

Write up
Comments
Code

Operating Systems Homework #3

Maurice Wong, mzw7af@virginia.edu

The assignment was successfully completed.

1. Problem:

The goal of this assignment was to create a read-only 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 read operations. 5 functions were implemented:

- *OS_cd(const char *path)* – changes the current working directory to the specified path
- *OS_open(const char *path)* – opens a file descriptor to the file specified by the path
- *OS_close(int fd)* – closes the given file descriptor
- *OS_read(int fildes, void *buffer, int nbyte, int offset)* – given a file descriptor to a previously opened file, reads nbytes into the given buffer from a given offset into the file
- *OS_readDir(const char *dirname)* – reads the directory entries at the specified directory

2. Approach:

Following the FAT16 specification document to navigate around the file system was crucial to the success of this assignment. First, structs were defined to extract sections of the file system in a manner that would be convenient to access particular fields. The following structs were defined:

- **fat_BS_t** – holds the boot sector located in the beginning reserved sectors of the file system, containing specification information about the system
- **dirEnt** – defines the structure of each directory entry, which contains information about the associated directory or file

For all API functions, it was important to first read in the boot sector into memory in order to have readily accessible information about the file system. The FAT system specification document contains various useful calculations to determine important information such as where the data sector starts, where a specific cluster starts, reading entries in the FAT itself, etc. Functions were implemented to calculate these values so that they could be invoked where necessary:

- *getFirstRootDirSecNum()*
- *getFirstDataSector()*
- *getFirstSectorOfCluster()*
- *getFatValue()*

- *getBytesPerCluster()*

For the library functions in general, some state needed to persist across function calls. In particular, the current working directory needed to be stored, as well as a local file descriptor table to keep track of open files. These were implemented as static variables in the library, with a `char*` string to keep track of the current working directory and an array of unsigned shorts to maintain the beginning cluster numbers of open files pointed to by file descriptors.

With this information setup, each library function can be implemented to perform the appropriate read operations. A recurring operation that occurs in multiple API functions is following a file path and traversing down multiple directories. For example in *OS_cd()*, the file path needs to be traversed to ensure that all parts of the file path are valid directories. In *OS_open()*, the filepath needs to be traversed to find the specified *dirEntry*. In *OS_readDir()*, the file path also needs to be traversed to locate the list of *dirEntry*'s in the specified directory. This method was implemented as *traverseDirectories()*, which splits the filepath into parts using the *strtok()* function and scanning through lists of directory entries to find ones that match parts of the file path. When an appropriate directory entry is found, it is verified to be a directory (not a file) and the cluster number is followed to find the next list of directory entries in the data volume. This eventually returns a real file descriptor pointing to the beginning of the data volume containing the last part of the file path. For instance, if `"/home/student/foo.py"` is traversed, a file descriptor pointing to the beginning of the *student* volume for `"/home/student"` is found. The appropriate library function will differ in operation after this.

The other function of interest not mentioned yet is *OS_read()*, where following the cluster chain is more important in order to read large files that span across multiple clusters. This function reads the FAT table to find the next cluster in the chain, or to determine it has reached the end of the cluster. Calculations to find the FAT entry for a specific cluster are implemented in *getFatValue()* listed above.

Another interesting discussion point is the handling of absolute vs. relative file path names. To handle this, all relative file paths are converted to absolute file paths, and the *traverseDirectories()* function traverses appropriately down the full absolute file path.

3. Problems Encountered

One major difficulty encountered was compiling the program into a shared library and linking it to another program to test the library functions. Multiple sources were consulted, as well as the TA, to figure out how to properly link the shared library into a test program.

A few other challenges were encountered when implementing the library functions itself. The hardest parts involved implementing the proper calculations to access particular parts of the file system, as specified in the FAT document. Issues were encountered in terms of the unsigned types and casting values for appropriate calculation. Many of these issues were fixed by looking at the file system through a hex editor to aid in debugging.

