Writeup:

Comments:

Code:

CS 4414 Operating Systems – Fall 2017

Homework 4: FAT Write Library

Jonathan Colen jc8kf@virginia.edu

The assignment was completed and runs correctly for both FAT16 and FAT32. I used my own implementation of Machine Problem 3 in this assignment.

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 library implemented for this homework had to implement functions to create and remove directories, create and remove files, and write to existing files contained in a FAT16 or FAT32 formatted volume. The library also had to support the FAT read functions implemented in a prior assignment.

2. Approach

The library created for this assignment supports five new functions. *OS_mkdir* creates a new directory. *OS_rmdir* removes an empty directory. *OS_creat* creates a new file and *OS_rm* removes a file. *OS_write* writes data to a previously opened file specified by a file descriptor. The file descriptor is obtained by called *OS_open*, which was implemented in Machine Problem 3. *OS_readDir* and *OS_open* were modified slightly to simplify the write operations supported by this library. *OS_readDir* now sets a global variable called *readDir_cluster* which stores the most recent cluster read by this function. *OS_open* previously stored the directory entry for an opened file in an array at an index equal to the file descriptor. It now also stores the first cluster number of the opened file's parent directory. Since *OS_open* calls *OS_readDir* to search for files at the provided path, the parent cluster number of the opened file is obtained by looking at *readDir_cluster*. This addition allows the directory entries to be easily modified after calling

OS_write. In addition, the initialization method now iterates through each cluster number and sets a global variable called *available_clusters*, which stores how many free clusters exist in the FAT.

All of the functions require use of two helper methods: *write_cluster* and *write_dirEnt*. The *write_cluster* method accepts the same arguments as *OS_write*, but instead of a file descriptor, it accepts a cluster number. First, the function uses the current file size and desired number of bytes to determine how many new clusters will be required. If this number is greater than the number of available clusters in the system, the method terminates. The function uses the FAT table and the offset to find which cluster to begin writing to and at what offset. It then writes until the desired number of bytes has been written or it reaches the end of the cluster. If it reaches the end of the cluster, it calls a function *find_free_cluster*, which iterates through cluster numbers until one is found with a FAT entry of 0, indicating that it is free. It then uses a method called *set_cluster_value* to set the value in the FAT for the current cluster to the next cluster number, and the value for the next cluster number to the "end of cluster chain" value. The *set_cluster_value* method writes to the FAT as specified by the FAT specification provided with Machine Problem 3. The *write_cluster* function continues writing, allocating new clusters as needed, until it has written the specified number of bytes, after which it terminates.

The *write_dirEnt* function overwrites a directory entry in a given cluster or creates a new one. It accepts a cluster number and dirEnt structure as arguments. It first calls a helper function *find_dirEnt_match*, which gives the index of the directory entry at the specified cluster, or -1 times the number of entries in the cluster if the entry is not found. If the entry was found, *write_cluster* is called on the cluster number with the dirEnt structure as the buffer, *sizeof(dirEnt)* as the number of bytes to write, and *index * sizeof(dirEnt)* as the offset. If no match is found, the index is negative, and *write_cluster* is called twice. The first call uses the same arguments as when the match is found, but with *-index * sizeof(dirEnt)* as the offset. The second call writes 0x00 at *-(index+1) * sizeof(dirEnt)* in order to null terminate the list of directory entries stored at that cluster.

OS_creat and *OS_mkdir* require creation of a new directory entry inside a parent directory file. To do this, both methods call a helper function called *create_new_dirEnt*. This function accepts a string path and a character attribute. The attribute is set to 0x10 for *OS_mkdir* and 0x20 for *OS_creat*. The function first separates the path to the new entry into *pathname* (the

path to the parent directory of the new file) and *filename* (the name of the new file). For example, "/People/PRF4V/Soaring.txt" would result in a *pathname* of "/People/PRF4V" and a *filename* of "Soaring.txt". The function then calls *OS_readDir* on *pathname* to get the current list of entries in the parent directory. If the pathname is invalid, *OS_readDir* will return NULL and *create_new_dirEnt* will return -1. If an entry with the same name as *filename* is found in the list of entries, the function returns -2. Otherwise, a new dirEnt structure is created with a name equal to *filename* in the 8.3 format, a file size of 0, and an attribute equal to that specified by the function argument (0x10 or 0x20). In addition, the creation date and time are set using the C "time.h" library. Finally, the dirEnt cluster number is set using *find_free_cluster* and this cluster's entry in the FAT is set to the "end of chain" value using *set_cluster_value*. This new dirEnt is written at the cluster last accessed by *OS_readDir*, stored in *readDir_cluster*, using the *write_dirEnt* function. The number of available clusters is decremented. If the attribute argument to *create_new_dirEnt* is equal to 0x10, signifying a directory, two new directory entries are written to the new dirEnt's cluster. One has the name "." and has the same cluster number as the new dirEnt, and the other has the name ".." and has the same cluster number as the new dirEnt's parent cluster.

