CS 4414: Operating Systems – Spring 2018
Machine Problem 1: File System Reading

Cassie Willis

Completion Level:

The code works completely when run on using the “gcc -Wall -o myfat myfat.c” command line instructions. The code may not work completely using the makefile as I am very unfamiliar with makefiles and library construction and I am not sure I completely understood what I was doing with the makefiles. With the exception of the makefile, I feel like my code performs pretty well.

Problem Description:

The goal of this assignment was to develop an API to allow for file system traversal using FATs. A file system, housed on a disk, was mined through to find directories, and five functions were implemented to traverse through the directories, create arrays of directories within the current directory, open and close files inside the file system, and read files based on a specific file name, file offset, and number of bytes to read from the file.

Approach:

The approach to developing this file system traversal API was to develop functions to handle each of the required five actions for the API. Many functions were created which were called either within other functions or in one or more of the five main functions. The functions were broken down into individual parts so that there was a separate function for every action that may need to be performed to make the API run successfully. A description of each of the functions is listed below. The functions beginning with “OS_” are API functions.

init_boot():

This function opens up the disk file and reads the boot block into memory. After the boot block is in a buffer in memory, the function finds the location of the root directory on the disk using the equation found in the FAT_spec document. This function is called in the beginning of each of the API functions to ensure that the disk is read properly.

read_root():

This function reads the root directory into the memory buffer. It does so by finding the first sector in the first cluster that the root directory is located in. It then seeks to the location of that sector and reads the root directory into the buffer. There is error checking for a buffer size that is too small or an incorrect sector location. Finally, the root directory is consisted of several other directory entires, which are read into a directory array using the create_directory_array function. The read_root function is called at the beginning of goto_dir in order to ensure proper path traversal for every possible path.

change_dir():

This function is basically the same as the read_root function, but for sub-directories within and moving past the root. The function finds the first sector of the cluster holding the directory and uses this sector to read the directory into the memory buffer. Like read_root, this function creates a an array of directory entries and includes checks for the correct location and buffer size. Change_dir is called in goto_dir in order to traverse into directories in a path.

create_directory_array():
This function creates an array of all of the directories and files that are in the current working directory. The function uses memcpy to add chunks of data from the buffer, the chuncks being the size of a directory entry, into a directory entry structure. The structures are then added to an array of structures. This function is called in read_root and change_dir to create arrays for the current directories as soon as those directories are created.

chew_path():
This function uses the current working directory's directory array to search through a path and attempt to find the next directory listed in the path. The function takes in a pathNumber integer parameter which tells the function which item in a names[] array is being searched for. The names array is instantiated in goto_dir and holds each item in the path list (i.e. 'root', 'people', 'ccw5ft', 'coffee.txt'). The chew_path function compares the name in the names[pathNumber] entry to each of the current directory array's entries (specifically the dir_name variable) to find a match. If a match is found, a newPath variable is set which points the the directory array entry in which the next directory or file in the path is held. This function is called in goto_dir to assist with traversing the path.

goto_dir():
This is the largest function which performs the most operations in the program. This function uses read_dir, change_dir, and chew_path to traverse the directories from the root to the end of the path. This function can handle up to five pathnames, which was chosen due to errors in recursive implementation that were not able to be solved, and the assumption that the majority of common path names will not be more entrenched than five directories deep. If more time was given, this is the one part of the program the author would change first, as recursive functionality would have been a much more robust option. The goto_dir function first creates an array of names[] using strsep(), as described in chew_path above. The names[] array holds all of the locations in the pathname. The function then reads the root directory using read_root to initialize the path at the root. The first element of names[] is ignored because the pathname always begins with '/' and therefore names[0] is always empty. The function then finds the first directory entry past the root, and uses change_dir to move to that directory. If the next pathname in names[] is empty, the CWD is set to the current directory's name, as this signifies the end of the path. Otherwise, chew_path is used to go to the next directory in the path. The CWD is set if the current pathname is a file (determined using the dir_attr in the dirEnt structure) or is the end of the path. The process of moving through change_dir and chew_path is repeated until the CWD is set or until the path length is equal to five, which is the maximum path length the function can handle. The function also keeps track of a 'failure' variable, which is set to one if the path is traversed successfully, and negative one if the path is not traversed successfully. The goto_dir function is called in OS_cd and in OS_readDir, as this function is the majority or the functionality to make these two API functions work.

