[j];
  }
  // Handle the first character in the file name extension
  if (data[8] != ' ') {
    // There is a file name extension
    name += '.';
```

```
    // Read the remainder of the file name extension
    for (int j = 8; j < 11; j++) {
      if (data[j] == ' ')
        // No more characters to read
        break;
      name += data[j];
    }
  }
  // Return the file name
  return name;
}

/* Returns a c-style string representing a directory entry file name and extension */
char* createEntryName(string name) {
  // Create a buffer to hold the returned string
  char* buffer = (char*) malloc(11*sizeof(char));
  // Fill the buffer with ' ' characters
  for (int i = 0; i < 11; i++) {
    buffer[i] = ' ';
  }
  // Find the offset of the file name extension
  size_t extOffset = name.find(".");
  if (extOffset == string::npos) {
    // Entry does not have a file name extension
    if (name.length() > 8) {
      // Create a shortened directory entry
      for (int i = 0; i < 6; i++) {
        buffer[i] = name[i];
      }
      // Use the '~' notation to shorten the entry
      buffer[6] = '~';
      buffer[7] = name[name.length()-1];
    } else {
      // Create a normal directory entry
      for (int i = 0; i < name.length(); i++) {
        buffer[i] = name[i];
      }
    }
  } else {
    // Entry has a file name extension
    if (extOffset > 8) {
      // Create a shortened directory entry
      for (int i = 0; i < 6; i++) {
        buffer[i] = name[i];
      }
      // Use the '~' notation to shorten the entry
      buffer[6] = '~';
      buffer[7] = name[extOffset-1];
    } else {
      // Create a normal directory entry
      for (int i = 0; i < extOffset; i++) {
        buffer[i] = name[i];
      }
    }
    // Add the file name extension
    for (int i = extOffset+1; i < name.length(); i++) {
      buffer[i-extOffset+7] = name[i];
    }
  }
  return buffer;
}

/* Prints a directory entry file name and extension to stdout according to the specification */
void printEntryName(uint8_t* data, string name) {
  if ((data[11] ^ 0x08) == 0) {
    // Entry corresponds to the volume label
    cout << "Volume Label: " << name << endl;
  } else if ((data[11] ^ 0x02) == 0) {
    // Entry corresponds to a hidden file
    cout << "H  " << name << endl;
  } else if ((data[11] ^ 0x04) == 0) {
```

```
    // Entry corresponds to a system file
    cout << "S  " << name << endl;
  } else if ((data[11] ^ 0x10) == 0) {
    // Entry corresponds to a directory
    cout << "D  " << name << endl;
  } else {
    // Entry corresponds to a regular file
    cout << "   " << name << endl;
  }
}

/* Reads the cluster with the given number into a buffer and returns the next cluster number */
uint16_t readCluster(uint8_t* buffer, uint16_t clusterNum) {
  // Compute the offset of the cluster to read
  int clusterOffset = dataOffset + (bytesPerCluster * (clusterNum - 2));
  // Read the cluster into the buffer
  lseek(fd, clusterOffset, SEEK_SET);
  read(fd, buffer, bytesPerCluster);
  // Compute and return the next cluster number in the file
  clusterNum = FATBuffer[clusterNum];
  return clusterNum;
}

/* Writes the contents of the specified buffer to the cluster with the given number */
void writeCluster(uint8_t* buffer, uint16_t clusterNum) {
  // Compute the offset of the cluster to write
  int clusterOffset = dataOffset + (bytesPerCluster * (clusterNum - 2));
  // Write the contents of the buffer to the cluster
  lseek(fd, clusterOffset, SEEK_SET);
  write(fd, buffer, bytesPerCluster);
}

