The assignment was completed! Note that to enter a directory or open/read a file, the EXACT name must be used as given by lsdir (including case).

Problem Description

Create a library that implements the Microsoft FAT file system read API containing the following commands: OS_cd, OS_open, OS_close, OS_read, and OS_readDir as specified in the problem description handout on collab. Source code must be provided with a makefile that compiles the source code into a shared library (.so). Source code must be written in either C or C++. It is only required to implement either FAT16 or FAT32.

Approach

FAT32 was implemented in C. Before being linked to the testing shell given by the instructor, basic functionality to recurse through all directories, read the FAT table, and follow a cluster chain were implemented. The FATSPEC was carefully studied and referred to throughout the whole development process. The following methods were created before implementation of the required functions used by the testing shell:

- **Init()** - This function reads in the FAT disk image specified by the environment variable "FAT_FS_PATH". The bpb was read into to the given struct bpbFat32 and a number of global variables were set to be used by other functions: bytesPerClus, rootDirSectors, firstDataSector, numDataSec, and numClusters. The calculations for those variables were taken from the FATSPEC. Space was allocated for openDir, an array of dirEnts with a max size of 128 entries. This array is used to contain all the open files as specified in OS_open. The current directory's cluster number is set to cwdCluster which keeps track of the directory we are in. Lastly, the fat table is read in through the function "readFatTable" as described below
- r**ecurseThroughDir()** - This function recurses through all directories in the DATA section of the FAT disk image. It checks the directories attribute and the first character in the filename to better understand if it needs to skip the directory or stop iterating through the cluster.
- **firstClusterSector()** - This function returns the sector and the inputted cluster relative to the entire disk. This equation was taken from the FATSPEC.
- **printDir()** - This function prints important metadata of the directory including the name, attribute, NTRes, and the directory's cluster number.
- **tableVale()** - This function returns the FAT table value of the specified cluster while also applying the mask.
- **clusterChainSize()** - This function returns the length of the cluster chain until to EOC is reached.
- **clusterChain()** - This function returns an unsigned int array containing the cluster chain.
- **readFatTable()** - This function reads the FAT table into memory. Each element fatTable[i] contains the next cluster in the cluster chain of the i'th cluster.

Afterwards, to following functions were created to implement the required functions for the given testing shell:

- **OS_cd()** - follows the directories set out in path and changes the cwdCluster to the cluster of the last directory chain specified in the path. This function itself only handles a relative path to a single directory in its current working directory by looking if the path name matches the filename of a directory in its cwd. Should an absolute path or multiple directories (cd 1/2/3) is encountered, the function relegates the path to the helper function "cdAbsolute()".
- **cdAbsolute()** - This function breaks the path down by calling OS_cd() for each directory in between the delimiter "/".
- **dirName()** - This function returns the directory or filename of the specified dirEnt as a string.

- **OS_readDIr()** - This function returns an array of each file or directory at the specified path. This function changes the directory to the specified path (if applies) and then iterates though the cluster, picking up all the files/directories. Finally, it follows the cluster chain in the event there are more files in the directory than can be contained in a cluster.
- **OS_open()** - This function loads the file specified in the path to the global variable "openDIr" as described above. It follows the path, changing the working directory until the file is reached and checks if the file is a valid file. The current working directory is set back to the original working directory before this function was called and the position of the file loaded into openDir is returned as the file descriptor. This function does a linear search throughout openDir to find the first free spot.
- **OS_close()** - This function replaces the entry in openDir at position fd with an empty dirEnt that can be reused.
- **OS_read()** - This function loads the file in openDir at position fildes. It computes the first cluster to read from the cluster chain given the offset and reads the cluster until the end of the cluster or until the number of bytes to read (that is specified as an argument) is read. If the end of the cluster is reached and there are more bytes to be read, it loads the next cluster in the cluster chain and continues reading until again, the end of the cluster is reached or until the number of bytes specified are read.

Results

The compiled source library is able to link with the given testing shell with all of the read API commands operational. Testing was done via print statements (as seen commented out throughout the source code) and through trial and error with a main() and through the shell test script. All files were read and double checked against the mounted file manually to ensure correctness.