OS_cd():
This function sets the CWD to the end of the path that is passed in as the parameter. This is one of the API functions, and it relies heavily on the goto_dir function to complete cd functionality. The function returns a success variable, which is equal to one if the CWD is set successfully and negative one otherwise.

OS_open():
This is also one of the API functions. Its purpose is to open a file given a pathname to the file. This

function is very simple and opens the file using open() in read/write mode. The function returns the file descriptor if the file was opened successfully, and otherwise returns negative one.

OS_close():
This API function closes a file given the file descriptor for the file. It uses the close() command to close the file, and returns negative one upon unsuccessful close, and one upon successful close.

OS_read():
This function is also one of the API functions. The function is given a file descriptor, an offset (from the beginning of the file) where the API user wishes to start reading the file, the number of bytes the API user wishes to read, and a buffer of at least that many bytes to hold the bytes that were read from the file. OS_read uses lseek() to go to the offset location of the file. It then reads the parameterized number of bytes from the file into the buffer. If the file was read successfully, the function returns the number of bytes read. Otherwise, the file returns negative one. This return number is set as a built in return of the read() function which is used to read from the file. The only aspect of this function that was not implemented due to time constraints was the functionality to move to a different cluster to continue reading the file if the file takes up more than one cluster space.

OS_readDir():
This function returns the directory array for the current working directory. Given a path to the directory, the function calls on goto_dir to traverse to the directory at the end of the path. When at the end, goto_dir automatically creates an array of directory entry structures that are in the current working directory. If there are no directory entries in the current path, NULL is returned. Otherwise, the directory array is returned.

Results:
The program when using the gcc compiler without the makefile, as well as when using the sampledisk32.raw file, worked nearly as expected. Every function performs as it is supposed to the the FAT file system can be traversed successfully as long as the pathnames are no more than five directories or file names in length. Files can also be read successfully as long as they are held in only one cluster. These two caveats to complete working code would be fixed by changing the goto_dir function to be recursive (which was attempted unsuccessfully), and to use information in the dirEnt structures to determine if a file continued on to another cluster and using that information to move to the next cluster and continue reading the file from there. Equations for that functionality are in the FAT_spec documentation, they were just not implemented because the author preferred to have the rest of the code functioning properly with time to turn in the machine problem without too much of a late penalty. It should be noted that all testing for this program was done through print statements and a main() function in myfat.c, which was removed for submission as the API cannot function properly with a main() function.

Analysis:
Several problems were found when constructing the API to read through the file system. The primary problems were as listed above in the results function, with the two areas of functionality in the code that are currently missing. More research will need to be done in particular with the recursive functionality to determine why the implementation was not correct, forcing the author to move to a more static and lackluster traversal approach. The author also had trouble with the makefile, as makefiles have only been

seen previously in CS 2150, which was taken by the author two years ago. Unfamiliarity with makefiles definitely caused setbacks with getting the program fully operational in time, and it is still not completely known if the makefile runs properly. This is due completely to a lack of understanding on the part of the author, who will attempt more research in the future on the workings of makefiles and how to construct them. This will particularly be useful with library construction in makefiles, which the author had not done before this assignment and had particular trouble with. The final problem that the author ran into with this program was determining the location of the boot sector and root directory. The author knew that the boot sector was at the beginning of the disk and was able to use the boot sector to find the location of the root directory in a hex editor, however putting these findings into workable code proved challenging for the author and took the most time out of all of the problem implementation. Once understanding how to move around in the disk, a lot of the remaining functionality came easily to the author.

