Writeup:
Comments:
Code:

CS 4414 Operating Systems – Fall 2017

Homework 3: FAT Read Library

Jonathan Colen jc8kf@virginia.edu

The assignment was completed and runs correctly for both FAT16 and FAT32.

1. Problem Description

The objective of this assignment was to become familiar with file system organization and learn how file allocation tables (FATs) are used to organize disk space. The program had to provide functions to change directory, list entries of a directory, and open, close, and read files contained in a FAT16 or FAT32 formatted volume.

2. Approach

In order to use the library created for this assignment, a FAT formatted volume must be designated by setting the environment variable FAT_FS_PATH. There are five library methods that are implemented in this assignment. *OS_cd* changes the current working directory. *OS_open* locates a file at a specified path and returns a file descriptor that can be used to read the file. *OS_close* closes a file specified by a file descriptor. *OS_read* reads a set number of bytes from a given file into a buffer. *OS_readDir* returns a list of entries in a specified directory. The library also defines a structure called *dirEnt* which represents a directory entry as described in the FAT specification. The current working directory is represented by a pointer to a list of *dirEnt* structures representing entries in the current working directory. These five library methods are implemented by using a combination of several helper methods, such as *init_fat*, *read_cluster_dirEnt*, *findDir*, and *findDirEntry*, which will be described below.

At the beginning of each method, the program checks if the FAT volume has been opened for reading. If it has not been opened, the volume is opened and basic information is recorded using the *init_fat* method. The purpose of *init_fat* is to load basic information about the FAT volume, store that information for later use, and initialize the current working directory and other critical information. The first 36 bytes of the FAT volume are read into a *BPB_structure* which represents the BIOS Parameter Block (BPB) as described in the FAT specification. The program reads the byte sequence starting at the end of the BPB into extended boot record (EBR) structures for both FAT16 and FAT32 systems. The information contained in the BPB and EBR is used to determine the count of clusters in the system. From this count, the file system type, FAT16 or FAT32, is determined and stored in a global variable. Next, the root entries are read in. For FAT16 volumes, the location of the root directory can be calculated from the BPB and the number of root directory entries are specified by the BPB by the field *BPB_RootEntCnt*. The system starts at the root sector and reads information in *BPB_RootEntCnt* consecutive *dirEnt* structures. For FAT32 volumes, the cluster containing the root directory entry is stored in the FAT32 EBR, and the entries are read in using the *read_cluster_dirEnt* method. A pointer

representing the list of these root directory entries is stored globally, and the current working directory pointer is set to this pointer.

The *read_cluster_dirEnt* method is one of several helper methods that are used in this library. The purpose of this method is to read a list of directory entries from a given cluster. First, the length of the cluster chain is calculated by following the entries in the FAT table starting at the given cluster. This is done by calling the *value_in_FAT* method, which retrieves a FAT table entry at a specified cluster using the process in the FAT specification, repeatedly until the “no next cluster” value (y > 0xFFF8 or y > 0x0FFFFFF8 for FAT16 and FAT32 respectively) is found. The total number of clusters in the chain multiplied by the number of directory entries that can be contained in a single cluster and a pointer containing that number of *dirEnt* structs is allocated. The function then reads 32 bytes at a time, starting from the sector corresponding to the first cluster, into the *dirEnts* represented by this pointer. When the end of the cluster is reached, the next cluster is found by consulting the FAT table. This process is repeated until either the end of the final cluster is reached, or a *dirEnt* is read with a first byte value of 0. This indicates that there are no more entries to be read, and the method returns the pointer containing the directory entries.

The *findDir* and *findDirEntry* methods are also useful in implementing the five library functions. The *findDirEntry* method finds a directory entry in a list of directory entries matching a specified name and stores it in a pointer passed as an argument to the function. The pointer of directory entries is iterated through until either an entry with a first byte of 0 is found, indicating no more entries, or a name match is found. The names are matched either by using a sequence of long filename structures or by using the 8.3 specification. This program does support long filenames, but because they are beyond the scope of this assignment, the implementation will not be discussed further here. The 8.3 specification creates a filename from an 11 character directory entry name by placing a period in between the first 8 and last 3 characters. This, “CONGRATSTXT” becomes “CONGRATS.TXT”. When a directory entry is found with a name matching that specified by the argument to *findDirEntry*, it is stored in the pointer passed to the function and a value of 1 is returned. If no matching entry is found, the pointer is left unmodified and the return code is 0.

The *findDirEntry* function is used to implement the *findDir* method, which is the most important helper method for this library. The purpose of *findDir* is to return a pointer to a list of directory entries contained at a specified path, given a directory to start from. The starting directory is represented by a list of directory entries contained in that directory, similar to how the current working directory is represented. This function separates the path name into *first* and *remaining*, where *first* is the first entry in the path and *remaining* is the path to be followed after the first entry. The function then calls *findDirEntry* on the current directory to match an entry to the name in *first*. If no such entry is found, the function returns NULL. Otherwise, if that directory entry has an attribute corresponding to a directory, the list of entries in that directory is retrieved using *read_cluster_dirEnt* and stored in *next_dir*. Next, *findDir* is called recursively on this list of entries with the new path given by *remaining*. The base case for the recursive call is when the specified path is empty, upon which the function returns NULL. If an upper level function receives NULL from a recursive call to *findDir*, it will return *next_dir* if it sees *remaining* is empty and will return NULL otherwise. If the upper level function receives a result that is not NULL, it will return that result. At the top level, the function returns the list of directory entries in the directory at the location of the specified path.