/* Returns a bool indicating whether the given directory contains a file with the given name */
bool containsEntryName(string name, uint16_t clusterNum) {
  // Check whether or not we are in the root directory
  if (clusterNum == 0) {
    // Loop through the entries in the root directory
    for (int i = 0; i < numRootDirEntries; i++) {
      // Retrieve the file name of the current entry
      string entryName = getEntryName(RootDirBuffer[i].DIR_Name);
      if (entryName.compare("  ") == 0)
        // There are no more entries in the root directory
        return false;
      else if (entryName.compare(name) == 0)
        // Found a matching directory entry
        return true;
    }
  } else {
    // Create a buffer to hold the contents of the directory
    uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
    struct dirEntry* CurrDirBuffer = (struct dirEntry*) buffer;
    // Loop through the clusters of the directory
    while (clusterNum != 0xFFFF) {
      // Read the next cluster into the buffer
      clusterNum = readCluster(buffer, clusterNum);
      // Loop through the entries in the directory
      for (int i = 0; i < dirEntPerCluster; i++) {
        // Retrieve the file name of the current entry
       string entryName = getEntryName(RootDirBuffer[i].DIR_Name);
        if (entryName.compare("  ") == 0)
          // There are no more entries in the directory
          return false;
        else if (entryName.compare(name) == 0)
          // Found a matching directory entry
          return true;
      }
    }
  }
  return false;
}
```

```
/* Writes the FATs from the FATBuffer back to the raw file system */
void writeFAT() {
  // Seek the write head to the proper location
  lseek(fd, FATOffset, SEEK_SET);
  // Write the FATs back to the raw file system
  for (int i = 0; i < numFATs; i++) {
    write(fd, FATBuffer, FATSize);
  }
}

/* Writes the root directory from the RootDirBuffer back to the raw file system */
void writeRootDirectory() {
  // Seek the write head to the proper location
  lseek(fd, RootDirOffset, SEEK_SET);
  // Write the root directory back to the raw file system
  write(fd, RootDirBuffer, RootDirSize);
}

/* Prints the prompt to stdout according to the specification */
void prompt(string cwd) {
  if (cwd.length() == 0) {
    // We are in the root directory
    cout << ": / > ";
  } else {
    // We are in a normal directory
    cout << ": " << cwd << " > ";
  }
}

/* Changes directories by editing the cwd string and cwd cluster number */
bool cd(string& cwd, string path, uint16_t& cwdClustNum) {
  // Check if a change of directory is required
  if (path.length() == 0)
    return true;

  // Generate an absolute path
  string absPath = createAbsPath(cwd, path);

  // Check if the resulting path is equal to the root directory
  if (absPath.length() == 0) {
    cwd = "";
    cwdClustNum = 0x0000;
    return true;
  }

  // Local variables
  vector<string> dirs;
  uint16_t clusterNum;
  bool dirFound;

  // Parse the absolute path into the directory vector
  dirsToVector(dirs, absPath);
  // Simplify the path by removing '..' and '.' entries
  int numDirChanges = simplifyPath(dirs, 0);

  // Check if the resulting path is equal to the root directory
  if (numDirChanges == 0) {
    cwd = "";
    cwdClustNum = 0x0000;
    return true;
  }

  // Perform the first directory change from the root directory

  // Loop through the entries in the root directory
  for (int i = 0; i < numRootDirEntries; i++) {
    // Retrieve the file name of the current entry
    string entryName = getEntryName(RootDirBuffer[i].DIR_Name);
    if (entryName.compare("  ") == 0) {
      // There are no more entries in the root directory
      cout << "Directory does not exist" << endl;
```

```
      return false;
    } else if (entryName.compare("") == 0 || entryName.compare(" ") == 0) {
      // The current entry does not exist
      continue;
    } else if (entryName.compare(dirs[0]) == 0 && (RootDirBuffer[i].DIR_Attr & 0x10)) {
      // Found a matching directory entry
      clusterNum = RootDirBuffer[i].DIR_FstClusLO;
      break;
    }
  }

