Write up
Comments
Code

____

Operating Systems Homework #4

Maurice Wong, mzw7af@virginia.edu

The assignment was successfully completed.

1. Problem:

The goal of this assignment was to create a write API on a FAT16 file system. Given a FAT16 file system, the library functions need to navigate the allocation table and the clusters within the system, performing the appropriate write and delete operations. 5 functions were implemented:

- *OS_mkdir(const char *path)* – makes a new directory at the specified path
- *OS_rmdir(const char *path)* – removes the directory at the specified path
- *OS_rm(const char *path)* – removes the file at the specified path
- *OS_creat(const char *path)* – creates a file at the specified path
- *OS_write(int fildes, const void *buf, int nbyte, int offset)* – given a file descriptor to a previously opened file, writes nbytes into the file specified at a given offset.

2. Approach:

The implementation of this assignment was heavily based on the previous work done on Homework #3. Built on top of the code submitted for the previous assignment, a lot of essential structs and helper functions for navigating the FAT file system were already implemented. Bootstrapping off of these functions used to extract information from the boot sector and traversing through file directories, a minimal amount of additional code logic was necessary to implement the write functions required by this project.

However, some additional helper functions were written to aid in specific write-oriented features. Among these were:

- *findFreeCluster()* – scans the FAT table to find the first free cluster available for allocation
- *clearClusterChain()* – given the first cluster number of a file data volume, clears the data regions allocated along the cluster chain and clears the FAT table values
- *isDirEmpty()* – determines if a given a directory is empty (useful for the *OS_rmdir()* function)
- *writeDirEnt()* – writes a *dirEnt* struct to disk
- *setFatValue()* – sets a FAT table entry to a specified value

These helper functions were used throughout the implementation of the 5 API functions. Each of these API functions follows a similar pattern. First it is necessary to traverse through the file system to reach the parent directory of the file or directory that is to be modified or created. Then, depending on what the API function is, a matching directory entry name is searched for and appropriate actions are performed. For example in *OS_mkdir()* and *OS_creat()*, a directory entry with a matching name of the file/directory to create is searched for. If one is found, then errors are returned, but if one is not found, an appropriate entry is added in the first available space. Conversely for *OS_rmdir()* and *OS_rm()*, if a directory with a matching name is found, then that directory entry is cleared, as well as its associated data regions, and errors are returned if the entry cannot be found. These similar functionalities result in similar code structures for each API function's implementation

OS_write() was more complicated to deal with particularly because it required simultaneous writing data to the disk file as well as allocating new clusters in the cluster chain and the FAT table. The structure of this code was based off of the *OS_read()* code implemented from last assignment since they both follow similar patterns of requiring action when reading/writing to the end of a cluster block. The difference in *write()* however was in searching for a free cluster block, allocating space for it in the FAT table, and then proceeding to write in the newly allocated space until all appropriate bytes were written.

An interesting discussion point includes how free cluster blocks are discovered. The documentation specifies that free cluster blocks are denoted with a 0x0000 present in the FAT entry for the corresponding cluster block. Thus the easiest way to find an available cluster block was to simply linearly scan through the FAT table and find the first cluster block that was free. Although this is not incredibly efficient, it was the simplest to implement for the purposes of this assignment.

Another interesting discussion point was the handling of free directory entry spots for each directory. Once directories are removed, 'holes' are introduced in the list of directory entries, and this was solved by marking the first character of the directory name with 0xE5 as specified by the documentation to indicate a free dirEntry space.

3. Problems Encountered

One major issue encountered was with *OS_write()* which was incredibly tricky to implement since it had to simultaneously write to disk while also allocating free clusters when necessary. The initial implementation of *setFatValue()* relied on the helper function *getFatValue()* to check each entry in the FAT table for a free cluster marker. However, the *getFatValue()* function opens a new file descriptor upon each invocation and scans through the FAT table from the beginning until the appropriate FAT entry is found. This resulted in O(n^2) runtime for *setFatValue()* which was horrendously slow, and this was remediated by performing the linear scan directly in *setFatValue()*. The *OS_write()* function was completed only after careful debugging through printing allocated cluster numbers.

Another problem encountered was forgetting to close real file descriptors to the raw disk file. At some points, too many file descriptors were being opened, causing undefined behavior to occur and parts of the disk resetting to strange values. This caused strange behavior in the entries for the FAT table, where some entries that were previously marked as allocated suddenly became marked as free, causing the *write()* function to fail. This was fixed after careful debugging and closing file descriptors when appropriate.

Finally another problem encountered was dealing with the 'holes' in the directory entry lists after remove API functions had been called. In the original implementation for *OS_creat()*

and *OS_mkdir()*, the first available space was utilized for the new dirEntry, but it was realized that a duplicate entry name that came after the free space would not be detected (since this condition should cause the create function to fail). This was remediated by first looping through the whole directory and searching for duplicate names before creating the actual entry.