With the described helper methods, all of the library functions can be implemented simply. *OS_readDir* is a wrapper around *findDir* to return a list of directory entries contained at a relative or absolute path. If the specified directory name contains a leading slash, *findDir* is called with the stored root directory entries from *init_fat* as the current directory. Otherwise, *findDir* is called with the current working directory. In either case, the results of *findDir* are returned by *OS_readDir*.

OS_cd accepts a path and sets the current working directory to that path. This is done by calling *OS_readDir* and setting the list of directory entries representing the current directory to the results of *OS_readDir*. If these results are NULL, the function returns -1 and the current directory is left unchanged. Otherwise, it is modified and the function returns 1.

OS_open accepts a path to a file, generates a file descriptor for the file, and returns it. The file is located by separating the path into the filename, which is the section of the path string after the final slash, and the file path, which is the part preceding the final slash. The function then calls *OS_readDir* to get the directory entries at the location specified by the file path. Next, *findDirEntry* is called on this list of directory entries with the filename as the name to be matched. If either of these methods fail, the function returns -1. Otherwise, a file descriptor for the file is found and returned.

File descriptors in this program are represented by a large array of integers and a corresponding array of *dirEnt* structures. The integer array stores the first cluster number of the file pointed to by each file descriptor, or -1 if the file is unopened. The *dirEnt* array stores the *dirEnts* representing the opened files. This array is stored to ensure that no reads are performed past the end of a file. The integer array was created separate from the *dirEnt* array in order to allow for future implementation of a *seek* method, which would allow a file to be read at a specified offset. In order to find a file descriptor, the integer array is iterated through until an entry with value -1 is found. The index of this entry is the file descriptor, and the first cluster of the file and its *dirEnt* are stored in the two arrays.

When *OS_close* is called, the entry in the integer array of file descriptors corresponding to the argument is set to -1. If this entry is already set to -1, or if it is outside the bounds of the array, the function returns -1. Otherwise, the function returns 1.

The final method in the library is *OS_read*. This function allows a user to read a number of bytes into a buffer from an opened file starting from some offset. If the file descriptor passed as an argument does not correspond to an opened file as per the file descriptor arrays, the function returns -1. Otherwise the function attempts to read the specified number of bytes into the given buffer. There are a few complications that can arise that *OS_read* seeks to handle. First, it is possible that the specified offset goes past the end of the file. To handle this, the function checks whether the offset is either greater than the file size as specified by the file's *dirEnt*, or if the offset would result in reading at a cluster past the end of the file. The second condition is evaluated by counting the number of clusters the file spans by following the cluster chain in the FAT, and checking if the offset, converted from bytes to clusters, is greater than this number. If the offset is valid, the function reads the file until the desired number of bytes has been read, or until the end of the file, whichever comes first. If the number of bytes to read will result in reaching the end of a cluster, the function reads until the end of the cluster, seeks to the next cluster, and starts reading again from there. Once the desired number of bytes has been read, or the end of the file has been reached (which happens when the number of bytes read is equal to the file size specified by the *dirEnt* for the given file descriptor), the function returns the total number of bytes read from the file.

3. Problems Encountered

One significant problem encountered was posed by the ".. " entries for subdirectories of root. In all other directories, this entry has a first cluster number corresponding to the directory file of the current directory's parent. However, for those directories whose parents are root, this entry had a cluster number of 0 in the FAT volumes given. This would result in an incorrect sector to read from, causing calls such as *OS_cd("/People/../");* to fail. To circumvent this problem, *read_cluster_dirEnt* was modified to check if the cluster argument was 0. If so, the function started reading from the calculated root sector from the BPB in FAT16, or from the root cluster specified by the EBR in FAT32.

Another set of problems were those posed by *OS_read*. It is possible for users to specify an offset past the end of a file, which could result in reads to other files. In addition, it is possible for the desired number of bytes to be more than the number contained in the opened file. These problems were resolved by checking the offset before beginning to read, and by counting the number of bytes read and terminating the read when this number reaches the file size.

4. Testing

The library functions were tested on two given FAT volumes. One was formatted using FAT16, and the other was formatted using FAT32. In the testing program, the *OS_cd* and *OS_readDir* functions were called on a variety of path names. These path names could be relative or absolute, valid or invalid, and often contained several ".." entries to check the problem described earlier. The structure of the given volumes were inspected using the *hexcurse* hex editor, and the output of *OS_readDir* was compared to the raw data in this volume. The byte offset in the volume was tracked through extensive print statements at first to ensure that the programs were navigating the volume correctly. The correctness of *OS_cd* was evaluated by ensuring valid output from *OS_readDir* called on relative pathnames after *OS_cd* calls.