OS_rm and *OS_rmdir* require removal of a directory entry and deallocation in the FAT. Both methods call a helper function called *remove_dirEnt*. This function accepts the same arguments as *create_new_dirEnt*, a path string and an attribute. The path string is once again separated into a *pathname* and a *filename*, and *OS_readDir* is called on *pathname*. If this path turns out to be invalid or *filename* is not contained in the resulting list of directory entries, the function returns -1. If a file matching *filename* is found but it does not have a *dir_attr* field matching the attribute argument, the function returns -2 signifying incompatible types. If the attribute is 0x10, signifying a directory, the entries in that directory are checked. If any entries except for "." and ".." exist, the function returns -3, as the directory is not empty. If no problems are found, the function checks the cluster of *filename*'s parent, stored in *readDir_cluster*, and determines the index of the matched directory entry using *find_dirEnt_match*. It then modifies the matched entry by setting the first bit to 0xE5, signifying an empty entry. Next, the function calls *write_cluster* on the parent cluster with the modified entry and an offset given by *index * sizeof(dirEnt)*. Finally, the FAT entry for the file's cluster value is set to 0, signifying a free cluster, and the number of available clusters is incremented.

OS_write is a wrapper around *write_cluster* with a slight modification. First, the cluster number of the opened file is found by checking the array of file information using the file descriptor as the index. The function calls *write_cluster* on this cluster number with the buffer, number of bytes, and offset arguments equal to those passed to *OS_write*. If the offset and number of bytes written results in an increase in file size, the directory entry of the opened file is modified to reflect this change. In addition, the write date and time of the entry is modified using the C time library. Finally, the modified directory entry is written to the cluster of the opened file's parent directory using *write_dirEnt*. This is possible due to the modification of *OS_open* mentioned earlier. Since *OS_open* now stores the parent directory cluster numbers in addition to the file information, modification of the file directory entries is very straightforward.

3. Problems Encountered

The most difficult problem to address was that of modifying directory entries of opened files. This problem arose because the *OS_read* function stops reading after the number of bytes reaches the file size. In order to read modifications, the *OS_write* function had to also update the directory entry. This problem was solved by modifying *OS_open* to store the parent directory entry, which simplified later writes. The most straightforward way to do this was to have *OS_readDir* set a global variable equal to the cluster being accessed for a given path, so that other functions could modify the list of entries at that path. The modification of *OS_readDir* and *OS_open* to accommodate this problem in *OS_write* also solved later problems related to changing directory entries in *OS_rm* and *OS_rmdir*.

Another problem that arose was that of finding the offset in the cluster for a directory entry to be modified. Because deleting a file resulted in the directory entry being set to 0xE5, and since *OS_readDir* ignored entries with that value, finding the offset was more complicated than getting the index of the entry in the list returned by *OS_readDir*. A separate method called *find_dirEnt_match* was written to solve this problem by reading directly from the cluster, resulting in minimal changes to the existing FAT read functionality.

4. Testing

This library was tested in several ways. First, the test script written for Machine Problem 3 was used for both FAT16 and FAT32. The results for the new library were compared with the results of the old library to ensure no functionality was lost by modifying *OS_readDir* and

OS_open. A new testing script was also written to check the new write functionality. *OS_mkdir, OS_rmdir, OS_rm,* and *OS_creat* were called on a variety of valid and invalid filenames, existing and nonexistent files, and empty and non-empty directories. The return codes were checked to ensure that the functions exited correctly. *OS_mkdir* was also checked by calling *OS_readDir* into the new directory and ensuring that "." and ".." were contained inside. In addition, paths such as "newdir/./../" were validated using *OS_cd*. *OS_creat*, *OS_rm,* and *OS_rmdir* were tested by checking the various directories using *OS_readDir* and ensuring files were created and deleted as necessary. Finally, *OS_write* was checked by writing to both created and existing files. Cluster management was verified by appending the contents of "/People/DHO2B/THE-GAME.TXT" to itself, requiring the allocation of a new cluster in *OS_write*. The contents of the written files were verified using *OS_read*.