  // Perform all other normal directory changes

  // Loop through the remaining directory changes
  for (int i = 1; i < numDirChanges; i++) {
    // Create a buffer to hold the contents of the directory
    uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
    struct dirEntry* CurrDirBuffer = (struct dirEntry*) buffer;
    // Loop through the clusters of the directory
    while (clusterNum != 0xFFFF) {
      // Read the next cluster into the buffer
      clusterNum = readCluster(buffer, clusterNum);
      // Loop through the entries in the directory
      for (int j = 0; j < dirEntPerCluster; j++) {
        // Retrieve the file name of the current entry
        string entryName = getEntryName(CurrDirBuffer[j].DIR_Name);
        if (entryName.compare("  ") == 0) {
          // There are no more entries in the directory
          cout << "Directory does not exist" << endl;
          return false;
        } else if (entryName.compare("") == 0 || entryName.compare(" ") == 0) {
          // The current entry does not exist
          continue;
        } else if (entryName.compare(dirs[i]) == 0 && (CurrDirBuffer[j].DIR_Attr & 0x10)) {
          // Found a matching directory entry
          clusterNum = CurrDirBuffer[j].DIR_FstClusLO;
         // Set the flag to quit searching the remaining clusters
         dirFound = true;
         break;
        }
      }
      if (dirFound) {
        break;
      }
    }
    // Deallocate the buffer
    delete buffer;
    // There are no more clusters in the directory
    if (!dirFound) {
      cout << "Directory does not exist" << endl;
      return false;
    }
  }

  // Update the path of the current working directory
  cwd = "";
  for (int i = 0; i < dirs.size(); i++) {
    cwd = cwd + "/" + dirs[i];
  }
  // Update the cluster number of the current working directory
  cwdClustNum = clusterNum;
  // Indicate that we have successfully changed directories
  return true;
}
```

```
/* Lists the entries in the specified directory */
void ls(string cwd, string path, uint16_t cwdClustNum) {

  // Attempt to change into the specified directory
  if (!cd(cwd, path, cwdClustNum)) {
    // The above call will produce an error message if the directory does not exist
    return;
  }

  // Check whether or not we are in the root directory
  if (cwdClustNum == 0) {
    // Loop through the entries in the root directory
    for (int i = 0; i < numRootDirEntries; i++) {
      // Retrieve the file name of the current entry
      string entryName = getEntryName(RootDirBuffer[i].DIR_Name);
      if (entryName.compare("  ") == 0) {
        // There are no more entries in the root directory
        return;
      } else if (entryName.compare("") == 0 || entryName.compare(" ") == 0) {
        // The current entry does not exist or is part of a long entry
        continue;
      } else {
        // Print out the name of the current entry
        printEntryName(RootDirBuffer[i].DIR_Name, entryName);
      }
    }
  } else {
    // Create a buffer to hold the contents of the directory
    uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
    struct dirEntry* CurrDirBuffer = (struct dirEntry*) buffer;
    // Loop through the clusters of the directory
    while (cwdClustNum != 0xFFFF) {
      // Read the next cluster into the buffer
      cwdClustNum = readCluster(buffer, cwdClustNum);
      // Loop through the entries in the directory
      for (int i = 0; i < dirEntPerCluster; i++) {
        // Retrieve the file name of the current entry
        string entryName = getEntryName(CurrDirBuffer[i].DIR_Name);
        if (entryName.compare("  ") == 0) {
          // There are no more entries in the directory
          delete buffer;
          return;
        } else if (entryName.compare("") == 0 || entryName.compare(" ") == 0) {
          // The current entry does not exist or is part of a long entry
          continue;
        } else {
          // Print out the name of the current entry
          printEntryName(CurrDirBuffer[i].DIR_Name, entryName);
        }
      }
    }
    // Deallocate the buffer
    delete buffer;
  }
}

/* Copies the contents of a file into the file system loaded by the program */
void cpin(string cwd, string fromPath, uint16_t cwdClustNum, string toPath) {