OS_open was called on a variety of real and fake filenames to ensure that file descriptors were only being returned when appropriate. *OS_close* was called on valid and invalid file descriptors to ensure that only opened files could be closed. *OS_read* was called on several files such as "CONGRATS.TXT" in root, "SOARING.TXT" in some of the subdirectories of "/PEOPLE/", and several of the photos in "/MEDIA/". These files were read at various byte offsets and for various numbers of bytes in order to check that users could not read past the end of a file and that cluster transitions were also handled correctly.

OS_read was also called on the file "/People/DHO2B/THE-GAME.TXT" which spanned multiple clusters in both the FAT16 and FAT32 volumes. The output for this read made sense linguistically, suggesting that *OS_read* was able to transition between clusters correctly. In addition, *OS_read* was called on this file with an offset greater than the number of bytes per cluster in order to ensure that large offsets were correctly handled.

The structure of the FAT16 and FAT32 volumes were the same. In order to test the FAT32 functions, the FAT16 functions were first checked for correctness as described above on the FAT16 volume. Next, the testing script was run on both the FAT16 and FAT32 volumes, and the outputs for each volume were compared with the *diff* command. When *diff* returned no difference between the two outputs, the FAT32 functionality was determined to be correct.

5. Conclusion

This assignment was successful in teaching file system structure and the use of file allocation tables in keeping track of disk space. I have learned how to use the various structures in the FAT specification to allow for navigation through the raw data of a filesystem and reading on files contained in that filesystem. The required library functions for both FAT16 and FAT32 work as expected and run quickly.

6. Pledge

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

File Name: read_api.h

```
/**
*       Name:            Jonathan Colen
*       Email:           jc8kf@virginia.edu
*       Class:           CS 4414
*       Professor:       Andrew Grimshaw
*       Assignment:      Machine Problem 3
*
*   The purpose of this program is to implement an API for reading a
*   FAT formatted volume. This can be used to navigate through a FAT
*   filesystem.
*
*   This program can be compiled with read_api.c via "make".
*/

#ifndef READ_API_H_
#define READ_API_H_

#include <stdlib.h>
#include <stdint.h>

/**
* Structure representing a directory entry in a FAT filesystem
*/
typedef struct __attribute__((packed)) {
    uint8_t dir_name[11];          //Short name
    uint8_t dir_attr;              //ATTR_READ_ONLY    0x01
                                //ATTR_HIDDEN       0x02
                                //ATTR_SYSTEM       0x04
                                //ATTR_VOLUME_ID    0x08
                                //ATTR_DIRECTORY    0x10
                                //ATTR_ARCHIVE      0x20
                                //ATTR_LONG_NAME    0x0F
    uint8_t dir_NTRes;             //Reserved for Windows NT
    uint8_t dir_crtTimeTenth;      //Millsecond stamp at creation time
    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 entry's first cluster number - 0 for FAT16
    uint16_t dir_wrtTime;      //Time of last write
    uint16_t dir_wrtDate;      //Date of last write
    uint16_t dir_fstClusLO;    //Low word of entry's first cluster numer
    uint32_t dir_fileSize;           //32 bit word holding size in bytes
} dirEnt;


/**
* Changes the current working directory to the specified path
* @param path The absolute or relative path of the file
* @return 1 on success, -1 on failure
*/
int OS_cd(const char * path);

/**
* Opens a file specified by path to be read/written to
* @param path The absolute or relative path of the file
* @return The file descriptor to be used, or -1 on failure
*/
int OS_open(const char * path);

/**
* Close an opened file specified by fd
* @param fd The file descriptor of the file to be closed
* @preturn 1 on success, -1 on failure
*/
int OS_close(int fd);

/**
* Read nbytes of a file from offset into buf
* @param fildes A previously opened file
* @param buf A buffer of at least nbyte size
* @param nbyte The number of bytes to read
* @param offset The offset in the file to begin reading
* @return The number of bytes read, or -1 otherwise
*/
int OS_read(int fildes, void * buf, int nbyte, int offset);

/**
* Gives a list of directory entries contained in a directory
* @param dirname The path to the directory
* @return An array of DIRENTRYs
*/
dirEnt * OS_readDir(const char * dirname);

#endif
```