4. Testing

To test these library functions, a test program was created that linked the shared library, and each function was extensively tested. All API functions with the exception of *OS_write()* require traversing down the file path and have errors associated with files of that name being found/not being found. These different cases were all tested thoroughly such that all error and success conditions were triggered, and the appropriate response was noted. Additionally, the resulting raw file was examined at different stages using a hex editor to ensure that appropriate values were being written in the correct cluster locations and places within the FAT file system.

To perform an integration test, the write API functions were called together with the successful read API functions implemented in the last assignment. Folders and files were created and written to, and then the read API functions attempted to read back the correct information. This was how *OS_write()* was tested by first reading an image file and then writing this data to a newly created file. The data was then read back from the newly created file and saved to verify that the image had been properly copied, written, and read.

5. Conclusion

Creating these write API functions was significantly easier through bootstrapping from the previous code in the previous assignment. The helper functions that aided in navigation around the file system were tremendously useful in being able to start coding functionality sooner for the write API functions. Additionally, having prior familiarity with the system and the documentation was a huge help in understanding what needed to be done for each function. Overall, this was a useful assignment in extending prior knowledge of the FAT file system and carefully manipulating byte-level data without error.

```c
#include <stdint.h>

typedef struct fat_BS {
    unsigned char       bootjmp[3];
    unsigned char       oem_name[8];
    unsigned short      bytes_per_sector;
    unsigned char       sectors_per_cluster;
    unsigned short      reserved_sector_count;
    unsigned char       table_count;
    unsigned short      root_entry_count;
    unsigned short      total_sectors_16;
    unsigned char       media_type;
    unsigned short      table_size_16;
    unsigned short      sectors_per_track;
    unsigned short      head_side_count;
    unsigned int        hidden_sector_count;
    unsigned int        total_sectors_32;

    //this will be cast to it's specific type once the driver actually knows what
     type of FAT this is.
    unsigned char       extended_section[54];

} __attribute__((packed)) fat_BS_t;


typedef struct  dir_ent {
    uint8_t dir_name[11];           // short name
    uint8_t dir_attr;               // File sttribute
    uint8_t dir_NTRes;              // Set value to 0, never chnage this
    uint8_t dir_crtTimeTenth;       // millisecond timestamp for file 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 fo this entry's first cluster
     number
    uint16_t dir_wrtTime;           // time of last write
    uint16_t dir_wrtDate;           // dat eof last write
    uint16_t dir_fstClusLO;         // low word of this entry's first cluster number
    uint32_t dir_fileSize;          // 32-bit DWORD hoding this file's size in bytes
} __attribute__((packed)) dirEnt;

int OS_cd(const char *path);
int OS_open(const char *path);
int OS_close(int fd);
int OS_read(int fildes, void *buf, int nbyte, int offset);
dirEnt *OS_readDir(const char *dirname);

int OS_mkdir(const char *path);
int OS_rmdir(const char *path);
int OS_rm(const char *path);
int OS_creat(const char *path);
int OS_write(int fildes, const void *buf, int nbytes, int offset);
```

```
/**
 * Homework 4: Write API for FAT 16 file system
 *
 * CS4414 Operating Systems
 * Fall 2017
 *
 * Maurice Wong - mzw7af
 *
 * main.c - Write API library for FAT16 file system
 *
 * COMPILE:     make
 * OBJECTS:     libFAT.so
 * To link the shared library, you also need to #include "main.h"
 */

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


static char* cwd = "/"; // file path to the current working directory. Ends with a
 '/'.
static unsigned short fd_table[128]; // file descriptor table tracks beginning clus
 no of file

//PRIVATE FUNCTION DECLARATIONS
unsigned short findFreeCluster(fat_BS_t* bs);
void clearClusterChain(unsigned short n, fat_BS_t* bs);
int traverseDirectories(char* dirname, fat_BS_t* bs);
int isDirEmpty(dirEnt* entry, fat_BS_t* bs);
int openFileSystem();
void getBootSector(void * boot_sector);
int openRootDir(fat_BS_t* bs);
unsigned int getFirstRootDirSecNum(fat_BS_t* boot_sector);
unsigned int getFirstDataSector(fat_BS_t* boot_sector);
unsigned int getFirstSectorOfCluster(unsigned int n, fat_BS_t* boot_sector);