Another major challenge was implementing the *OS_read()* function that involved reading values into the buffer. It took some pointer arithmetic to determine how to continually read to the same buffer for files that spanned multiple clusters. Data had to be read into the buffer up until the end of the cluster, and then the file descriptor had to be shifted to the next cluster to resume reading into the buffer from where it had left off. Pointer arithmetic was performed to update the location in the buffer to read values into, as well as calculations involving the number of bytes remaining to be read and the number of bytes in a cluster.

4. Testing

To test these library functions, a test program was written that linked the shared library, and each function was extensively tested. The *OS_cd()* function was tested repeatedly throughout the test program, which changed the directory multiple times to test relative file path arguments for the other functions. The *OS_open()* function was tested in conjunction with the *OS_read()* function by opening file descriptors first and then reading from them. Reading was tested under multiple conditions. Text files were read (files under the PEOPLE directory) and printed to verify correctness. Text files were also read with the offset invoked, to ensure that the beginning characters that were offset were missing. However, most of these text files only spanned a single cluster. To test that the FAT entry reading and cluster chain following worked, the larger MEDIA files were tested on. This was tested by reading an image or movie file in the FAT16 file system, and these bytes were read into a buffer and written to a separate file. These files were then opened as the appropriate media type to ensure they had been read correctly. *OS_close()* was then tested on the open file descriptors to ensure they were closed properly.

Finally to test *OS_readDir()*, multiple directories were read, and then the resulting array of *dirEnt*'s were looped through and their *dir_name*'s were printed to ensure they had been read properly.

5. Conclusion

Navigating the FAT file system required careful adherence to the file specification documentation. Since all the information is packed so tightly and precisely, small errors in calculations could result in drastic errors, so it was crucial learning to debug with the hex editor to view values and using C properly to manipulate at the byte-level. Ultimately, a more thorough understanding of the FAT file system and its organization was achieved through this assignment, as well as following cluster chains and read operations.

```
#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);
```

```

/**
 * Homework 3: Read API for FAT 16 file system
 *
 * CS4414 Operating Systems
 * Fall 2017
 *
 * Maurice Wong - mzw7af
 *
 * main.c - Read 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
int traverseDirectories(char* dirname, 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);
unsigned int getFatValue(unsigned int n, fat_BS_t* boot_sector);
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 fixStrings(char* newString, char* oldString);
void toShortName(char* newString, char* oldString);
void splitFilePath(char** buffer, const char* path);
char* getAbsolutePath(char * oldPath);
// BEGIN IMPLEMENTATION

int OS_cd(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* newPath = getAbsolutePath(strdup(path));