File Name: read_api.c

```
/**
*       Name:           Jonathan Colen
*       Email:          jc8kf@virginia.edu
*       Class:          CS 4414
*       Professor:      Andrew Grimshaw
*       Assignment:     Machine Problem 3
*
*   The purpose of this program is to implement an API for reading a
*   FAT formatted volume. This can be used to navigate through a FAT
*   filesystem. This particular file implements read operations
*   such as cd, open, read, close, and readDir, which is like ls
*
*   This program can be compiled with read_api.h via "make".
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include "read_api.h"

#define NUM_FD 100

/**
* Structure representing a long directory entry name
*/
typedef struct __attribute__((__packed__)) FAT_LONG_DIRENTRY    {
    char LDIR_Ord;              //Order of this entry in the sequence. 0x4N means last in the sequence
    short int LDIR_Name1[5];    //First 5 2-byte characters in this subcomponent
    char LDIR_Attr;             //Must be ATTR_LONG_NAME
    char LDIR_Type;             //0 implies sub-component of long name
    char LDIR_Chksum;           //Checksum of name in short dir name
    short int LDIR_Name2[6];    //Characters 6-11 of subcomponent
    short int LDIR_FstClusLO;   //Must be ZERO
    short int LDIR_Name3[2];    //Characters 12-13 of subcomponent
} LDIR;

/**
* Structure representing the Bios Partition Block
* NOTE: Without __attribute__((__packed__)), the size of this struct
* is set to 40 instead of 36 and the BPB is not loaded correctly
* NOTE: All values are stored in little-endian format, so they must
* be converted to big_endian upon loading
*/
typedef struct __attribute__((__packed__)) FAT_BS_BPB_Structure {
    char BS_jmpBoot[3];
    char BS_OEMName[8];
    short int BPB_BytsPerSec;   //Number of bytes per sector
    char BPB_SecPerClus;        //Number of sectors per cluster (power of two)
    short int BPB_RsvdSecCnt;   //Number of reserved sectors in the reserved region
    char BPB_NumFATs;           //The number of FAT data structures - 2 for redundancy
    short int BPB_RootEntCnt;   //The number of 32-byte directory entries in the root directory
    short int BPB_TotSec16;     //16-bit total count of sectors on the volume (0 for FAT32)
    char BPB_Media;             //0xF8 is standard value for fixed media. 0xF0 is for removable
    short int BPB_FATSz16;      //16-bit count of sectors occupied by a single FAT (0 for FAT32)
    short int BPB_SecPerTrk;    //Sectors per track for the read operation
    short int BPB_NumHeads;     //Number of heads for the read operation
    int BPB_HiddSec;            //Number of hidden sectors preceding partition containing FAT volume
    int BPB_TotSec32;           //32-bit count of sectors on the volume
} BPB_Structure;

/**
* Extended Boot Record for FAT16 volumes
*/
typedef struct __attribute__((__packed__)) FAT16_BS_EBR_Structure   {
    char BS_DrvNum;
    char BS_Reserved1;
    char BS_BootSig;
    int BS_VolID;
    char BS_VolLab[11];
    char BS_FilSysType[8];
} EBR_FAT16;

/**
* Extended Boot Record for FAT32 volumes
*/
typedef struct __attribute__((__packed__)) FAT32_BS_EBR_Structure   {
    int BPB_FATSz32;
    short int BPB_ExtFlags;
    short int BPB_FSVer;
    int BPB_RootClus;
    short int BPB_FSInfo;
    short int BPB_BkBootSec;
    char BPB_Reserved[12];
    char BS_DrvNum;
    char BS_Reserved1;
    char BS_BootSig;
```

```
    int BS_VolID;
    char BS_VolLab[11];
    char BS_FilSysType[8];
} EBR_FAT32;

/**
* Global variables
*/

int fat_fd = -1;
char fsys_type = 0;         //0x01 for FAT16, 0x02 for FAT32
BPB_Structure bpb_struct;   //Stores the bios partition block once the volume is loaded
EBR_FAT16 ebr_fat16;        //Stores the extended boot record for FAT16 volumes
EBR_FAT32 ebr_fat32;        //Stores the extended boot record for FAT32 volumes
int root_sec;               //Sector of the Root directory
int data_sec;               //Sector of the data section after Root
dirEnt * root_entries;    //Stores the dirENTRY values in the root directory
dirEnt * cwd_entries;     //Represents current working directory

int fd_base[NUM_FD];        //Stores the first cluster number of a file at a given file descriptor
dirEnt fd_dirEnt[NUM_FD]; //Stores the dirENTs opened by a file descriptor

/**
* Given a valid cluster number N, where is the offset in the FAT?
* NOTE: Given the return value FATOffset:
*   FATSecNum = BPB_RsvdSecCnt + (FATOffset / BPB_BytsPerSec);
*   FATEntOffset = FATOffset % BPB_BytsPerSec
* @param clus_nbr The cluster number
* @return The index in the FAT for that cluster number
*/
int offset_in_FAT(int clus_nbr)  {