When all functionality for the implemented methods was confirmed for FAT16, the output of the test script was compared for both FAT16 and FAT32 using the *diff* feature. When *diff* returned no difference between the two outputs, the FAT32 functionality was determined to be correct.

5. Conclusions

The assignment was successful in teaching file system structure and the use of file allocation tables in managing disk space. I have learned how the use of the various structures in the FAT specification allow for navigation and modification of the raw data in a file system. The required read and write 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.

Sources:

File: fat_api.h

```
/**
*       Name:           Jonathan Colen
*       Email:          jc8kf@virginia.edu
*       Class:          CS 4414
*       Professor:      Andrew Grimshaw
*       Assignment:     Machine Problem 4
*
*   The purpose of this program is to implement an API for operating on
*   a FAT formatted volume. This can be used to navigate through and modify
*   a FAT filesystem.
*
*   This program can be compiled with fat_api.c via "make".
*/

#ifndef FAT_API_H_
#define FAT_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);

/**
* Creates a new directory at the specified path
* @param path The path to the directory
* @return 1 if created, -1 if the path is invalid, -2 if final
*   path element already exists
*/
int OS_mkdir(const char * path);
```

```
/**
* Remove a directory at the specified path
* @param path The path to the directory
* @return 1 if removed, -1 if path is invalid,
*   -2 if path does not refer to a file,
*   -3 if directory is not empty
*/
int OS_rmdir(const char * path);

/**
* Remove a file at the specified path
* @param path The path to the file
* @return 1 if removed, -1 if path is invalid,
*   -2 if file is a directory
*/
int OS_rm(const char * path);

/**
* Create a file at a desired path name
* @param path The path to the file
* @return 1 if file is created, -1 if path is invalid,
*   -2 if final path element already exists
*/
int OS_creat(const char * path);

/**
* Write to an opened file
* @param fildes The file descriptor
* @param buf The buffer of bytes to be written
* @param nbytes The number of bytes to write
* @param offset The offset at which to write
* @return The number of bytes written, or -1 on failure
*/
int OS_write(int fildes, const void * buf, int nbytes, int offset);

#endif
```