int getCountOfClusters(fat_BS_t* bs);
unsigned int getFatValue(unsigned int n, fat_BS_t* boot_sector);
void setFatValue(unsigned int n, unsigned short value, fat_BS_t* bs);
void seekFirstSectorOfCluster(unsigned int n, int* fd, fat_BS_t* boot_sector);
unsigned int isEndOfClusterChain(unsigned int fat_value);
unsigned int getBytesPerCluster(fat_BS_t* bs);
unsigned int byteAddressToClusterNum(unsigned int byte_address, fat_BS_t* bs);
void writeDirEnt(int fd, unsigned char* name, unsigned char attr, unsigned short
 clusLo, unsigned int fileSize);
void fixStrings(char* newString, char* oldString);
void toShortName(char* newString, char* oldString);
void splitFilePath(char** buffer, const char* path);
char* getAbsolutePath(char * oldPath);
///////////////////// WRITE API ///////////////////////////
```

```
int OS_mkdir(const char *path) {
    // get boot sector
    void * boot_sector[sizeof(fat_BS_t)];
    getBootSector(boot_sector);
    fat_BS_t* bs = (fat_BS_t *) boot_sector;

    char* fileParts[2];
    splitFilePath(fileParts, path);
    char* dirpath = getAbsolutePath(fileParts[0]);
    int fd = traverseDirectories(dirpath, bs);
    if (fd == -1) {
        close(fd);
        return -1;
    }

    unsigned int currByteAddress = (unsigned int) lseek(fd, 0, SEEK_CUR);

    // CHECK IF IN ROOT OR NOT
    int readingRoot = 0;
    if (currByteAddress >= getFirstRootDirSecNum(bs) * bs->bytes_per_sector &&
        currByteAddress < getFirstDataSector(bs)* bs->bytes_per_sector) {
            readingRoot = 1;
    }

    // loop through directory entries
    int bytes_read = 0;
    unsigned int bytes_per_cluster = getBytesPerCluster(bs);
    void *currDirSpace[sizeof(dirEnt)];
    int clusterNum = byteAddressToClusterNum(currByteAddress, bs);
    // check for duplicate names
    while (1) {
        // break if read past root directory
        if ((readingRoot && bytes_read >= bs->root_entry_count) || (!readingRoot &&
         bytes_read >= bytes_per_cluster)) {
            close(fd);
            return -1;
        }

        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        char dir_name[12];
        char path_dir_name[12];
        fixStrings(dir_name, (char *) currDir->dir_name);
        toShortName(path_dir_name, fileParts[1]);
        if (strcmp(path_dir_name, dir_name) == 0) {
            close(fd);
            return -2;
        }
        if (currDir->dir_name[0] == 0x00) {
            break;
        }

        bytes_read += sizeof(dirEnt);
    }
    // no duplicate name found. Scan back to beginning of directory and look for
     first empty space
    lseek(fd, currByteAddress, SEEK_SET);
    while (1) {
        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        char dir_name[12];
        char path_dir_name[12];
        fixStrings(dir_name, (char *) currDir->dir_name);
        toShortName(path_dir_name, fileParts[1]);

        if (currDir->dir_name[0] == 0x00 || currDir->dir_name[0] == 0xE5) { // empty
         space found. Create the directory
            int freeClusterNum = findFreeCluster(bs);
            int currCluster = clusterNum;
            if (readingRoot) {
                currCluster = 0;
            }
            // write dirEnt for this new dir:
            // seek back to beginning of this dirEntry
            lseek(fd, -sizeof(dirEnt), SEEK_CUR);
            writeDirEnt(fd, (unsigned char *) path_dir_name, 0x10, (unsigned short)
             freeClusterNum, 0);
            // set fat value
            setFatValue(freeClusterNum, 0xFFFF, bs);
            // move fd to new cluster num where directory data will be
            lseek(fd, getFirstSectorOfCluster(freeClusterNum, bs) * bs-
             >bytes_per_sector, SEEK_SET);
            // create . and .. entries:
            writeDirEnt(fd, (unsigned char *) ".          ", 0x10, (unsigned short)
             freeClusterNum, 0);
            writeDirEnt(fd, (unsigned char *) "..         ", 0x10, (unsigned short)
             currCluster, 0);

            close(fd);
            return 1;
        }
    }
    close(fd);
    return -1;

}

int OS_rmdir(const char *path) {
    // get boot sector
    void * boot_sector[sizeof(fat_BS_t)];
    getBootSector(boot_sector);
    fat_BS_t* bs = (fat_BS_t *) boot_sector;

    char* fileParts[2];
    splitFilePath(fileParts, path);
    char* dirpath = getAbsolutePath(fileParts[0]);
    int fd = traverseDirectories(dirpath, bs);
    if (fd == -1) {
        close(fd);
        return -1;
    }

    unsigned int currByteAddress = (unsigned int) lseek(fd, 0, SEEK_CUR);

    // CHECK IF IN ROOT OR NOT
    int readingRoot = 0;
    if (currByteAddress >= getFirstRootDirSecNum(bs) * bs->bytes_per_sector &&