    int FATSz, FATOffset;
    if (bpb_struct.BPB_FATSz16 != 0)
        FATSz = bpb_struct.BPB_FATSz16;
    else
        FATSz = ebr_fat32.BPB_FATSz32;

    if (fsys_type == 0x01)  //If FAT16
        FATOffset = clus_nbr * 2;
    else
        FATOffset = clus_nbr * 4;

    return FATOffset;
}

/**
* Given a FAT cluster, return the FAT entry at that cluster
* @param cluster The cluster number
* @return The value in the FAT
*/
int value_in_FAT(int cluster)    {
    int offset = offset_in_FAT(cluster);
    //Calculate the sector number and offset in that sector
    int FATSecNum = bpb_struct.BPB_RsvdSecCnt +
        (offset / bpb_struct.BPB_BytsPerSec);
    int FATEntOffset = offset % bpb_struct.BPB_BytsPerSec;

    //Now lseek to the sector and load it into an array
    char sec_buffer[bpb_struct.BPB_BytsPerSec];
    lseek(fat_fd, FATSecNum * bpb_struct.BPB_BytsPerSec, SEEK_SET);
    read(fat_fd, sec_buffer, bpb_struct.BPB_BytsPerSec);

    //Locate the value in the FAT buffer
    int val_FAT;
    if (fsys_type == 0x01)
        val_FAT = *((unsigned short int *) &sec_buffer[FATEntOffset]);
    else
        val_FAT = (*((unsigned int *) &sec_buffer[FATEntOffset])) & 0x0FFFFFFF;

    return val_FAT;
}

/**
* Given a cluster number, how many clusters does it chain to?
* @param cluster The cluster number to be checked
* @return The number of clusters in the chain
*/
int cluster_chain_length(int cluster)   {
    int count = 1;
    int curr = cluster;
    int flag = 1;
    while(flag) {
        int next = value_in_FAT(curr);
        if (fsys_type == 0x01 && next >= 0xFFF8)  {
            flag = 0;
            continue;
        }
        if (fsys_type == 0x02 && next >= 0x0FFFFFF8)  {
            flag = 0;
            continue;
```

```
        }
        curr = next;
        count ++;
    }

    return count;
}

/**
* Read in directory entries from a cluster and follow the cluster chain
* @param cluster The cluster number to be read
* @return The list of directory entries in a cluster
*/
dirEnt * read_cluster_dirEnt(int cluster)    {
    int curr = cluster;
    int sector = (curr - 2) * bpb_struct.BPB_SecPerClus + data_sec;

    //cluster number of 0 is the root directory
    if (curr == 0)  {
        if (fsys_type == 0x01)  {
            sector = root_sec;
        } else if (fsys_type == 0x02)   {
            curr = ebr_fat32.BPB_RootClus;
            sector = (curr - 2) * bpb_struct.BPB_SecPerClus + data_sec;
        }
    }

    int chain_length = cluster_chain_length(cluster);

    int num_entries = chain_length * bpb_struct.BPB_SecPerClus *
        bpb_struct.BPB_BytsPerSec / sizeof(dirEnt);

    dirEnt * entries = (dirEnt *) malloc(sizeof(dirEnt) * num_entries);
    lseek(fat_fd, sector * bpb_struct.BPB_BytsPerSec, SEEK_SET);
    int entry_count = 0;    //Tracks the number of entries read in
    int cluster_count = 0;  //Tracks the number of entries in the current cluster
    while(entry_count < num_entries)    {
        read(fat_fd, (char*)&(entries[entry_count]), sizeof(dirEnt));
        if(((char*)&entries[entry_count])[0] == 0)  //First byte 0 means no more
            break;
        entry_count ++;
        cluster_count ++;
        //If we've read past the end of the cluster, we need to follow the chain
        if ((cluster_count * sizeof(dirEnt)) >=
            (bpb_struct.BPB_BytsPerSec * bpb_struct.BPB_SecPerClus))    {
            curr = value_in_FAT(curr);
            if (fsys_type == 0x01 && curr >= 0xFFF8)
                break;
            if (fsys_type == 0x02 && curr >= 0x0FFFFFF8)
                break;
            sector = (curr - 2) * bpb_struct.BPB_SecPerClus + data_sec;
            lseek(fat_fd, sector * bpb_struct.BPB_BytsPerSec, SEEK_SET);
            cluster_count = 0;
        }
    }

    return entries;
}

/**
* Initialize the FAT volume and load all relevant data
*/
int init_fat()   {
    char * basedir = getenv("FAT_FS_PATH");   //Directory of FAT16 volume
    if (basedir == NULL)    {
        return -1;
    }

    fat_fd = open(basedir, O_RDWR, 0);    //Store the file descriptor

    if (fat_fd == -1)   {
        return -1;
    }

    //Read in the BPB_Structure
    read(fat_fd, (char*)&bpb_struct, sizeof(BPB_Structure));
    read(fat_fd, (char*)&ebr_fat16, sizeof(EBR_FAT16)); //Load EBR for FAT16
    lseek(fat_fd, sizeof(BPB_Structure), SEEK_SET);    //Reset offset for FAT32
    read(fat_fd, (char*)&ebr_fat32, sizeof(EBR_FAT32)); //Load EBR for FAT32

    //Determine FAT16 or FAT32
    //Number of sectors occupied by root directory
    int RootDirSectors = ((bpb_struct.BPB_RootEntCnt * 32) +
        (bpb_struct.BPB_BytsPerSec -1)) / bpb_struct.BPB_BytsPerSec;