  // Verify that we can read the file specified by the 'fromPath' variable
  int fdIn;
  if ((fdIn = open(fromPath.c_str(), O_RDONLY)) == -1) {
    // Operating system was unable to open the specified file
    cout << "Directory does not exist" << endl;
    return;
  }

  // Calculate the size of the file
  FILE* fp = fopen(fromPath.c_str(), "r");
  fseek(fp, 0L, SEEK_END);
  int fileSize = ftell(fp);
```

```
fclose(fp);

// Compute the number of clusters needed to store the file
int numClustersNeeded;
if (fileSize % bytesPerCluster == 0)
  // File fits in an even number of clusters
  numClustersNeeded = (fileSize / bytesPerCluster);
else
  // File does not fit in an even number of clusters
  numClustersNeeded = (fileSize / bytesPerCluster) + 1;

// Search through the FAT for the needed number of free clusters
uint16_t* clusterArray = (uint16_t*) malloc(numClustersNeeded*sizeof(uint16_t));
int numClustersFound = 0;
// Loop through the entries in the FAT
for (int i = 2; i <  numFatEntries; i++) {
  if (FATBuffer[i] == 0x0000) {
    // Add the free cluster to the array
    clusterArray[numClustersFound++] = i;
    // Quit searching if we have found enough clusters
    if (numClustersFound == numClustersNeeded)
     break;
  }
}
// Check if we have found enough free clusters
if (numClustersFound < numClustersNeeded) {
  cout << "Insufficient space" << endl;
  // Deallocate memory and close files
  delete clusterArray;
  close(fdIn);
  return;
}

// Calculate the name and path of the new file from the 'toPath' variable
string fileName;
size_t indexSlash = toPath.rfind("/");
if (indexSlash == string::npos) {
  // Path variable represents a file name
  fileName = toPath;
  toPath = "";
} else {
  // Path variable represents a path and a file name
  fileName = toPath.substr(indexSlash+1);
  toPath = toPath.substr(0,indexSlash);
}

// Attempt to change into the specified directory
if (!cd(cwd, toPath, cwdClustNum)) {
  // The above call will produce an error message if the directory does not exist
  // Deallocate memory and close files
  delete clusterArray;
  close(fdIn);
  return;
}

// Check if the destination file already exists
if (containsEntryName(fileName, cwdClustNum)) {
  cout << "File already exists" << endl;
  // Deallocate memory and close files
  delete clusterArray;
  close(fdIn);
  return;
}

// Attempt to create a directory entry for the given file

bool wroteEntry = false;
```

```
// Check whether or not we are in the root directory
if (cwdClustNum == 0) {
  // Loop through the entries in the root directory
  for (int i = 0; i < numRootDirEntries; i++) {
    // Retrieve the file name of the current entry
    string entryName = getEntryName(RootDirBuffer[i].DIR_Name);
    if (entryName.compare(" ") == 0 || entryName.compare("  ") == 0) {
      // The current entry does not exist
      // Create a file name and extension formatted for FAT16
      char* name = createEntryName(fileName);
      // Copy the file name and extension into the directory entry
      for (int j = 0; j < 11; j++) {
        RootDirBuffer[i].DIR_Name[j] = name[j];
      }
      // Deallocate the buffer
      delete name;
      // Set the starting cluster of the file
      RootDirBuffer[i].DIR_FstClusLO = clusterArray[0];
      // Set the flag to indicate that we have created a directory entry
      wroteEntry = true;
    }
    if (entryName.compare("  ") == 0) {
      if (i+1 < numRootDirEntries)
        RootDirBuffer[i+1].DIR_Name[0] = 0x00;
    }
    if (wroteEntry)
      break;
  }
  // Check if we were able to create a directory entry
  if (!wroteEntry) {
    cout << "Root directory is full" << endl;
    // Deallocate memory and close files
    delete clusterArray;
    close(fdIn);
    return;
  }
  // Write the root directory back to the raw file system
  writeRootDirectory();
} else {
  // Create a buffer to hold the contents of the directory
  uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
  struct dirEntry* CurrDirBuffer = (struct dirEntry*) buffer;
  // Loop through the clusters of the directory
  while (cwdClustNum != 0xFFFF) {
    // Create a temporary variable to be used if we find a free entry
    uint16_t tmpClustNum = cwdClustNum;
    // Read the next cluster into the buffer
    cwdClustNum = readCluster(buffer, cwdClustNum);
    // Loop through the entries in the directory
    for (int i = 0; i < dirEntPerCluster; i++) {
      // Retrieve the file name of the current entry
      string entryName = getEntryName(CurrDirBuffer[i].DIR_Name);
      if (entryName.compare(" ") == 0 || entryName.compare("  ") == 0) {
        // The current entry does not exist
        // Create a file name and extension formatted for FAT16
        char* name = createEntryName(fileName);
        // Copy the file name and extension into the directory entry
        for (int j = 0; j < 11; j++) {
          CurrDirBuffer[i].DIR_Name[j] = name[j];
        }
        // Deallocate the buffer
        delete name;
        // Set the starting cluster of the file
        CurrDirBuffer[i].DIR_FstClusLO = clusterArray[0];
        // Write the directory back to the raw file system
        writeCluster(buffer, tmpClustNum);
        // Set the flag to indicate that we have created a directory entry
        wroteEntry = true;
      }
```