        currByteAddress < getFirstDataSector(bs)* bs->bytes_per_sector) {
            readingRoot = 1;
    }

    // loop through directory entries
    int bytes_read = 0;
    unsigned int bytes_per_cluster = getBytesPerCluster(bs);
    void *currDirSpace[sizeof(dirEnt)];
    while (1) {
        // break if read past root directory
        if ((readingRoot && bytes_read >= bs->root_entry_count) || (!readingRoot &&
         bytes_read >= bytes_per_cluster)) {
            close(fd);
            return -1;
        }

        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        char dir_name[12];
        char path_dir_name[12];
        fixStrings(dir_name, (char *) currDir->dir_name);
        toShortName(path_dir_name, fileParts[1]);
        if (strcmp(path_dir_name, dir_name) == 0) { // found the directory
            if (currDir->dir_attr != 0x10) { // make sure it's a directory
                close(fd);
                return -2;
            }
            if (!isDirEmpty(currDir, bs)) { // cannot remove if dir contains entries
                return -3;
            }
            // clear the cluster:
            clearClusterChain(currDir->dir_fstClusLO, bs);
            // overwrite the dirEntry space:
            unsigned char clearDirEnt[sizeof(dirEnt)];
            memset(clearDirEnt, 0x00, sizeof(dirEnt));
            clearDirEnt[0] = 0xE5;
            lseek(fd, -sizeof(dirEnt), SEEK_CUR);
            write(fd, clearDirEnt, sizeof(dirEnt));
            close(fd);
            return 1;
        }
        if (currDir->dir_name[0] == 0x00) { // couldn't find dir
            close(fd);
            return -1;
        }
        bytes_read += sizeof(dirEnt);
    }
    close(fd);
    return -1;
}

int OS_rm(const char *path) {
    // get boot sector
    void * boot_sector[sizeof(fat_BS_t)];
    getBootSector(boot_sector);
    fat_BS_t* bs = (fat_BS_t *) boot_sector;

    char* fileParts[2];
    splitFilePath(fileParts, path);
    char* dirpath = getAbsolutePath(fileParts[0]);
    int fd = traverseDirectories(dirpath, bs);
    if (fd == -1) {
        close(fd);
        return -1;
    }

    unsigned int currByteAddress = (unsigned int) lseek(fd, 0, SEEK_CUR);

    // CHECK IF IN ROOT OR NOT
    int readingRoot = 0;
    if (currByteAddress >= getFirstRootDirSecNum(bs) * bs->bytes_per_sector &&
        currByteAddress < getFirstDataSector(bs)* bs->bytes_per_sector) {
            readingRoot = 1;
    }

    // loop through directory entries
    int bytes_read = 0;
    unsigned int bytes_per_cluster = getBytesPerCluster(bs);
    void *currDirSpace[sizeof(dirEnt)];
    while (1) {
        // break if read past root directory
        if ((readingRoot && bytes_read >= bs->root_entry_count) || (!readingRoot &&
         bytes_read >= bytes_per_cluster)) {
            close(fd);
            return -1;
        }

        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        char dir_name[12];
        char path_dir_name[12];
        fixStrings(dir_name, (char *) currDir->dir_name);
        toShortName(path_dir_name, fileParts[1]);
        if (strcmp(path_dir_name, dir_name) == 0) { // found the directory
            if (!((currDir->dir_attr & (0x10 | 0x08)) == 0x00)) { // make sure it's
             a directory
                close(fd);
                return -2;
            }
            // clear the cluster:
            clearClusterChain(currDir->dir_fstClusLO, bs);
            // overwrite the dirEntry space:
            unsigned char clearDirEnt[sizeof(dirEnt)];
            memset(clearDirEnt, 0x00, sizeof(dirEnt));
            clearDirEnt[0] = 0xE5;
            lseek(fd, -sizeof(dirEnt), SEEK_CUR);
            write(fd, clearDirEnt, sizeof(dirEnt));
            close(fd);
            return 1;
        }
        if (currDir->dir_name[0] == 0x00) { // couldn't find file
            close(fd);
            return -1;
        }
        bytes_read += sizeof(dirEnt);
    }
    close(fd);
    return -1;
}

int OS_creat(const char *path) {
    // get boot sector
    void * boot_sector[sizeof(fat_BS_t)];
    getBootSector(boot_sector);
    fat_BS_t* bs = (fat_BS_t *) boot_sector;

    char* fileParts[2];
    splitFilePath(fileParts, path);
    char* dirpath = getAbsolutePath(fileParts[0]);
    int fd = traverseDirectories(dirpath, bs);
    if (fd == -1) {
        close(fd);
        return -1;
    }

    unsigned int currByteAddress = (unsigned int) lseek(fd, 0, SEEK_CUR);

    // CHECK IF IN ROOT OR NOT
    int readingRoot = 0;
    if (currByteAddress >= getFirstRootDirSecNum(bs) * bs->bytes_per_sector &&
        currByteAddress < getFirstDataSector(bs)* bs->bytes_per_sector) {
            readingRoot = 1;
    }

    // loop through directory entries
    int bytes_read = 0;
    unsigned int bytes_per_cluster = getBytesPerCluster(bs);