Analysis

Let n be the number of files in the current working directory and assuming loading data into memory (seek and read) takes constant time. **OS_readDir(".")** then runs in **O(n)** time as it stops after iterating through the cluster if the first character in the directory name is 0x00. Iterating through each directory entry in the cluster dominates the runtime of this function.
Let m be the number of directories in a path. **OS_cd()** thus runs in **O(m * n)** as it calls OS_readDir(".") for each directory directory in the specified path. Iterating though each directory and reading the directories in each directory dominates the run time in this function.

OS_open also runs in **O(m * n)** as it follows the same operations as OS_cd() and then iterates through openDir to find the first empty spot. Iterating through the entire openDir array takes constant time O(128).

OS_close runs in **O(1)** as the function simply checks the openDir array at the specified position and either replaces that entry with a blank entry or returns -1.

Let p be the number of clusters in the cluster chain to follow. At worst case, **OS_read** runs in **O(m * n * p)** if the entire file is read, making us follow the entire cluster chain.

Conclusion

Performance of the OS functions can be increased due to some redundancy iterating through filenames and directories. Unit tests could be made to replace the extensive manual testing that took place for further QA. Overall, I believe the problem description was successfully resolved.

Pledge

I have neither given nor received aid on this assignment
Benjamin Trans (bvt2nc)

```
#include "myfat.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

int start = 0;
int bytesPerClus;
int rootDirSectors;
int firstDataSector;
int FATSz;
int numDataSec;
int totSec;
int numClusters; // number of DATA clusters
bpbFat32 bpb;
//int count = 0;
int fatType = 0;
uint32_t *fatTable;
//uint16_t fatTable[1000];
int EOC;
int cwdCluster; //current working directory cluster used throughout code to represent "ptr
locaton"
FILE * fd;

void init();
int firstClusterSector(int n);
void printDir(dirEnt dir);
void recurseThroughDir(FILE *fd, int offset);
unsigned int tableValue(int cluster);
void readFatTable(FILE *fd);
int clusterSize(int cluster, int size);
unsigned int * clusterChain(int cluster);
char * dirName(dirEnt dir, int file);
int cdAbsolute(const char * path);
dirEnt * openDir;

int main() {

        //Sandbox for testing functions

   OS_cd("PEOPLE");
   OS_readDir(".");
   OS_cd("ABK2Y");
   OS_readDir(".");
   OS_cd("/PEOPLE/ABK2Y");
   OS_cd("~");
   OS_readDir(".");
   OS_readDir("/PEOPLE/ABK2Y");
   OS_open("CONGRATSTXT");
   OS_cd("MEDIA");
   OS_open("EPAA22~1JPG");
```

```
    /*OS_cd("../PEOPLE");
    dirEnt * currentDirs = OS_readDir(".");
    int i;
    for(i = 0; i < 128; i++)
    {
        if(currentDirs[i].dir_name[0] == 0x00)
            break;

        printDir(currentDirs[i]);
    }*/
    OS_open("~/MEDIA/EPAA22~1JPG");
    /*init();
    //readFatTable(fd);
    recurseThroughDir(fd, firstClusterSector(2) * bpb.bpb_bytesPerSec);*/

    return 0;
}

//Load the disk and set all the globals variables
//Loads bpb and FAT table into memory
//Allocates memory to global arrays used by core functions
void init()
{
    //fd = fopen("sampledisk32.raw", "rb");
        //If you don't want to do the below, uncomment above statement
        //Assumes sampledisk32.raw is in the same directory as code
    fd = fopen(getenv("FAT_FS_PATH"), "rb"); //get the env var FAT_FS_PATH
        //Assuming it has been set though...

    fseek(fd, 0, SEEK_SET);
        fread(&bpb, sizeof(bpbFat32), 1, fd);

    if(bpb.bpb_FATSz16 != 0)