```
        if (entryName.compare("  ") == 0) {
          if (i+1 < dirEntPerCluster) {
             CurrDirBuffer[i+1].DIR_Name[0] = 0x00;
             // Write the directory back to the raw file system
             writeCluster(buffer, tmpClustNum);
          }
        }
        if (wroteEntry)
          break;
      }
      // Quit searching the remaining clusters
      if (wroteEntry) {
       // Deallocate the buffer
       delete buffer;
       break;
      }
    }
    // Check if we were able to create a directory entry
    if (!wroteEntry) {
      // Deallocate the buffer
      delete buffer;
      cout << "Current directory is full" << endl;
      // Deallocate memory and close files
      delete clusterArray;
      close(fdIn);
      return;
    }
  }

  // Link the file's clusters together in the FAT
  for (int i = 0; i < numClustersNeeded - 1; i++) {
    FATBuffer[clusterArray[i]] = clusterArray[i+1];
  }
  // Set the last cluster to point to 0xFFFF
  FATBuffer[clusterArray[numClustersNeeded-1]] = 0xFFFF;

  // Write the FAT back to the raw file system
  writeFAT();

  // Write the contents of the file to the raw file system

  // Seek to the beginning of the input file
  lseek(fdIn, 0, SEEK_SET);
  // Allocate a buffer to hold the contents of one cluster
  uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
  // Loop through the clusters in the file
  for (int i = 0; i < numClustersNeeded; i++) {
    // Read a cluster of the file into the buffer
    read(fdIn, buffer, bytesPerCluster);
    // Write the current cluster to the raw file system
    writeCluster(buffer, clusterArray[i]);
  }
  // Deallocate the buffer
  delete buffer;
  // Deallocate memory and close files
  delete clusterArray;
  close(fdIn);
}

/* Copies the contents of a file into the "real" unified file system */
void cpout(string cwd, string fromPath, uint16_t cwdClustNum, string toPath) {

  // Check if the destination file already exists
  int fdOut;
  if ((fdOut = open(toPath.c_str(), O_RDONLY)) != -1) {
    // Operating system was able to open a file with the given file name
    cout << "File already exists" << endl;
    // Close files
    close(fdOut);
    return;
  }
```

```
// Calculate the name and path of the old file from the 'fromPath' variable
string fileName;
size_t indexSlash = fromPath.rfind("/");
if (indexSlash == string::npos) {
  // Path variable represents a file name
  fileName = fromPath;
  fromPath = "";
} else {
  // Path variable represents a path and a file name
  fileName = fromPath.substr(indexSlash+1);
  fromPath = fromPath.substr(0,indexSlash);
}