File: fat_api.c

```
/**
*       Name:           Jonathan Colen
*       Email:          jc8kf@virginia.edu
*       Class:          CS 4414
*       Professor:      Andrew Grimshaw
*       Assignment:     Machine Problem 4
*
*   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 the operations
*   such as cd, open, read, close, readDir, write, mkdir, rm, rmdir,
*   and creat.
*
*   This program can be compiled with fat_api.h via "make".
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include "fat_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
int CountofClusters;        //Stores the number of clusters after cluster 2
dirEnt * root_entries;    //Stores the dirENTRY values in the root directory
dirEnt * cwd_entries;     //Represents current working directory
char cwd_path[1024];
int cwd_cluster;            //Stores the cluster that the current working directory was read from

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
char * fd_path[NUM_FD];    //Stores the paths to the file opened by each file descriptor
int fd_parent_cluster[NUM_FD];  //Stores the cluster number of the parent directory for an opened file

int available_clusters;     //Stores the number of available clusters
int readDir_cluster;        //Stores the cluster number read by the current call to OS_readDir

/**
* Get the current time and store the date in date and the time in
* time. The format, according to FAT spec, is
*   Date:           Time:
*   Year    7       Hour    5
*   Month   4       Minute  6
*   Day     5       Second  5   (seconds / 2)
*/
void get_date_time(short int * date, short int * tim)  {
    //According to cppreference.com, this is the number of seconds since the epoch
    //The epoch for unix is apparently January 1, 1970
    time_t t = time(NULL);
    struct tm * tinfo = localtime(&t);
    *date = (tinfo->tm_mday & 0xfffff) |
            (((tinfo->tm_mon + 1) & 0xffff) << 5) |     //struct tm tm_mon is months since January
            (((tinfo->tm_year-80) & 0xfffffff) << 9);   //struct tm tm_year is years since 1900
    *tim = ((tinfo->tm_sec / 2) & 0xfffff) |
            ((tinfo->tm_min & 0xffffff) << 5) |
            ((tinfo->tm_hour & 0xfffff) << 11);
}

/**
* 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 FATOffset;
    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;
}

/**
* Find the first open cluster by searching the FAT
* @return The cluster number, or -1 on failure
*/
int find_free_cluster() {
   int i;
   for (i = 0; i < CountofClusters; i ++)   {
        if (value_in_FAT(i) == 0)
            return i;
   }

   return -1;
}


/*
* Set the FAT table entry for a given cluster to a specified value
* @return 1 on success, -1 on failure
*/
int set_cluster_value(int cluster, int value)   {
    if (cluster < 0 || cluster >= CountofClusters)
        return -1;
    int offset;
    if (fsys_type == 0x02 && cluster == 0)
        offset = offset_in_FAT(ebr_fat32.BPB_RootClus);
    offset = offset_in_FAT(cluster);

    int FATSecNum = bpb_struct.BPB_RsvdSecCnt +
        (offset / bpb_struct.BPB_BytsPerSec);
    int FATEntOffset = offset % bpb_struct.BPB_BytsPerSec;

    if (fsys_type == 0x01)   {
        lseek(fat_fd, FATSecNum * bpb_struct.BPB_BytsPerSec +
            FATEntOffset, SEEK_SET);
        unsigned short int val = (unsigned short int) (value & 0xFFFF);
        write(fat_fd, (void*)(&val), sizeof(unsigned short int));
    } else if (fsys_type == 0x02)    {
        unsigned int curr_entry = value_in_FAT(cluster); //Need to keep first 4 bits same if FAT32
        unsigned int new_entry = (curr_entry & 0xF0000000) | (value & 0x0FFFFFFF);
        lseek(fat_fd, FATSecNum * bpb_struct.BPB_BytsPerSec +
            FATEntOffset, SEEK_SET);
        write(fat_fd, (void*)&new_entry, sizeof(unsigned int));
    }

    return 1;
}

/**
* 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;

    readDir_cluster = cluster;

    //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;
            readDir_cluster = curr;
            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);
    dirEnt curr_entry;
    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*)&curr_entry, sizeof(dirEnt));
        if (curr_entry.dir_name[0] != 0xE5) {   //Store non-free entries
            entries[entry_count] = curr_entry;
            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);
    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
        cwd_cluster = 0;
        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
        cwd_cluster = ebr_fat32.BPB_RootClus;
        root_entries = read_cluster_dirEnt(ebr_fat32.BPB_RootClus);
    }

    //Initialize cwd to root
    cwd_entries = root_entries;
    strcpy(cwd_path, "/");

    //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;