    int FATSz, TotSec;
    if (bpb_struct.BPB_FATSz16 != 0)
        FATSz = bpb_struct.BPB_FATSz16;
    else
```

```
        FATSz = ebr_fat32.BPB_FATSz32;

    if (bpb_struct.BPB_TotSec16 != 0)
        TotSec = bpb_struct.BPB_TotSec16;
    else
        TotSec = bpb_struct.BPB_TotSec32;

    int DataSec = TotSec - (bpb_struct.BPB_RsvdSecCnt +
        (bpb_struct.BPB_NumFATs * FATSz) + RootDirSectors);
    int CountofClusters = DataSec / bpb_struct.BPB_SecPerClus;

    if (CountofClusters < 4085) {   //Volume is FAT12 - exit
        return -1;
    } else if (CountofClusters < 65525) {   //Volume is FAT16
        fsys_type = 0x01;
    } else  {   //Volume is FAT32
        fsys_type = 0x02;
    }

    //Initialize more global variables
    root_sec = bpb_struct.BPB_RsvdSecCnt +
        (bpb_struct.BPB_NumFATs * FATSz);
    data_sec = root_sec + RootDirSectors;

    //Load in root directory
    if (fsys_type == 0x01)  {   //FAT16
        root_entries = (dirEnt *) malloc(sizeof(dirEnt) *
            bpb_struct.BPB_RootEntCnt);
        lseek(fat_fd, root_sec * bpb_struct.BPB_BytsPerSec, SEEK_SET);
        int entry_count = 0;
        while (entry_count < bpb_struct.BPB_RootEntCnt)  {
            read(fat_fd, (char*)&(root_entries[entry_count]), sizeof(dirEnt));
            if (((char*)&root_entries[entry_count])[0] == 0)    //First byte 0 means no more to read
                break;
            entry_count ++;
        }
    } else if (fsys_type == 0x02)   {   //FAT32
        //Root is read like any other cluster
        root_entries = read_cluster_dirEnt(ebr_fat32.BPB_RootClus);
    }

    //Initialize cwd to root
    cwd_entries = root_entries;

    //Free all file descriptors except for 0, 1 (Those are stdin, stdout by convention)
    int i;
    for(i = 0; i < NUM_FD; i ++)    {
        fd_base[i] = -1;
    }

    fd_base[0] = 0;
    fd_base[1] = 0;

    return 1;
}

/**
* Get the first element in the path
* For example, /home/grimshaw/data -> home
* @param dest The pointer in which the element should be placed
* @param path The path to be analyzed
* @return The first element in the path
*/
char * getFirstElement(char * dest, const char * path)   {
    if (path == NULL || strlen(path) == 0)
        return NULL;

    char * slash1 = strchr(path, '/');
    if (slash1 == NULL) {
        strcpy(dest, path);
        return dest;
    }


    if (slash1 - path == 0)    {
        char * slash2 = strchr(slash1 + 1, '/');
        if (slash2 == NULL) {
            strcpy(dest, path + 1);
            return dest;
        }
        strncpy(dest, slash1 + 1, slash2 - slash1 - 1);
        dest[slash2-slash1 - 1] = '\0';
        return dest;
    }
    strncpy(dest, path, slash1-path);
    dest[slash1-path] = '\0';
    return dest;
}

/**
```

```
* Get the remaining elements in the path
* For example, /home/grimshaw/data -> grimshaw/data
* @param dest The point in which the remaining elements should be placed
* @param path The path to be analyzed
* @return A pointer to dest
*/
char * getRemaining(char * dest, const char * path)  {
    if (path == NULL || strlen(path) == 0)
        return NULL;

    char * slash1 = strchr(path, '/');
    if (slash1 == NULL)  {
        return NULL;
    }

    if (slash1 - path == 0) {
        slash1 = strchr(slash1 + 1, '/');
    }

    if (slash1 == NULL) {
        return NULL;
    }

    int num_cpy = strlen(path) - (slash1 + 1 - path);
    strncpy(dest, slash1 + 1, num_cpy);
    dest[num_cpy] = '\0';
    return dest;
}

void wide_strcat(char * dest, const short int * source, int len) {
    char buff[len + 1];
    buff[len] = '\0';
    int i;
    for(i = 0; i < len; i ++)   {
        if(source[i] == 0x0000 || source[i] == -1)  {   //Padded with -1 or null terminated
            buff[i] = '\0';
            break;
        }
        buff[i] = (char)(source[i]);
    }
    strcat(dest, buff);
}

/**
* Find a directory entry matching a desired name.
* @param dest The destination for the matching directory entry
* @param current The current set of directory entries
* @param name The name to be matched
* @param directory 1 if the entry searched for must be a directory
* @return 1 if it is found, 0 otherwise
*/
int findDirEntry(dirEnt * dest, const dirEnt * current, char * name, int directory)    {
    int i = 0;
    char * lfilename = NULL;
    while(1)    {
        dirEnt de = current[i];
        i ++;
        if (de.dir_name[0] == 0)    //No more entries in directory
            break;
        if (de.dir_name[0] == 0xE5) //Unused entry
            continue;

        if (de.dir_attr == 0x0F)    {   //Long filename
            //Read in long filename