// Attempt to change into the specified directory
if (!cd(cwd, fromPath, cwdClustNum)) {
  // The above call will produce an error message if the directory does not exist
  return;
}

// Compute the starting cluster number of the specified file

bool entryFound = false;
uint16_t fileClustNum;

// Check whether or not we are in the root directory
if (cwdClustNum == 0) {
  // Loop through the entries in the root directory
  for (int i = 0; i < numRootDirEntries; i++) {
    // Retrieve the file name of the current entry
    string entryName = getEntryName(RootDirBuffer[i].DIR_Name);
    if (entryName.compare("  ") == 0) {
      // There are no more entries in the root directory
     cout << "Unable to locate file" << endl;
      return;
    } else if (entryName.compare("") == 0 || entryName.compare(" ") == 0) {
      // The current entry does not exist or is part of a long entry
      continue;
    } else if (entryName.compare(fileName) == 0) {
      // Found a matching directory entry
      fileClustNum = RootDirBuffer[i].DIR_FstClusLO;
     // Set the flag to indicate that we have found the matching directory entry
     entryFound = true;
     break;
    }
  }
  // Check if we were able to find a matching entry
  if (!entryFound) {
    cout << "Unable to locate file" << endl;
    return;
  }
} else {
  // Create a buffer to hold the contents of the directory
  uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
  struct dirEntry* CurrDirBuffer = (struct dirEntry*) buffer;
  // Loop through the clusters of the directory
  while (cwdClustNum != 0xFFFF) {
    // Read the next cluster into the buffer
    cwdClustNum = readCluster(buffer, cwdClustNum);
    // Loop through the entries in the directory
    for (int i = 0; i < dirEntPerCluster; i++) {
      // Retrieve the file name of the current entry
      string entryName = getEntryName(CurrDirBuffer[i].DIR_Name);
      if (entryName.compare("  ") == 0) {
        // There are no more entries in the directory
       cout << "Unable to locate file" << endl;
        return;
      } else if (entryName.compare("") == 0 || entryName.compare(" ") == 0) {
        // The current entry does not exist or is part of a long entry
        continue;
      } else if (entryName.compare(fileName) == 0) {
```

```
          // Found a matching directory entry
          fileClustNum = CurrDirBuffer[i].DIR_FstClusLO;
          // Set the flag to indicate that we have found the matching directory entry
          entryFound = true;
          break;
        }
      }
      // Quit searching the remaining clusters
      if (entryFound) {
        // Deallocate the buffer
        delete buffer;
        break;
      }
    }
    // Check if we were able to find a matching entry
    if (!entryFound) {
      // Deallocate the buffer
      delete buffer;
      cout << "Unable to locate file" << endl;
      return;
    }
  }

  // Write the contents of the file to the "real" unified file system

  // Check if the destination path is valid
  if ((fdOut = open(toPath.c_str(), O_WRONLY | O_CREAT | O_APPEND, 00700)) == -1) {
    cout << "Directory does not exist" << endl;
    return;
  }
  // Seek to the beginning of the output file
  lseek(fdOut, 0, SEEK_SET);
  // Allocate a buffer to hold the contents of one cluster
  uint8_t* buffer = (uint8_t*) malloc(bytesPerCluster);
  // Loop through the clusters in the file
  while (fileClustNum != 0xFFFF) {
    // Read the next cluster of the input file into the buffer
    fileClustNum = readCluster(buffer, fileClustNum);
    // Write the contents of the buffer to the output file
    write(fdOut, buffer, bytesPerCluster);
  }
  // Deallocate the buffer
  delete buffer;
}

int main(int argc, char* argv[]) {

  /* Definitions of local data */
  /*****************************/

  uint16_t cwdClustNum;        /* Current working directory's starting cluster number */
  string cwd;                  /* String representing the current working directory */
  string line;                 /* Variable to hold the user's full line of input */
  string command, arg1, arg2;  /* Variables to hold information about each command and parameters
*/