    //Initialize number of empty clusters
    available_clusters = 0;
    for (i = 0; i < CountofClusters; i ++)  {
         if (value_in_FAT(i) == 0)
            available_clusters ++;
    }

    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;
}

/**
* Separate a path to a file into the filename and the pathname
* @param filename Where the filename will be stored
* @param pathname Where the pathname will be stored
* @param path The path to be split up
*/
void separate_path(char * filename, char * pathname, const char * path) {
    char * buff = malloc(sizeof(char) * strlen(path));
    char * pathbuff = malloc(sizeof(char) * strlen(path));
    char * filebuff = malloc(sizeof(char) * strlen(path));

    pathbuff = getRemaining(pathbuff, path);
    filebuff = getFirstElement(filebuff, path);

    while(pathbuff != NULL && strlen(pathbuff) != 0) {
        buff = pathbuff;
        filebuff = getFirstElement(filebuff, buff);
        pathbuff = getRemaining(pathbuff, buff);
    }

    free(buff);
    pathbuff = malloc(sizeof(char) * strlen(path));
    int pathlen = strstr(path, filebuff) - path;
    strncpy(pathbuff, path, pathlen);
    pathbuff[pathlen] = '\0';
    strcpy(pathname, pathbuff);
    strcpy(filename, filebuff);
}

/**
* Translate an 8.3 formatted filename into an 11 character string
* @param dest The place for the translated name to be stored
* @param source The 8.3 formatted filename
*/
void fill_dir_name(char * dest, const char * source)    {
    char * period = strchr(source, '.');
    if (period != NULL) {
        if (period - source < 8)    {
            strncpy(dest, source, period-source);
            int i;
            for (i = period - source; i < 8; i ++)
                dest[i] = 0x20;  //Pad end of string
        } else  {
            strncpy(dest, source, 8);
        }
        if (strlen(period + 1) < 3) {
            strncpy(dest+8, period + 1, strlen(period + 1));
```

```
            int i;
            for (i = 8 + strlen(period+1); i < 11; i ++)
                dest[i] = 0x20;
        } else  {
            strncpy(dest + 8, period + 1, 3);
        }
    } else  {
        if (strlen(source) < 11)    {
            strncpy(dest, source, strlen(source));
            int i;
            for (i = strlen(source); i < 11; i ++)
                dest[i] = 0x20;
        } else  {
            strncpy(dest, source, 11);
        }
    }
}

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);

    if (path[0] == '/')
        strcpy(cwd_path, path);
    else
        strcat(cwd_path, path);

    cwd_entries = current;
    cwd_cluster = readDir_cluster;
    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);

    if (current == NULL)
        return -1;

    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;
    fd_parent_cluster[fd] = readDir_cluster;

    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;
    free(fd_path[fd]);

    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;
    int at_root = 0;

    //If absolute path name start path at /
    if (dirname[0] == '/') {
        at_root = 1;
        current = read_cluster_dirEnt(0);   //Read in cluster 0, which represents root
        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)   {
        if (at_root)
            free(current);
        return NULL;
    }
    if (at_root)
        free(current);
    return ret;
}

/**
* Write at a cluster number, updating the FAT and going to another cluster if
* necessary.
* @param cluster The cluster number to be written to
* @param buf The buffer of bytes to be written
* @param nbytes The number of bytes to write
* @param offset The offset at which to write
* @return The number of bytes written, or -1 on failure
*/
int write_cluster(int cluster, const void * buf, int nbytes, int offset)  {
    int bytesPerClus = bpb_struct.BPB_SecPerClus * bpb_struct.BPB_BytsPerSec;

    int cluster_offset = offset % bytesPerClus;
    int cluster_num = offset / bytesPerClus;
```

```
    int count = 0;
    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)
        return -1;

    int bytesWritten = 0; //Tracks the number of bytes written
    count = -1;

    //Seek to sector of cluster and cluster offset
    int sector = (cluster - 2) * bpb_struct.BPB_SecPerClus + data_sec;
    if (cluster == 0)   {
        if (fsys_type == 0x01)
            sector = root_sec;
        else if (fsys_type == 0x02)
            sector = (ebr_fat32.BPB_RootClus - 2) * bpb_struct.BPB_SecPerClus + data_sec;
    }
    lseek(fat_fd, sector * bpb_struct.BPB_BytsPerSec + cluster_offset, SEEK_SET);

    int untilEOC = bytesPerClus - cluster_offset - bytesWritten;
    int required_clusters = (nbytes - untilEOC) / bytesPerClus;
    if (available_clusters < required_clusters) //Break if we won't have enough space