A good deal was learned from this machine problem. The author was able to more fully understand the FAT file system and how it is laid out in memory. Though this was discussed in lecture, having the hands on experience of actually going through the file system provided more in-depth understanding of the topic. The author also learned more about the C programming language. In particular, more experience was gained in opening and reading files and moving through files using lseek(). Some information was learned about makefiles, but the author realizes they have much more to learn on the subject before comfortable creating makefiles from scratch without aid. Finally, the author learned how to manage time better. It was not realized by the author that operating systems tasks take so long to develop and that so much work needs to be completed before coding can even begin. The author was not prepared to spend 22 hours on this assignment and therefore did not give themselves enough time to comfortably work on the assignment as they should have. This is a big lesson learned to start these assignments even earlier and manage time more effectively in the future.

Conclusion:

The FAT file system is a very robust setup that proved fairly easy to navigate once it was fully understood. It is interesting to see how this file system is actually in use in operating systems running on current machines, as it is not very often in a classroom setting that real-world applications are put into direct use. Usually real-world applications are discussed but an easier version of such applications is implemented, however this was not the case in the machine problem implementation here. It was useful to take a hands on approach to FAT in order to more thoroughly understand it, as the author feels as though much was learned through this process. Knowing more about FAT and operating systems gives the author a greater appreciation for operating systems. It is interesting to know that the inner workings of operating systems are much more approachable than previously considered. As if opening up what was previously a black box and realizing that though developing an operating system is not easy, taking it in individual chunks creates a manageable system to develop.

Pledge:

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

Cassie Willis

Code Dump:

myfat.h:

```
#ifndef __MYFAT_H__
#define __MYFAT_H__

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

typedef struct __attribute__ ((packed)) {

    uint8_t bs_jmpBoot[3];          // jmp instr to boot code
    uint8_t bs_oemName[8];          // indicates what system formatted this field, default=MSWIN4.1
    uint16_t bpb_bytesPerSec;       // Count of bytes per sector
    uint8_t bpb_secPerClus;         // no.of sectors per allocation unit
    uint16_t bpb_rsvdSecCnt;        // no.of reserved sectors in the resercved region of the volume
                                       starting at 1st sector
    uint8_t bpb_numFATs;            // The count of FAT datastructures on the volume
    uint16_t bpb_rootEntCnt;        // Count of 32-byte entries in root dir, for FAT32 set to 0
    uint16_t bpb_totSec16;          // total sectors on the volume
    uint8_t bpb_media;              // value of fixed media
    uint16_t bpb_FATSz16;           // count of sectors occupied by one FAT
    uint16_t bpb_secPerTrk;         // sectors per track for interrupt 0x13, only for special devices
    uint16_t bpb_numHeads;          // no.of heads for intettupr 0x13
    uint32_t bpb_hiddSec;           // count of hidden sectors
    uint32_t bpb_totSec32;          // count of sectors on volume
    uint32_t bpb_FATSz32;           // define for FAT32 only
    uint16_t bpb_extFlags;          // Reserved for FAT32
    uint16_t bpb_FSVer;             // Major/Minor version num
    uint32_t bpb_RootClus;          // Clus num of 1st clus of root dir
    uint16_t bpb_FSInfo;            // sec num of FSINFO struct
    uint16_t bpb_bkBootSec;         // copy of boot record
    uint8_t bpb_reserved[12];       // reserved for future expansion
    uint8_t bs_drvNum;              // drive num
    uint8_t bs_reserved1;           // for ue by NTS
    uint8_t bs_bootSig;             // extended boot signature
    uint32_t bs_volID;              // volume serial number
    uint8_t bs_volLab[11];          // volume label
    uint8_t bs_fileSysTye[8];       // FAT12, FAT16 etc
} bpbFat32 ;

typedef struct __attribute__ ((packed)) {
    uint8_t dir_name[11];           // short name
    uint8_t dir_attr;               // File attribute
```

```
        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
} dirEnt;


extern int OS_cd(const char *path); //Go to the directory at the end of the path, on success returns 1 and
                                        changes the CWD to path. On failure returns -1. The CWD is
                                        initially "/"
extern int OS_open(const char *path); //opens a file read/write, path refers to the absolute or relative path
                                        of the file. On success returns the file descriptor to be used. On
                                        failure returns -1
extern int OS_close(int fd); //closes the file. On success returns 1, on failure returns -1
extern int OS_read(int fildes, void *buf, int nbyte, int offset); //fildes refers to a previously opened file. buf
                                refers to a buffer of at least nbyte; nbyte is the number of bytes to read.
                                 //offset is the offset in the file to begin reading. returns the number of bytes
                                read if positive. Otherwise returns -1
extern dirEnt * OS_readDir(const char *dirname); //dirname is the path to the directory, readDir returns
                                a newly allocated array of DIRENTRYs. It is the users responsibility to free
                                 //the array when done. If the directory could not be found a null is
                                returned.



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

void read_root(); //This function reads the root directory
int init_boot(); //This function initializes the boot block and reading of the disk, as well as finding the root
                directory
void change_dir(int clusNum); //move to a directory given the cluster number and read into the buffer
void create_directory_array(); //This function creates an array of directory entries from the current
                                working directory. Entries can be other directories or files
void chew_path(int pathNumber); //Iterates through the path to find the next directory or file
void readFile(char input[]);
int goto_dir(const char *pathname); //read the root dir, find name of next dir, read that dir,... find name
                                        of file. If file or dir not found, erroneous pathname
```

```
char *cwdPath;          // current working dir name
int fdCount;

#endif
```