  // Check for valid number of command line args
  if (argc == 1) {
    cout << "No file provided" << endl;
    return 1;
  } else if (argc != 2) {
    cout << "Invalid command line arguments" << endl;
    return 1;
  }

  // Attempt to open the file
  if ((fd = open(argv[1], O_RDWR)) == -1) {
    cout << "Invalid command line arguments" << endl;
    return 1;
  }
```

```
// Allocate a buffer to read the boot block
bpb = (struct mbr_bpbFAT16*) malloc(sizeof(struct mbr_bpbFAT16));
// Read the boot block into the allocated buffer
lseek(fd, 0, SEEK_SET);
read(fd, bpb, sizeof(struct mbr_bpbFAT16));

// Compute information about the size and location of the FAT
FATOffset = bpb->BPB_BytesPerSec * bpb->BPB_RsvdSecCnt;
FATSize = bpb->BPB_FATSz16 * bpb->BPB_BytesPerSec;
numFatEntries = FATSize / 2;
numFATs = bpb->BPB_NumFATs;
// Allocate a buffer to read the FAT
FATBuffer = (uint16_t*) malloc(numFatEntries*sizeof(uint16_t));
// Read the FAT into allocated buffer
lseek(fd, FATOffset, SEEK_SET);
read(fd, FATBuffer, FATSize);

// Compute information about the size and location of the root directory
RootDirOffset = FATOffset + (FATSize * bpb->BPB_NumFATs);
numRootDirEntries = bpb->BPB_RootEntCnt;
RootDirSize = numRootDirEntries * 32;
// Allocate a buffer to read the root directory and the current directory
RootDirBuffer = (struct dirEntry*) malloc(numRootDirEntries*sizeof(struct dirEntry));
// Read the root directory into the allocated buffer
lseek(fd, RootDirOffset, SEEK_SET);
read(fd, RootDirBuffer, RootDirSize);

// Compute the data offset
dataOffset = RootDirOffset + RootDirSize;
// Compute information about the cluster size
bytesPerCluster = bpb->BPB_SecPerClust * bpb->BPB_BytesPerSec;
dirEntPerCluster = bytesPerCluster / 32;

// Set the current working directory to the root directory
cwd = "";
cwdClustNum = 0;

// Wait for user input
while (true) {

  // Print the prompt to stdout
  prompt(cwd);

  // Read a line of input
  getline(cin,line);
  // Parse the input into a command and two arguments
  stringstream lineStream(line);
  getline(lineStream, command, ' ');
  getline(lineStream, arg1, ' ');
  getline(lineStream, arg2, ' ');

  // Check if the user entered the 'exit' command
  if (command.compare("exit") == 0)
    break;

  // Check if the user entered the 'cd' command
  if (command.compare("cd") == 0)
    cd(cwd, arg1, cwdClustNum);

  // Check if the user entered the 'ls' command
  else if (command.compare("ls") == 0)
    ls(cwd, arg1, cwdClustNum);
```

```
    // Check if the user entered the 'cpin' command

    /* <dst> parameter must represent a path and desired new file name
     *
     * examples:
     *   - /DIR1/DIR2/file.txt  -->  valid
     *   - /DIR1/DIR2/DIR3  ------>  invalid
     */
    else if (command.compare("cpin") == 0)
      cpin(cwd, arg1, cwdClustNum, arg2);

    // Check if the user entered the 'cpout' command

    /* <dst> parameter must represent a path and desired new file name
     *
     * examples:
     *   - /DIR1/DIR2/file.txt  -->  valid
     *   - /DIR1/DIR2/DIR3  ------>  invalid
     */
    else if (command.compare("cpout") == 0)
      cpout(cwd, arg1, cwdClustNum, arg2);

    // User entered an invalid command
    else
      cout << "Invalid command. Please try again" << endl;

    // Reset the strings for the next iteration
    command = "";
    arg1 = "";
    arg2 = "";

  }

  return 0;

}
```