```

```

    int fd = traverseDirectories(newPath, bs);
    if (fd == -1) {
        close(fd);
        return -1;
    }
    close(fd);
    cwd = newPath;
    return 1;
}

int OS_open(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;

    // directories to the directory containing the desired file
    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);
    while (1) {
        // break if read past root directory
        if (readingRoot && bytes_read >= bs->root_entry_count) {
            close(fd);
            return -1;
        }
        // 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)) {
                close(fd);
                return -1; // end of cluster chain, could not find file 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, fileParts[1]);
    if (strcmp(path_dir_name, dir_name) == 0 && currDir->dir_attr != 0x10) {
        // found file, make entry in the file descriptor table
        int i;
        for (i = 0; i < 128; i++) {
            if (fd_table[i] == 0) {
                fd_table[i] = currDir->dir_fstClusLO;
                close(fd);
                return i;
            }
        }
        // no empty spaces in the local fd table. return error
        close(fd);
        return -1;
    }
    if (currDir->dir_name[0] == 0x00) {
        close(fd);
        return -1;
    }
    bytes_read += sizeof(dirEnt);
}
close(fd);
return -1;
}

int OS_close(int fd) {
    if (fd_table[fd] != 0) {
        fd_table[fd] = 0;
        return 1;
    } else {
        return -1;
    }
}

int OS_read(int fildes, void *buf, int nbyte, 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_read_from_curr_cluster = 0;
while (1) {
    if (bytes_to_offset > getBytesPerCluster(bs)) {
        // advance cluster chain:
        clusterNum = getFatValue(clusterNum, bs);
        bytes_to_offset -= getBytesPerCluster(bs);
    } else {
        seekFirstSectorOfCluster(clusterNum, &real_fd, bs);
        lseek(real_fd, bytes_to_offset, SEEK_CUR);
        bytes_read_from_curr_cluster = bytes_to_offset;
        break;
    }
}
unsigned int bytes_read_total = 0;

while (bytes_read_total < nbyte) {
    int fat_value = getFatValue(clusterNum, bs);
    int remaining_bytes_in_cluster = getBytesPerCluster(bs) -
        bytes_read_from_curr_cluster;
    int remaining_bytes_total = nbyte - bytes_read_total;

    if (remaining_bytes_in_cluster < remaining_bytes_total) { // trying to read
        the rest of the cluster
        int bytes_read = read(real_fd, buf, remaining_bytes_in_cluster);
        if (bytes_read == -1) { // unsuccessful read
            return -1;
        } else { // successful read
            bytes_read_total += bytes_read;
            buf += bytes_read;
            if (bytes_read < remaining_bytes_in_cluster) { // reached end of
                file. terminate
                break;
            }
            if (isEndOfClusterChain(fat_value)) { // trying to read more, but at
                end of cluster chain. terminate
                break;
            }
            // advance cluster chain:
            clusterNum = fat_value;
            seekFirstSectorOfCluster(clusterNum, &real_fd, bs);
            bytes_read_from_curr_cluster = 0;
            continue;
        }
    } else { // not reading past the current cluster
        int bytes_read = read(real_fd, buf, remaining_bytes_total);
        bytes_read_total += bytes_read;
        break;
    }
}
close(real_fd);
return bytes_read_total;
}

```



```

dirEnt* OS_readDir(const char *dirname) {
    // get boot sector
    void * boot_sector[sizeof(fat_BS_t)];
    getBootSector(boot_sector);
    fat_BS_t* bs = (fat_BS_t *) boot_sector;

    // traverse directories to the directory
    char* path = getAbsolutePath(strdup(dirname));
    int fd = traverseDirectories(path, bs);
    if (fd == -1) {
        close(fd);
        return NULL;
    }
    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;
    }
    // create array to store dirEnts:
    int dir_len = 1000;
    dirEnt* directories = malloc(sizeof(dirEnt) * dir_len);
    int count = 0; // count how many dirEnts we get

    // loop through directory entries
    int bytes_read = 0;
    unsigned int bytes_per_cluster = getBytesPerCluster(bs);
    int clusterNum = byteAddressToClusterNum(currByteAddress, bs);
    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)) {
                break; // end of cluster chain, could not find file name
            } else {
                clusterNum = fat_value;
                seekFirstSectorOfCluster(clusterNum, &fd, bs);
            }
            bytes_read = 0;
        }

        read(fd, &directories[count], sizeof(dirEnt));
        if (directories[count].dir_name[0] == 0x00) {
            break;
        }
        count++;
        bytes_read += sizeof(dirEnt);
        if (count >= dir_len) {

```

```

        dir_len *= 2;
        directories = realloc(directories, sizeof(dirEnt) * dir_len);
    }
}
directories = (dirEnt*) realloc(directories, count * sizeof(dirEnt));
close(fd);
return directories;
}

/**
 * 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_fstClusL0;
                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;
}

/**
 * Opens a file descriptor to the file system file
 */
int openFileSystem() {
    char* filepath = getenv("FAT_FS_PATH");
    return open(filepath, O_RDONLY);
}

/**
 * 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;
}

/**
 * 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);
    return (unsigned int) *((short *) &secBuff[fatEntOffset]);
}

/**

```

```

    * 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;
}

/**
 * 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;
}
```