    void *currDirSpace[sizeof(dirEnt)];
    while (1) {
        // break if read past cluster
        if ((readingRoot && bytes_read >= bs->root_entry_count) || (!readingRoot &&
         bytes_read >= bytes_per_cluster)) {
            close(fd);
            return -1;
        }

        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        char dir_name[12];
        char path_dir_name[12];
        fixStrings(dir_name, (char *) currDir->dir_name);
        toShortName(path_dir_name, fileParts[1]);
        if (strcmp(path_dir_name, dir_name) == 0) {
            close(fd);
            return -2;
        }
        if (currDir->dir_name[0] == 0x00) {
            break;
        }
        bytes_read += sizeof(dirEnt);
    }
    lseek(fd, currByteAddress, SEEK_SET);
    while (1) {
        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        char dir_name[12];
        char path_dir_name[12];
        fixStrings(dir_name, (char *) currDir->dir_name);
        toShortName(path_dir_name, fileParts[1]);
        if (currDir->dir_name[0] == 0x00 || currDir->dir_name[0] == 0xE5) { // empty
         space found. Create the directory
            int freeClusterNum = findFreeCluster(bs);
            // write dirEnt for this new dir:
            // seek back to beginning of this dirEntry
            lseek(fd, -sizeof(dirEnt), SEEK_CUR);
            writeDirEnt(fd, (unsigned char *) path_dir_name, 0x20, (unsigned short)
             freeClusterNum, 0);
            // set fat value
            setFatValue(freeClusterNum, 0xFFFF, bs);

            close(fd);
            return 1;
        }
    }
    close(fd);
    return -1;
}

int OS_write(int fildes, const void *buf, int nbytes, int offset) {
    // get boot sector
    void * boot_sector[sizeof(fat_BS_t)];
    getBootSector(boot_sector);
    fat_BS_t* bs = (fat_BS_t *) boot_sector;

    // move real file descriptor to the correct place:
    unsigned int clusterNum = (unsigned int) fd_table[fildes];
    unsigned int firstSectorOfCluster = getFirstSectorOfCluster(clusterNum, bs);
    int real_fd = openFileSystem();
    lseek(real_fd, firstSectorOfCluster * bs->bytes_per_sector, SEEK_SET);

    // seek to offset
    unsigned int bytes_to_offset = offset;
    unsigned int bytes_write_from_curr_cluster = 0;
    while (1) {
        if (bytes_to_offset > getBytesPerCluster(bs)) {
            // advance cluster chain:
            unsigned int prevClusterNum = clusterNum;
            clusterNum = getFatValue(clusterNum, bs);
            if (isEndOfClusterChain(clusterNum)) {
                // need to allocate more clusters to move the offset:
                unsigned int freeClusterNum = findFreeCluster(bs);
                setFatValue((unsigned short) prevClusterNum, (unsigned short)
                 freeClusterNum, bs);
                setFatValue((unsigned short) freeClusterNum, 0xFFFF, bs);
                clusterNum = freeClusterNum;
            }
            bytes_to_offset -= getBytesPerCluster(bs);
        } else {
            seekFirstSectorOfCluster(clusterNum, &real_fd, bs);
            lseek(real_fd, bytes_to_offset, SEEK_CUR);
            bytes_write_from_curr_cluster = bytes_to_offset;
            break;
        }
    }
    unsigned int bytes_write_total = 0;

    while (bytes_write_total < nbytes) {
        int fat_value = getFatValue(clusterNum, bs);
        int remaining_bytes_in_cluster = getBytesPerCluster(bs) -
         bytes_write_from_curr_cluster;
        int remaining_bytes_total = nbytes - bytes_write_total;

        if (remaining_bytes_in_cluster < remaining_bytes_total) { // trying to write
         the rest of the cluster
            int bytes_write = write(real_fd, buf, remaining_bytes_in_cluster);
            if (bytes_write == -1) { // unsuccessful write
                close(real_fd);
                return -1;
            } else { // successful write
                bytes_write_total += bytes_write;
                buf += bytes_write;
                if (isEndOfClusterChain(fat_value)) { // trying to write more, but
                 at end of cluster chain. terminate
                    // allocate another cluster block
                    unsigned freeClusterNum = findFreeCluster(bs);
                    setFatValue((unsigned short) clusterNum, (unsigned short)
                     freeClusterNum, bs);
                    setFatValue((unsigned short) freeClusterNum, 0xFFFF, bs);

                    fat_value = freeClusterNum;
```

```c
                }
                // advance cluster chain:
                clusterNum = fat_value;
                seekFirstSectorOfCluster(clusterNum, &real_fd, bs);
                bytes_write_from_curr_cluster = 0;
                continue;
            }