myfat.c:

```
//myfat.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include "myfat.h"

#define bufSize 10000 //size of the buffer
#define EoC 0x0FFFFFF8 //end of the cluster signal, no more clusters in directory

int RootLocation16 = 0x3E000; //Root Directory Name = LEXAR
int RootLocation32 = 0xF8000;

int bytesPerCluster; //total number of clusters on the disk
bpbFat32 bootBlock;
dirEnt directory;
int fd; //file descriptor
unsigned char *buffer; //buffer to read clusters into memory
int firstDataSector; //first sector of the data section of the disk
dirEnt directoryArray[32];
char *names[10];
int newPath;
int isDirectory = 1; //initialized to 1 because root is a directory, not a file

char nameOfFile[] = "sampledisk32.raw"; //name of the disk file to be run

//This function initializes the boot block and reading of the disk, as well as finding the root directory
int init_boot() {
        buffer=(unsigned char *)malloc(bufSize); //init buffer to hold disk info
        OS_open((const char*) nameOfFile);
        if (!fd) {
                printf("Unable to open file"); //file here is sampledisk32.raw
                return -1;
        }
        else { //check if the file can be read properly
                int errorCheck;
```

```
		if ((errorCheck = read(fd, buffer, bufSize)) != bufSize) {
			printf("Error reading boot %d\n", errorCheck);
			exit(0);
		}
		memcpy(&bootBlock, buffer, sizeof(bpbFat32)); //read the boot block into the buffer
		bytesPerCluster = bootBlock.bpb_bytesPerSec * bootBlock.bpb_secPerClus; //finding
										number of bytes per cluster
		firstDataSector = bootBlock.bpb_rsvdSecCnt + (bootBlock.bpb_numFATs *
			bootBlock.bpb_FATSz32); //finding location of the first sector of data in the disk
	}
	return 1;
}

/*----------------------------------------------------------------------------------------------------------*/

//This function reads the root directory
void read_root() {
	memset(buffer, 0, bufSize);
	int rootDirfirstSector = ((bootBlock.bpb_RootClus - 2) * bootBlock.bpb_secPerClus) +
		firstDataSector; //First sector in the first cluster for the root directory
	int location = lseek(fd, rootDirfirstSector*bootBlock.bpb_bytesPerSec, SEEK_SET); //go to the
											root directory
	if(bootBlock.bpb_secPerClus*bootBlock.bpb_bytesPerSec > bufSize) {
		printf("Buffer size too small\n");
		exit(0);
	}
	int info = read(fd, buffer, bytesPerCluster); //read root into buffer
	if(info != bootBlock.bpb_secPerClus*bootBlock.bpb_bytesPerSec) {
		printf("Cannot read sector\n");
		exit(0);
	}
	create_directory_array(); //set up the array of files and directories inside the root directory (root
												FAT)
	return;
}

/*----------------------------------------------------------------------------------------------------------------*/

//This function creates an array of directory entries from the current working directory. Entries can be
	other directories or files
void create_directory_array() {
	int i;
	for (i = 0; i < sizeof(dirEnt); i++) {
		memcpy(&directory, buffer + i*sizeof(dirEnt), sizeof(dirEnt));
		if(directory.dir_name != NULL) { //if there is actually something here, add it to the array
			directoryArray[i] = directory;
		}
	}
	return;
}

/*-----------------------------------------------------------------------------------------------------------*/

//move to a directory given the cluster number and read into the buffer
//calls create_directory array()
void change_dir(int clusNum) {
	int FirstSectorofCluster = ((clusNum - 2) * bootBlock.bpb_secPerClus) + firstDataSector; //finds
											the first sector in the current cluster
	int location = lseek(fd, FirstSectorofCluster*bootBlock.bpb_bytesPerSec, SEEK_SET); //goes to
											the cluster the directory is located in
	if(bootBlock.bpb_secPerClus*bootBlock.bpb_bytesPerSec > bufSize) { //check size of buffer to
											make sure it can hold directory
		printf("Buffer size too small\n");
		exit(0);
	}
	int info = read(fd, buffer, bytesPerCluster); //read cluster into memory buffer
	if(info != bootBlock.bpb_secPerClus*bootBlock.bpb_bytesPerSec) { //if buffer doesn't equal size
							of cluster then not everything in sector was read properly
		printf("Cannot read sector\n");
		exit(0);
	}
	create_directory_array();
	return;
}

/*------------------------------------------------------------------------------------------------------------*/