            if(lfilename == NULL)   {
                lfilename = malloc(sizeof(char) * 255); //FAT spec says max length is 255
                lfilename[0] = '\0';    //Null terminate for concatenations
            }

            LDIR ldir = *(LDIR*)&de;
            wide_strcat(lfilename, ldir.LDIR_Name1, 5);
            wide_strcat(lfilename, ldir.LDIR_Name2, 6);
            wide_strcat(lfilename, ldir.LDIR_Name3, 2);
            continue;
        }

        if (lfilename == NULL)  {   //Parse the short filename
            lfilename = malloc(sizeof(char) * 11);
            lfilename[0] = '\0';    //Null terminate for concatenations
            char * padding = strchr((char*)(de.dir_name), 0x20); //beginning of padding
            if (padding == NULL || padding - (char*)(de.dir_name) > 8)
                strncat(lfilename, (char*)(de.dir_name), 8);
            else
                strncat(lfilename, (char*)(de.dir_name), padding - (char*)(de.dir_name));
            padding = strchr((char*)(de.dir_name) + 8, 0x20); //beginning of padding in extension
            if (padding != (char*)(de.dir_name) + 8)
                strcat(lfilename, ".");
            if (padding == NULL)
                strncat(lfilename, (char*)(de.dir_name) + 8, 3);
            else
```

```
                strncat(lfilename, (char*)(de.dir_name) + 8, padding - (char*)(de.dir_name) - 8);
        }

        if(strcmp(lfilename, name) == 0)   {   //Filename match
            if (!directory || (directory && de.dir_attr & 0x10))    {
                free(lfilename);
                *dest = de;
                return 1;
            }
        }

        free(lfilename);
        lfilename = NULL;
    }

    return 0;
}

/**
* Given a directory name and a current working directory, locate
* a list of entries contained in the named directory.
* @param path The path to the directory
* @param current The current directory
* @return A list of directory entries, or NULL if it doesn't exist
*/
dirEnt * findDir(const char * path, const dirEnt * current)   {
    if (path == NULL || strlen(path) == 0)   //Base case
        return NULL;

    char *first, *remaining;
    first = (char*) malloc(sizeof(char) * strlen(path));
    remaining = (char*) malloc(sizeof(char) * strlen(path));
    first = getFirstElement(first, path);
    remaining = getRemaining(remaining, path);

    //Locate first element in current directory
    dirEnt dir_Ent;
    if (!findDirEntry(&dir_Ent, current, first, 1)) {
        return NULL;
    }

    //Load in next directory and recurse down
    int cluster = (dir_Ent.dir_fstClusHI << 2) | dir_Ent.dir_fstClusLO;
    dirEnt * next_dir = read_cluster_dirEnt(cluster);
    dirEnt * ret = findDir(remaining, next_dir);
    int no_more_path = (remaining == NULL || strlen(remaining) == 0);
    free(first);
    free(remaining);
    if (ret == NULL && no_more_path)    {
        return next_dir;
    }
    free(next_dir);
    return ret;
}

/**
* Changes the current working directory to the specified path
* @param path The absolute or relative path of the file
* @return 1 on success, -1 on failure
*/
int OS_cd(const char * path)    {
    if(path == NULL || strlen(path) == 0)    {
        return -1;
    }

    if (fat_fd == -1)    {   //If directory hasn't been loaded yet
        int err = init_fat();   //Load the FAT volume
        if (err == -1)
            return -1;
    }

    dirEnt * current = OS_readDir(path);
    if (current == NULL)
        return -1;

    if (cwd_entries != root_entries && cwd_entries != current)
        free(cwd_entries);

    cwd_entries = current;
    return 1;
}

/**
* Opens a file specified by path to be read/written to
* @param path The absolute or relative path of the file
* @return The file descriptor to be used, or -1 on failure
*/
int OS_open(const char * path)  {
    if (fat_fd == -1)    {   //If directory hasn't been loaded yet
        int err = init_fat();   //Load the FAT volume
```

```
        if (err == -1)
            return -1;
    }

    //Get the file name and the path name
    char * filename = malloc(sizeof(char) * strlen(path));
    char * pathname = malloc(sizeof(char) * strlen(path));
    char * buff = malloc(sizeof(char) * strlen(path));
    pathname = getRemaining(pathname, path);
    filename = getFirstElement(filename, path);

    while (pathname != NULL) {
        buff = pathname;
        filename = getFirstElement(filename, buff);
        pathname = getRemaining(pathname, buff);
    }

    free(buff);

    pathname = malloc(sizeof(char) * strlen(path));
    int pathlen = strstr(path, filename) - path;
    strncpy(pathname, path, pathlen);
    pathname[pathlen] = '\0';