        } else { // not writing past the current cluster
            int bytes_write = write(real_fd, buf, remaining_bytes_total);
            bytes_write_total += bytes_write;
            break;
        }
    }
    close(real_fd);
    return bytes_write_total;
}

//////////////////// Helper Functions //////////////////////////

/**
 * scans the FAT table and returns the int of a free cluster number
 * returns -1 if no free cluster is found
 */
unsigned short findFreeCluster(fat_BS_t* bs) {
    unsigned int resvdSecCnt = (unsigned int) bs->reserved_sector_count;
    unsigned int bytsPerSec = (unsigned int) bs->bytes_per_sector;
    int fd = openFileSystem();
    unsigned char secBuff[bytsPerSec];
    unsigned short i = 2;
    unsigned int fatOffset = i * 2;
    unsigned int fatSecNum = resvdSecCnt + (fatOffset / bytsPerSec);
    unsigned int fatEntOffset = fatOffset % bytsPerSec;
    unsigned int currFatSecNum = fatSecNum;
    unsigned short currFatValue;
    lseek(fd, fatSecNum * bytsPerSec, SEEK_SET);
    read(fd, secBuff, bytsPerSec);

    int countOfClusters = getCountOfClusters(bs);
    for (i = 2; i < countOfClusters + 2; i++) {
        fatOffset = i * 2;
        fatSecNum = resvdSecCnt + (fatOffset / bytsPerSec);
        fatEntOffset = fatOffset % bytsPerSec;
        if (fatSecNum != currFatSecNum) {
            lseek(fd, fatSecNum * bytsPerSec, SEEK_SET);
            read(fd, secBuff, bytsPerSec);
            currFatSecNum = fatSecNum;
        }
        currFatValue = (unsigned short) *((short *) &secBuff[fatEntOffset]);
        if (currFatValue == 0) {
            close(fd);
            return i;
        }
    }
    close(fd);
    return 9999;
}

/**
 * Given a cluster number, follows the FAT table through the cluster chain and
 * clears the contents in data volume and FAT table
 */
void clearClusterChain(unsigned short n, fat_BS_t* bs) {
    unsigned int bytsPerSec = (unsigned int) bs->bytes_per_sector;
    unsigned short currCluster = n;
    int fd = openFileSystem();
    unsigned char clearBuffer[bytsPerSec];
    memset(clearBuffer, 0x00, bytsPerSec);

    while (!isEndOfClusterChain(currCluster)) {
        seekFirstSectorOfCluster(currCluster, &fd, bs);
        write(fd, clearBuffer, bytsPerSec); // clear the first cluster
        unsigned short oldCluster = currCluster;
        currCluster = getFatValue(currCluster, bs);
        setFatValue(oldCluster, 0x0000, bs);
    }
    close(fd);
}

/**
 * Traverses directories down the specified absolute path. Returns -1 if failure,
  else
 * file descriptor to the directory.
 */
int traverseDirectories(char* dirname, fat_BS_t* bs) {
    int fd = openRootDir(bs);
    int readingRoot = 1;

    // go down file path. fd is set at beginning of data region for this dir/file
    void *currDirSpace[sizeof(dirEnt)];
    char* path_segment;
    char* path = strdup(dirname);
    path_segment = strtok(path, "/");
    unsigned int bytes_per_cluster = getBytesPerCluster(bs);
    unsigned int clusterNum = 2;
    while (path_segment != NULL) {
        // loop through directory entries
        unsigned int bytes_read = 0;
        while (1) {
            // break if read past root directory
            if (readingRoot && bytes_read >= bs->root_entry_count) {
                break;
            }
            // advance to next cluster in clusterchain if available
            if (!readingRoot && bytes_read >= bytes_per_cluster) {
                unsigned int fat_value = getFatValue(clusterNum, bs);
                if (isEndOfClusterChain(fat_value)) {
                    return -1; // end of cluster chain, could not find folder name
                } else {
                    clusterNum = fat_value;
                    seekFirstSectorOfCluster(clusterNum, &fd, bs);
                }
                bytes_read = 0;
            }

            read(fd, currDirSpace, sizeof(dirEnt));
            dirEnt* currDir = (dirEnt*) currDirSpace;
            char dir_name[12];
            char path_dir_name[12];
            fixStrings(dir_name, (char *) currDir->dir_name);
            toShortName(path_dir_name, path_segment);
            if (strcmp(path_dir_name, dir_name) == 0 && currDir->dir_attr == 0x10) {
                clusterNum = (unsigned int) currDir->dir_fstClusLO;
                seekFirstSectorOfCluster(clusterNum, &fd, bs);
                break;
            }
            if (currDir->dir_name[0] == 0x00) {
                return -1;
            }
            bytes_read += sizeof(dirEnt);
        }
        readingRoot = 0; // we have passed at least the root dir
        path_segment = strtok(NULL, "/");
    }
    return fd;
}

/**
 * Check that a directory is empty. Returns 1 if empty, 0 if not.
 */
int isDirEmpty(dirEnt* entry, fat_BS_t* bs) {