        return -1;

    //Only break if we've written the number of bytes required
    while (bytesWritten < nbytes)  {
        if ((nbytes - bytesWritten) < untilEOC) {
            count = write(fat_fd, buf + bytesWritten, nbytes - bytesWritten);
        } else  {
            count = write(fat_fd, buf + bytesWritten, untilEOC);
            //Find next cluster
            int next_cluster = find_free_cluster();
            set_cluster_value(cluster, next_cluster);
            set_cluster_value(next_cluster, -1);
            cluster = next_cluster;
            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;
            available_clusters --;
        }
        bytesWritten += count;
        untilEOC -= count;
    }

    return bytesWritten;
}

/**
* Find the index at which entry is located in the cluster. This includes
* entries of 0xE5.
* @param cluster The cluster to be examined
* @param entry The entry to be searched for
* @return The index, or -1 * the number of entries looked at if it isn't found
*/
int find_dirEnt_match(int cluster, dirEnt entry)    {
    int curr = cluster;
    int sector = (curr - 2) * bpb_struct.BPB_SecPerClus + data_sec;

    readDir_cluster = cluster;

    if (curr == 0)  {
        if (fsys_type == 0x01)  {
            sector = root_sec;
        } else if (fsys_type == 0x02)   {
            curr = ebr_fat32.BPB_RootClus;
            readDir_cluster = curr;
            sector = (curr - 2) * bpb_struct.BPB_SecPerClus + data_sec;
        }
    }

    lseek(fat_fd, sector * bpb_struct.BPB_BytsPerSec, SEEK_SET);
    int entry_count = 0;
    int cluster_count = 0;

    char name[12];
    char currname[12];

    strncpy(name, entry.dir_name, 11);
    name[11] = '\0';
    currname[11] = '\0';
    dirEnt curr_entry;
    while(1)    {
        read(fat_fd, (char*)&curr_entry, sizeof(dirEnt));
        if (curr_entry.dir_name[0] == 0) {
```

```
            break;
        }

        strncpy(currname, curr_entry.dir_name, 11);
        if(strcmp(name, currname) == 0) {   //Match found
            return entry_count;
        }

        entry_count ++;
        cluster_count ++;

        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 -1 * entry_count;
}

/**
* Overwrite a directory entry contained in a given cluster. If a
* directory entry at path does not exist with the same name as entry,
* then a new one will be created.
* @param cluster The cluster
* @param entry The entry to be written in the directory
* @return 1 on success
*/
int write_dirEnt(int cluster, dirEnt entry)   {
    int i = find_dirEnt_match(cluster, entry);
    if (i < 0)  {
        i = -1 * i;
        write_cluster(cluster, (void*)&entry, sizeof(dirEnt), i * sizeof(dirEnt));
        char toWrite = '\0';
        write_cluster(cluster, (void*)&toWrite, sizeof(char), (i+1) * sizeof(dirEnt));
    } else  {
        write_cluster(cluster, (void*)&entry, sizeof(dirEnt), i * sizeof(dirEnt));
    }

    return 1;
}

/**
* Create a new directory entry at the specified path with
* the desired attribute
* @param path The path to the entry
* @param attr The desired dirEnt attribute
* @return 1 if created, -1 if path is invalid, -2 if final
*   path element already exists
*/
int create_new_dirEnt(const char * path, char attr) {
    if (fat_fd == -1)   {
        int err = init_fat();
        if (err == -1)
            return -1;
    }

    char *filename, *pathname;
    pathname = malloc(sizeof(char) * strlen(path));
    filename = malloc(sizeof(char) * strlen(path));
    separate_path(filename, pathname, path);

    dirEnt * current = OS_readDir(pathname);


    if(current == NULL)
        return -1;  //Invalid path

    dirEnt file;
    if(findDirEntry(&file, current, filename, 0))   {
        //File with desired name already exists
        return -2;
    }

    dirEnt toWrite;
    toWrite.dir_attr = attr;
    fill_dir_name(toWrite.dir_name, filename);
    toWrite.dir_fileSize = 0;

    //Find next available cluster to allocate
    int next_cluster = find_free_cluster();
    set_cluster_value(next_cluster, -1);
    available_clusters --;

    toWrite.dir_fstClusHI = (unsigned short int)(next_cluster >> 16);   //Will be 0 for FAT16
    toWrite.dir_fstClusLO = (unsigned short int)(next_cluster & 0xFFFF);
```

```
    get_date_time(&(toWrite.dir_wrtDate), &(toWrite.dir_wrtTime));
    toWrite.dir_crtDate = toWrite.dir_wrtDate;
    toWrite.dir_crtTime = toWrite.dir_wrtTime;

    int parent_cluster = readDir_cluster;
    write_dirEnt(parent_cluster, toWrite);

    //If a directory, need to make . and .. entries
    if (attr & 0x10)    {
        toWrite.dir_name[0] = '.';
        int i;
        for (i = 1; i < 10; i ++)
            toWrite.dir_name[i] = 0x20;
        write_dirEnt(next_cluster, toWrite);

        toWrite.dir_name[1] = '.';