    dirEnt * current = OS_readDir(pathname);

    dirEnt file;
    if (!findDirEntry(&file, current, filename, 0))
        return -1;

    free(current);  //Avoid memory leaks

    //Find the first available file descriptor
    int fd = 0;
    while (fd < NUM_FD && fd_base[fd] != -1)
        fd ++;

    fd_base[fd] = (file.dir_fstClusHI << 2) | (file.dir_fstClusLO);
    fd_dirEnt[fd] = file;

    return fd;
}

/**
* Close an opened file specified by fd
* @param fd The file descriptor of the file to be closed
* @preturn 1 on success, -1 on failure
*/
int OS_close(int fd) {
    if (fat_fd == -1)    {   //If directory hasn't been loaded yet
        int err = init_fat();   //Load the FAT volume
        if (err == -1)
            return -1;
    }

    if (fd < 0 || fd > NUM_FD || fd_base[fd] == -1)
        return -1;


    fd_base[fd] = -1;

    return 1;
}

/**
* Read nbytes of a file from offset into buf
* @param fildes A previously opened file
* @param buf A buffer of at least nbyte size
* @param nbyte The number of bytes to read
* @param offset The offset in the file to begin reading
* @return The number of bytes read, or -1 otherwise
*/
int OS_read(int fildes, void * buf, int nbyte, int offset)  {
    if (fat_fd == -1)    {   //If directory hasn't been loaded yet
        int err = init_fat();   //Load the FAT volume
        if (err == -1)
            return -1;
    }

    if (fildes < 0 || fildes > NUM_FD || fd_base[fildes] == -1)
        return -1;

    int bytesPerClus = bpb_struct.BPB_SecPerClus * bpb_struct.BPB_BytsPerSec;

    //Calculate offset in cluster and how many links on cluster chain to follow
    int cluster_offset = offset % bytesPerClus;
    int cluster_num = offset / bytesPerClus;
    int cluster = fd_base[fildes];

    int count = 0;  //Count tracks the number of cluster chains reached
```

```
    while(count < cluster_num)  {
        int next = value_in_FAT(cluster);
        if (fsys_type == 0x01 && cluster >= 0xFFF8)
            break;
        if (fsys_type == 0x02 && cluster >= 0x0FFFFFF8)
            break;
        cluster = next;
        count ++;
    }

    if (count < cluster_num)    //Reached end of cluster chain before offset
        return -1;

    int bytesRead = 0;   //Tracks the number of bytes read
    count = -1;

    //Seek to sector of cluster and cluster offset
    int sector = (cluster - 2) * bpb_struct.BPB_SecPerClus + data_sec;
    lseek(fat_fd, sector * bpb_struct.BPB_BytsPerSec + cluster_offset, SEEK_SET);

    int untilEOF = fd_dirEnt[fildes].dir_fileSize - bytesRead - offset;
    int untilEOC = bytesPerClus - cluster_offset - bytesRead;

    //Break if we've read the number of bytes or we count 0 bytes read -> end of file
    while (bytesRead < nbyte && count != 0 && untilEOF > 0)  {
        //Read either nbytes or until end of cluster or until end of file
        if (untilEOF < untilEOC && untilEOF < (nbyte - bytesRead))  {
            count = read(fat_fd, buf + bytesRead, untilEOF);
        } else if (untilEOF > (nbyte - bytesRead) && (nbyte - bytesRead) < untilEOC)    {
            count = read(fat_fd, buf + bytesRead, nbyte - bytesRead);
        } else  {
            count = read(fat_fd, buf + bytesRead, untilEOC);
            //Go to next cluster
            cluster = value_in_FAT(cluster);
            if (fsys_type == 0x01 && cluster >= 0xFFF8)
                break;
            if (fsys_type == 0x02 && cluster >= 0x0FFFFFF8)
                break;
            sector = (cluster - 2) * bpb_struct.BPB_SecPerClus + data_sec;
            lseek(fat_fd, sector * bpb_struct.BPB_BytsPerSec, SEEK_SET);
            cluster_offset = 0;
            untilEOC = bytesPerClus + count;
        }
        bytesRead += count;
        untilEOF -= count;
        untilEOC -= count;
    }

    return bytesRead;
}

/**
* Gives a list of directory entries contained in a directory
* @param dirname The path to the directory
* @return An array of dirEnts
*/
dirEnt * OS_readDir(const char * dirname)  {
    if (fat_fd == -1)    {   //If directory hasn't been loaded yet
        int err = init_fat();   //Load the FAT volume
        if (err == -1)
            return NULL;
    }

    dirEnt * current = cwd_entries;

    const char * truncated_dirname;

    //If absolute path name start path at /
    if (dirname[0] == '/') {
        current = root_entries;
        truncated_dirname = dirname + 1;
    } else  {
        truncated_dirname = dirname;
    }

    dirEnt * ret = findDir(truncated_dirname, current);
    if (ret == NULL && strlen(truncated_dirname) == 0)  {
        return current;
    }
    else if (ret == NULL)   {
        return NULL;
    }
    return ret;
}
```