    {
        FATSz = bpb.bpb_FATSz16;
        totSec = bpb.bpb_totSec16;
        fatType = 16;
    }
        else
        {
            FATSz = bpb.bpb_FATSz32;
            totSec = bpb.bpb_totSec32;
            fatType = 32;
        }

    bytesPerClus = bpb.bpb_bytesPerSec * bpb.bpb_secPerClus;
    rootDirSectors = ((bpb.bpb_rootEntCnt * 32) + (bpb.bpb_bytesPerSec - 1)) /
bpb.bpb_bytesPerSec;
        firstDataSector = bpb.bpb_rsvdSecCnt + (bpb.bpb_numFATs * FATSz) + rootDirSectors;
        numDataSec = totSec - (bpb.bpb_rsvdSecCnt + (bpb.bpb_numFATs * FATSz) +
rootDirSectors);
        numClusters = numDataSec / bpb.bpb_secPerClus;
        openDir = (dirEnt*)malloc(sizeof(dirEnt) * 128);
```

```
        //if init has been read, start = 1
        //otherwise start = 0
        start = 1;
        cwdCluster = 2; //default to root directory
    readFatTable(fd);

        /*printf("end init \n");
        dirEnt *dir = (dirEnt*)malloc(numClusters * bytesPerClus);
        printf("reserved sector count: %d \n", bpb.bpb_rsvdSecCnt);
        printf("fatsz: %d \n", FATSz);
        printf("bytes per sec: %d \n", bpb.bpb_bytesPerSec);
        printf("sec per cluster: %d \n", bpb.bpb_secPerClus);
        printf("dirEnt size: %d \n", sizeof(dirEnt));

        int offset = firstClusterSector(2) * bpb.bpb_bytesPerSec;
        readFatTable(fd);
        printf("FAT[0]: 0x%x \n", tableValue(0));
        printf("FAT[1]: 0x%x \n", tableValue(1));
        EOC = 0x0FFFFFFF;
        printf("EOC: 0x%x \n", EOC);
        recurseThroughDir(fd, offset);*/

        /*char *c = "congrats.txt";
        printf("%s \n", c);
        char* p = c;
        while (*p)
        {
           *p++;
           printf("%d\n", strtoul(p, &p, 10));
        }
        //printf("%d \n", OS_cd(c));
        printf("Count: %d \n", count);*/
}

//Test function that recurses through each directory and their subdirectory
//Prints each directory and file info including cluster information
void recurseThroughDir(FILE * fd, int offset)
{
        int inc;
        dirEnt dir;
        for(inc = 0; inc < bytesPerClus - 64; inc += 32)
        {
                fseek(fd, offset + inc, SEEK_SET);
                fread(&dir, sizeof(dirEnt), 1, fd);
                if(dir.dir_name[0] == 0x00)
                        break;
                if(dir.dir_name[0] == 0xE5)
                        continue;
                if(dir.dir_attr == 8 || dir.dir_attr == 15)
                        continue;
                printf("==================================\n");
                printDir(dir);
                if(dir.dir_attr == 16)
                {
```

```
                        //printf("==================================\n");
                        //printf("recurring...\n");
                        //printf("FAT table value: 0x%x\n", tableValue(dir.dir_fstClusLO));
                        recurseThroughDir(fd, (firstClusterSector(dir.dir_fstClusLO) *
bpb.bpb_bytesPerSec) + 64);
                }
                else
                {
                        //unsigned int * chain = clusterChain(dir.dir_fstClusLO);
                }
                //printf("==================================\n");
        }
}

//As described in the FAT spec, returns the first sector of the specified cluster
//relative to the entire disk
int firstClusterSector(int n)
{
        return ((n - 2) * bpb.bpb_secPerClus) + firstDataSector;
}

//Print directory and file meta data
void printDir(dirEnt dir)
{
        int i, attr;
        attr = dir.dir_attr;
        char c;
        //Parse the filename
        for(i = 0; i < 11; i++)
        {
                if(attr != 16 && attr != 8 && i == 8)
                        printf(".");