        toWrite.dir_fstClusHI = (unsigned short int)(parent_cluster >> 16);
        toWrite.dir_fstClusLO = (unsigned short int)(parent_cluster & 0xFFFF);

        write_dirEnt(next_cluster, toWrite);
    }

    return 1;
}

/**
* Remove a directory entry and clear the required information from the FAT
* @param path The path to the directory entry to be removed
* @param attr The attribute of the entry to be removed
*   (0x10 for directory, 0x20 for file)
* @return 1 if removed, -1 if path is invalid
    If attr = 0x10
        -2 if path does not refer to a file
        -3 if directory is not empty
    If attr = 0x20
        -2 if file is a directory
*/
int remove_dirEnt(const char * path, char attr) {
    if (fat_fd == -1)   {
        int err = init_fat();
        if (err == -1)
            return -1;
    }

    char *filename, *pathname;
    pathname = malloc(sizeof(char) * strlen(path));
    filename = malloc(sizeof(char) * strlen(path));
    separate_path(filename, pathname, path);

    dirEnt * current = OS_readDir(pathname);

    if (current == NULL)
        return -1;  //Invalid path

    dirEnt file;
    if (!findDirEntry(&file, current, filename, 0)) {
        return -1;  //File does not exist
    }

    //If it's a file, attr & 0x20 will be true
    //If it's a directory, attr & 0x10 will be true
    //rmdir returns -2 if the path doesn't refer to a file
    //rm returns -2 if the path refers to a directory
    if (!(file.dir_attr & attr))    {
        return -2;
    }

    int parent_cluster = readDir_cluster;
    int cluster = (file.dir_fstClusHI << 16) | (file.dir_fstClusLO);

    if (attr & 0x10)    {
        //check if empty
        //Make sure that only . and .. are contained in the directory
        dirEnt * entries = read_cluster_dirEnt(cluster);
        char name1[12], name2[12];
        char currname[12];
        strcpy(name1, ".          ");
        strcpy(name2, "..         ");
        currname[11] = '\0';
        int i = 0;
        while (1)   {
            if (entries[i].dir_name[0] == 0)
                break;
            if (entries[i].dir_name[0] == 0xE5) {
                i ++;
                continue;
            }
            strncpy(currname, entries[i].dir_name, 11);
            if (strcmp(name1, currname) != 0 && strcmp(name2, currname) != 0)   {
                return -3;
            }

            i ++;
```

```
        }
    }

    //Delete directory entry and empty FAT entry
    //Can delete directory entry by changing Name[0] to 0xE5 and overwriting
    //Find index of file in current
    int i = find_dirEnt_match(parent_cluster, file);
    file.dir_name[0] = 0xE5;
    write_cluster(parent_cluster, (void*)&file, sizeof(dirEnt), i * sizeof(dirEnt));
    set_cluster_value(cluster, 0);
    available_clusters ++;

    return 1;
}

/**
* Creates a new directory at the specified path
* @param path The path to the directory
* @return 1 if created, -1 if the path is invalid, -2 if final
*   path element already exists
*/
int OS_mkdir(const char * path) {
    return create_new_dirEnt(path, 0x10);
}

/**
* Remove an empty directory at the specified path
* @param path The path to the directory
* @return 1 if removed, -1 if path is invalid,
*   -2 if path does not refer to a file,
*   -3 if directory is not empty
*/
int OS_rmdir(const char * path) {
    return remove_dirEnt(path, 0x10);
}

/**
* Remove a file at the specified path
* @param path The path to the file
* @return 1 if removed, -1 if path is invalid,
*   -2 if file is a directory
*/
int OS_rm(const char * path)    {
    return remove_dirEnt(path, 0x20);
}

/**
* Create a file at a desired path name
* @param path The path to the file
* @return 1 if file is created, -1 if path is invalid,
*   -2 if final path element already exists
*/
int OS_creat(const char * path) {
    return create_new_dirEnt(path, 0x20);
}

/**
* Write to an opened file
* @param fildes The file descriptor
* @param buf The buffer of bytes to be written
* @param nbytes The number of bytes to write
* @param offset The offset at which to write
* @return The number of bytes written, or -1 on failure
*/
int OS_write(int fildes, const void * buf, int nbytes, int offset)  {
    if (fat_fd == -1)   {
        int err = init_fat();
        if (err == -1)
            return -1;
    }

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

    int bytesWritten = write_cluster(fd_base[fildes], buf, nbytes, offset);

    //Need to now update the file size in its dirEnt
    if (offset + nbytes > fd_dirEnt[fildes].dir_fileSize)
        fd_dirEnt[fildes].dir_fileSize = offset + nbytes;
    get_date_time(&(fd_dirEnt[fildes].dir_wrtDate), &(fd_dirEnt[fildes].dir_wrtTime));
    write_dirEnt(fd_parent_cluster[fildes], fd_dirEnt[fildes]);

    return bytesWritten;
}
```