    int fd = openFileSystem();
    seekFirstSectorOfCluster(entry->dir_fstClusLO, &fd, bs);
    lseek(fd, sizeof(dirEnt) * 2, SEEK_CUR);
    void *currDirSpace[sizeof(dirEnt)];
    while (1) {
        read(fd, currDirSpace, sizeof(dirEnt));
        dirEnt* currDir = (dirEnt*) currDirSpace;
        if (currDir->dir_name[0] != 0x00 && currDir->dir_name[0] != 0xE5) {
            close(fd);
            return 0;
        }
        if (currDir->dir_name[0] == 0x00) {
            break;
        }
    }
    close(fd);
    return 1;
}


/**
 * Opens a file descriptor to the file system file
```

```c
 */
int openFileSystem() {
    char* filepath = getenv("FAT_FS_PATH");
    return open(filepath, O_RDWR);
}

/**
 * Gets the starting offset of the root directory
 * Takes in a pointer to the boot sector
 */
unsigned int getFirstRootDirSecNum(fat_BS_t* boot_sector) {
     unsigned int resvdSecCnt = (unsigned int) boot_sector->reserved_sector_count;
     unsigned int numFats = (unsigned int) boot_sector->table_count;
     unsigned int fatSz16 = (unsigned int) boot_sector->table_size_16;
     return resvdSecCnt + (numFats * fatSz16);
}

/**
 * Takes in a buffer and reads the boot sector into the buffer
 */
void getBootSector(void* boot_sector) {
    int fd = openFileSystem();
    read(fd, boot_sector, sizeof(fat_BS_t));
    close(fd);
}

/**
 * Open fd to point to beginning of root directory
 */
int openRootDir(fat_BS_t* bs) {
    int real_fd = openFileSystem();
    unsigned int rootDirStart = getFirstRootDirSecNum(bs);
    lseek(real_fd, rootDirStart * (bs->bytes_per_sector), SEEK_SET);
    return real_fd;
}

/**
 * returns the first data sector (start of the data region)
 */
unsigned int getFirstDataSector(fat_BS_t* boot_sector) {
    unsigned int rootEntCnt = (unsigned int) boot_sector->root_entry_count;
    unsigned int bytsPerSec = (unsigned int) boot_sector->bytes_per_sector;
    unsigned int resvdSecCnt = (unsigned int) boot_sector->reserved_sector_count;
    unsigned int numFats = (unsigned int) boot_sector->table_count;
    unsigned int fatSz16 = (unsigned int) boot_sector->table_size_16;

    unsigned int rootDirSectors = ((rootEntCnt * 32) + (bytsPerSec - 1)) /
     bytsPerSec;
    return resvdSecCnt + (numFats * fatSz16) + rootDirSectors;
}

/**
 * Takes in a cluster number N and file descriptor, and seeks the file descriptor to
 * the beginning of the cluster
 */
void seekFirstSectorOfCluster(unsigned int n, int* fd, fat_BS_t* boot_sector) {
    unsigned int firstSector;
    if (n == 0) {
        firstSector = getFirstRootDirSecNum(boot_sector);
    } else {
        firstSector = getFirstSectorOfCluster(n, boot_sector);
    }
    lseek(*fd, firstSector * boot_sector->bytes_per_sector, SEEK_SET);
}

/**
 * Takes in a cluster number N and returns the first sector of that cluster
 */
unsigned int getFirstSectorOfCluster(unsigned int n, fat_BS_t* boot_sector) {
    unsigned int secPerCluster = (unsigned int) boot_sector->sectors_per_cluster;
    unsigned int firstDataSector = getFirstDataSector(boot_sector);
    return ((n-2) * secPerCluster) + firstDataSector;
}

/**
 * Determines the total number of clusters available
 */
int getCountOfClusters(fat_BS_t* bs) {
    unsigned int totSec = (unsigned int) bs->total_sectors_32;
    unsigned int fatSz = (unsigned int) bs->table_size_16;
    unsigned int resvdSecCnt = (unsigned int) bs->reserved_sector_count;
    unsigned int secPerCluster = (unsigned int) bs->sectors_per_cluster;
    unsigned int numFats = (unsigned int) bs->table_count;
    unsigned int bytsPerSec = (unsigned int) bs->bytes_per_sector;
    unsigned int rootEntCnt = (unsigned int) bs->root_entry_count;
    unsigned int rootDirSectors = ((rootEntCnt * 32) + (bytsPerSec - 1)) /
     bytsPerSec;
    unsigned int dataSec = totSec - (resvdSecCnt + (numFats * fatSz) +
     rootDirSectors);
    return dataSec / secPerCluster;
}

/**
 * Takes in a cluster N and returns the FAT value for that cluster.
 */
unsigned int getFatValue(unsigned int n, fat_BS_t* boot_sector) {
    unsigned int resvdSecCnt = (unsigned int) boot_sector->reserved_sector_count;
    unsigned int bytsPerSec = (unsigned int) boot_sector->bytes_per_sector;
    unsigned int fatOffset = n * 2;
    unsigned int fatSecNum = resvdSecCnt + (fatOffset / bytsPerSec);
    unsigned int fatEntOffset = fatOffset % bytsPerSec;

    int fd = openFileSystem();
    lseek(fd, fatSecNum * bytsPerSec, SEEK_SET);
    unsigned char secBuff[bytsPerSec];
    read(fd, secBuff, bytsPerSec);
    close(fd);
    return (unsigned int) *((short *) &secBuff[fatEntOffset]);
```

```
}

/**
 * Takes in a cluster N and an unsigned short, setting the FAT entry to that value
 */
void setFatValue(unsigned int n, unsigned short value, fat_BS_t* bs) {
