```
/**
 * 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;
```

```
                }
                // 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
```

```
 */
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;
```

```
    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;
}
```