                c = dir.dir_name[i];
                printf("%c", c);
        }
        printf("\n");

        printf("Attributes: %d \n", attr);
        printf("NTRes: %d \n", dir.dir_NTRes);
        unsigned int * chain = clusterChain(dir.dir_fstClusLO);
        printf("First Cluster High: 0x%x \n", dir.dir_fstClusHI);
        printf("First Cluster Low: 0x%x \n", dir.dir_fstClusLO);
        printf("First Cluster Table Value: 0x%x \n", tableValue(dir.dir_fstClusLO));
        if(dir.dir_fstClusLO < EOC)
                printf("Next Cluster Table Value: 0x%x \n",
tableValue(tableValue(dir.dir_fstClusLO)));
}

//returns the value at fatTable[cluster]
unsigned int tableValue(int cluster)
{
        /*unsigned int fatOffset;
        if(fatType == 16)
                fatOffset = cluster * 2;
```

```
	else
		fatOffset = cluster * 4;

	unsigned int fatSector = (fatOffset / bpb.bpb_bytesPerSec);
	unsigned int fatEntOffset = fatOffset % bpb.bpb_bytesPerSec;*/

	//Above code is actually irrelevant due to how the FAT table was read into memory
	//Code is there as those are the calculations specified by the FAT spec
	return *(unsigned int*)&fatTable[cluster] & 0x0FFFFFFF;
}

//Helper function
//Returns the length of a cluster chain
//Lazy workaround
//Should make the cluster chain into a struct that includes the chain and the length
int clusterChainSize(int cluster, int size)
{
	int value = tableValue(cluster);
	size++;

	if(value >= 0x0FFFFFF8)
	{
		return size;
	}

	return clusterChainSize(value, size);
}

//Returns an array of the cluster chain for a file
unsigned int * clusterChain(int cluster)
{
	int value = tableValue(cluster);
	int size = clusterChainSize(cluster, 0);
	unsigned int * chain = (unsigned int*)malloc(sizeof(unsigned int) * size);
	chain[0] = cluster;

	int i;
	for(i = 1; i < size; i++)
	{
		chain[i] = tableValue(chain[i - 1]);
		//printf("0x%x ", chain[i]);
	}
	//printf("\n");

	return chain;

}

//Read FAT table into memory
void readFatTable(FILE * fd)
{
	fatTable = (uint32_t *)malloc(FATSz * bpb.bpb_bytesPerSec);
	fseek(fd, bpb.bpb_rsvdSecCnt * bpb.bpb_bytesPerSec, SEEK_SET);
	fread(fatTable, sizeof(uint32_t), FATSz * bpb.bpb_bytesPerSec / sizeof(uint32_t), fd);
```

```
	/*
	printf("done reading \n");
	int i;
	printf("length: %d \n", sizeof(fatTable));
	for(i = 0; i < 1000; i++)
	{
		printf("FAT[%d]: 0x%x \n", i, *(unsigned int*)&fatTable[i] & 0x0FFFFFFF);
	}*/
}

//Changes working directory to path
//Path must either be an absolute path starting at the root directory
//or must be a relative path only to an immediate folder
//Use '~' to go back to the root directory
//Use .. to go to back to parent directory
//
//EX:	cd path
//		cd /PEOPLE/BVT2NC		GOOD
//		cd PEOPLE			GOOD
//		cd ~				GOOD
//		cd ~/PEOPLE/BVT2NC		NOT IMPLEMENTED
int OS_cd(const char *path)
{
	if(start == 0)
		init();

	//If path is an absolute path starting at the root directory
	if(path[0] == '/')
	{
		cwdCluster = 2;
		return cdAbsolute(path);
	}

	//If it is a long relative path leading to other directories
	if(strstr(path, "/"))
	{
		return cdAbsolute(path);
	}

	if(path[0] == '~')
	{
		cwdCluster = 2;
		return 1;
	}

	dirEnt * lsDir = OS_readDir("."); //Get all the directories in the cwd
	dirEnt dir;
	int i, compare, j;
	int terminate = 0;
	char * dirNamee;
	char * realPath = (char *)malloc(sizeof(char) * 8);

	//reparse string passed in the arguement 'path' to manipulate characters after null
terminator
```

```
	//EX:
	//Assume MEDIA = 1 2 3 4 5
	//Passing MEDIA to path, it is possible that path = 1 2 3 4 5 0 45 22
	//This for loop will change make realPath = 1 2 3 4 5 32 32 32
	//To fit what we will read from dir.dir_name