    unsigned int resvdSecCnt = (unsigned int) bs->reserved_sector_count;
    unsigned int bytsPerSec = (unsigned int) bs->bytes_per_sector;
    unsigned int fatOffset = n * 2;
    unsigned int fatSecNum = resvdSecCnt + (fatOffset / bytsPerSec);
    unsigned int fatEntOffset = fatOffset % bytsPerSec;

    int fd = openFileSystem();
    lseek(fd, fatSecNum * bytsPerSec, SEEK_SET);
    unsigned char secBuff[bytsPerSec];
    read(fd, secBuff, bytsPerSec);
    lseek(fd, fatSecNum * bytsPerSec, SEEK_SET);
    *((unsigned short *) &secBuff[fatEntOffset]) = value;
    write(fd, secBuff, bytsPerSec);
    close(fd);
}

/**
 * Takes in a FAT table cluster value and determines if it is end of cluster chain
  (0 or 1)
 */
unsigned int isEndOfClusterChain(unsigned int fat_value) {
    return fat_value >= 0xFFF8;
}

/**
 * Calculate the number of bytes per cluster
 */
unsigned int getBytesPerCluster(fat_BS_t* boot_sector) {
    unsigned int bytsPerSec = (unsigned int) boot_sector->bytes_per_sector;
    unsigned int secPerCluster = (unsigned int) boot_sector->sectors_per_cluster;
    return bytsPerSec * secPerCluster;
}

/**
 * Get cluster number from byte address
 */
unsigned int byteAddressToClusterNum(unsigned int byte_address, fat_BS_t* bs) {
    unsigned int bytes_before_data = getFirstDataSector(bs) * bs->bytes_per_sector;
    return (byte_address - bytes_before_data) / getBytesPerCluster(bs) + 2;
}

/**
 * Given dirEnt parameters and a file descriptor, writes the dirEnt to the file
  system
 */
void writeDirEnt(int fd, unsigned char* name, unsigned char attr, unsigned short
 clusLo, unsigned int fileSize) {
    unsigned char NTres = 0;
```

```c
    unsigned short zero = 0;
    write(fd, name, 11);       // name
    write(fd, &attr, 1);        // file attr
    write(fd, &NTres, 1);       // NTRes
    write(fd, &zero, 1);        // crtTimeTenth
    write(fd, &zero, 2);        // crtTime
    write(fd, &zero, 2);        // crtDate
    write(fd, &zero, 2);        // lstAccDate
    write(fd, &zero, 2);        // fstClusHI
    write(fd, &zero, 2);        // wrtTime TODO fix wrtTime and wrtDate
    write(fd, &zero, 2);        // wrtDate
    write(fd, &clusLo, 2);      // fstClusLO
    write(fd, &fileSize, 4);    // fileSize
}

/**
 * Takes a char array buffer of size 12,
 * Takes a string (possibly missing null terminator) and trims
 * trailing spaces and adds the appropriate null terminator
 */
void fixStrings(char* newString, char* oldString) {
    int i = 0;
    for (i = 0; i < 11; i++) {
        newString[i] = oldString[i];
    }
    newString[11] = '\0';
}

/**
 * Converts lowercase filename to proper shortname
 */
void toShortName(char* newString, char* oldString) {
    int i = 0;
    int len = strlen(oldString);
    while (i < 8 && !(oldString[i] == '.' && i == len - 4) && i < len){
        newString[i] = toupper(oldString[i]);
        i += 1;
    }
    while (i < 8) {
        newString[i] = ' ';
        i += 1;
    }
    if (len - 4 > 0 && oldString[len-4] == '.') {
        for (i = 0; i < 3; i++) {
            newString[8+i] = toupper(oldString[len- 3 + i]);
        }
    } else {
        for (i = 8; i < 11; i++) {
            newString[i] = ' ';
        }
    }
    if (len - 4 > 8) {
        newString[6] = '~';
        newString[7] = '1';
    }
    newString[11] = '\0';
}

/**
 * Split file path into the last entry and the path leading up to it
 * Takes a buffer of two char* pointers and the file path
 */
void splitFilePath(char** buffer, const char* path) {
    int len = strlen(path);
    int i;
    int splitIndex = -1;
    for (i = len-1; i >= 0; i--) {
        if (path[i] == '/') {
            splitIndex = i;
            break;
        }
    }
    if (splitIndex == -1) {
        buffer[0] = "";
        buffer[1] = strdup(path);
    }
    char* firstPart = malloc(sizeof(char) * (splitIndex + 1));
    char* secondPart = malloc(sizeof(char) * (len - splitIndex));
    for (i = 0; i < splitIndex; i++) {
        firstPart[i] = path[i];
    }
    firstPart[i] = '\0';
    for (i = 0; i < len - splitIndex - 1; i++) {
        secondPart[i] = path[i + splitIndex + 1];
    }
    secondPart[i] = '\0';
    buffer[0] = firstPart;
    buffer[1] = secondPart;
}

/**
 * takes in a relative or absolute path name and returns the absolute path name
 */
char* getAbsolutePath(char * oldPath) {
    if (strlen(oldPath) > 0 && oldPath[0] == '/') {
        return oldPath; // already an absolute path
    }
    char* newPath = malloc(sizeof(char) * (strlen(oldPath) + strlen(cwd) + 2) );
    strcat(newPath, cwd);
    strcat(newPath, oldPath);
    int lastCharIndex = strlen(oldPath) + strlen(cwd) - 1;
    if (newPath[lastCharIndex] != '/') {
        strcat(newPath, "/");
    }
    return newPath;
}
```