//read the root dir, find name of next dir, read that dir,... find name of file. If file or dir not found,
	erroneous pathname
//calls read_root(), chage_dir(), and chew_path()
int goto_dir(const char *pathname) {
   int i = 0;
   int failure = -1; //failure is 1 if success, -1 if fail
   int j;
   char *temp = strdup(pathname);
	char *token;
	while ((token = strsep(&temp, "/")) != NULL) { //create an array of path names
	names[i] = token;
	i++;
	}
	read_root(); //start at the root directory and try to find the file name to start traversing
		//ignore the first element of names because empty, ignore second element of names
		because root name
```

```
for(j = 0; j < 32; j++) {
        if(strncmp(names[2], (char *) directoryArray[j].dir_name, 6) == 0) { //compare the name
                                                              of the directory in the CWD to the names of each
                                                              directory in the array of directories
                failure = 1;
                newPath = j;
                break;
        }
        else {
                failure = -1;
        }
}
if (failure == -1) { //if the next path cannot be found in the current directory
        printf("Path name was invalid / does not exist\n");
}
else { //path was found, move to it
        change_dir(directoryArray[j].dir_fstClusLO);
        //then need to repeat with the rest of the path based on length of names[]
        if (strlen(names[3]) == 0) { //end of path, change cwdPath
                cwdPath = directoryArray[newPath].dir_name;
                failure = 1;
        }
        else {
                chew_path(3);
                if(isDirectory == 0) { //reached  a file, no more directories, change cwd path
                        cwdPath = directoryArray[newPath].dir_name;
                        failure = 1;
                }
                else if (strlen(names[4]) == 0) { //end of path, change cwdPath
                        cwdPath = directoryArray[newPath].dir_name;
                        failure = 1;
                }
                else {
                        change_dir(directoryArray[newPath].dir_fstClusLO);
                        if(directoryArray[newPath].dir_attr == 0x20) isDirectory = 0;
                        if (isDirectory == 0) { //reached  a file, no more directories, change cwd
                                                          path
                                cwdPath = directoryArray[newPath].dir_name;
                                failure = 1;
                        }
                        else if (strlen(names[5]) == 0) { //end of path, change cwdPath
                                cwdPath = directoryArray[newPath].dir_name;
                                failure = 1;
                        }
                        else {
                                if (strlen(names[5]) == 0) { //end of path
                                        printf("End of Path, no file found in current directory\n");
```

```
                                                            }
                                                            else {
                                                                      chew_path(4); //only goes up to a path length of 5 because
                                                                                    recursive functions were not working
                                                                                    properly, example is max 5
                                                                      if(isDirectory == 0) { //reached  a file, no more directories,
                                                                                              change cwd path
                                                                                cwdPath = directoryArray[newPath].dir_name;
                                                                                failure = 1;
                                                                      }
                                                                      else if (strlen(names[6]) == 0) { //end of path, change
                                                                                                        cwdPath
                                                                                cwdPath = directoryArray[newPath].dir_name;
                                                                                failure = 1;
                                                                      }
(**Code moved to left-justify to make it easier to read**)
else {
          change_dir(directoryArray[newPath].dir_fstClusLO);
           if(directoryArray[newPath].dir_attr == 0x20) isDirectory = 0; //isDireectory = 0 means the
                                                                         current directory  placement is on a
                                                                         file, not a directory
          if (isDirectory == 0) { //reached  a file, no more directories, change cwd path
                    cwdPath = directoryArray[newPath].dir_name;
                    failure = 1;
          }
          else if (strlen(names[7]) == 0) { //end of path, change cwdPath
                    cwdPath = directoryArray[newPath].dir_name;
                    failure = 1;
          }
}

(**Moved back right to show all closing brackets**)
                                                  }
                                        }
                              }
                    }
          }
          return failure;
}

/*------------------------------------------------------------------------------------------------------------------*/

//Iterates through the path to find the next directory or file
void chew_path(int pathNumber) {
          int k;
          int failure = 0;
          //check if it is in the directory, then check if it's a directory or file
```

```
        for(k = 0; k < 31; k++) {
                if(strncmp(names[pathNumber], (char *) directoryArray[k].dir_name, 5) == 0) {
                        failure = 0;
                        if(directoryArray[k].dir_attr == 0x20) isDirectory = 0;
                        newPath = k;
                        break;
                }
                else {
                        failure = 1;
                }
        }
        if (failure == 1) { //if the next path cannot be found in the current directory
                printf("Path name was invalid / does not exist\n");
        }
        return;
}

/*-----------------------------------------------------------------------------------------------------------------*/

//set the CWD path to the end of the path
extern int OS_cd(const char *path){
        init_boot();
        int success = goto_dir(path);
        return success;
}

/*-----------------------------------------------------------------------------------------------------------------*/

//opens a file read/write
extern int OS_open(const char *path) {
        init_boot();
        fd =  open(path, O_RDWR);
        return fd;
}

/*-----------------------------------------------------------------------------------------------------------------*/

 //closes the file
extern int OS_close(int fd) {
        if (close(fd) < 0) return -1;
        else return 1;
}

/*-----------------------------------------------------------------------------------------------------------------*/

//read the given open file with parameter considerations
extern int OS_read(int fildes, void *buf, int nbyte, int offset) {
	init_boot();
	lseek(fildes, offset, 0); //go to the offset in the file where you want to start reading
	int successRead = read(fildes, &buf, nbyte); //copies nbytes of the file from the offset into the
								buffer
	return successRead;
}

/*-----------------------------------------------------------------------------------------------------------------*/

//return the list of directory entries in the current directory
//create_directory_array(), change_dir()
extern dirEnt * OS_readDir(const char *dirname) {
	init_boot();
	goto_dir(dirname);
	if (sizeof(directoryArray) == 0) return NULL;
	else return directoryArray;
}

/*------------------------------------------------------------------------------------------------------------*/

extern int OS_rmdir(const char *path) {
	return 0;
}
extern int OS_mkdir(const char *path) {
	return 0;
}
extern int OS_rm(const char *path) {
	return 0;
}
extern int OS_creat(const char *path) {
	return 0;
}
extern int OS_write(int fildes, const void *buf, int nbyte, int offset) {
	return 0;
}
```