	for(i = 0; i < 8; i++)
	{
		if(terminate == 1)
		{
			realPath[i] = 32;
			continue;
		}

		if(path[i] == 0)
		{
			realPath[i] = 32;
			terminate = 1;
		}
		else
			realPath[i] = path[i];
	}

	//Parse through all directories in cwd
	for(i = 0; i < 128; i++)
	{
		dir = lsDir[i];
		if(dir.dir_name[0] == 0) //If last entry
			break;

		dirNamee = dirName(dir, 0); //bad name... was a quick fix due to conflicting
variable and function name
		compare = strcmp(realPath, dirNamee);
		//printf("compare: %d \n", compare);
		if(compare == 0)
		{
			//printf("match! \n");
			cwdCluster = dir.dir_fstClusLO;

			//In the event we go back to the root directory
			if(cwdCluster == 0)
				cwdCluster = 2;

			return 1;
		}

	}

	return -1;
}

//Helper function to change to cwd to an absolute path
int cdAbsolute(const char * path)
{
	int i, status;
```

```
	char * subPath = (char *)malloc(sizeof(char) * 8);
	int index = 0;

	//Parse through path by finding the 'subpath' (.../*THIS_IS_THE_SUBPATH*/...)
	for(i = 0; i < strlen(path); i++)
	{
		if(path[i] == '/') //reached beginning/end of subpath
		{
			//printf("subPath: %s \n", subPath);
			status = OS_cd(subPath);
			index = 0;
			//We are only looking for directories so we don't care about ext
			subPath = (char *)malloc(sizeof(char) * 8);

			if(status == -1)
			{
				return -1;
			}
		}
		else //keep finding more of the subpath
		{
			subPath[index] = path[i];
			index++;
			//printf("%s \n", subPath);
		}
	}

	status = OS_cd(subPath);
	if(status == -1)
		return -1;

	return 1;
}

//Returns the directory name in a compareable format
//Will always be 8 bytes long padded with 32s (inherently as that is what is read from the FAT
disk)
char * dirName(dirEnt dir, int file)
{
	char * str = (char *)malloc(sizeof(char) * 8);
	int i;

	int max = 8;
	if(file == 1)
		max = 11;

	for(i = 0; i < max; i++)
	{
		str[i] = dir.dir_name[i];
		//printf("%d ", str[i]);
	}
	return str;
}
```

```
//Functions in the exact same way as ls
//Path must either be an absolute path starting at the root directory
//or must be a relative path only to an immediate folder
//Use '~' to go back to the root directory
//By default (if dirname is left blank) the shell will make dirname = "."
//
//EX:   ls dirname
//              ls /PEOPLE/BVT2NC           GOOD
//              ls PEOPLE                       GOOD
//              ls ~                            GOOD
//              ls ~/PEOPLE/BVT2NC              NOT IMPLEMENTED
dirEnt * OS_readDir(const char *dirname)
{
        int tempCWD = 0;
        //If absolute path, save the cwdcluster to be loaded back again at end of function and
cd
        if(dirname[0] != '.')
        {
                tempCWD = cwdCluster;
                //printf("tempCWD: %d \n", tempCWD);
                OS_cd(dirname);
        }

        dirEnt* ls = (dirEnt*)malloc(sizeof(dirEnt) * 128);

        if(start == 0)
                init();

        int inc, offset;
        int count = 0;
        dirEnt dir;
        //printf("cwdCluster %d \n", cwdCluster);
        offset = firstClusterSector(cwdCluster) * bpb.bpb_bytesPerSec;

        //Loop through each entry (32 bytes long)
        for(inc = 0; inc < bytesPerClus ; inc += 32)
        {
                fseek(fd, offset + inc, SEEK_SET);
                fread(&dir, sizeof(dirEnt), 1, fd);
                if(dir.dir_name[0] == 0x00) //last entry
                        break;
                if(dir.dir_name[0] == 0xE5)
                        continue;
                if(dir.dir_attr == 8 || dir.dir_attr == 15) //special case
                        continue;

                ls[count] = dir;
                count++;
                //printDir(dir);
        }

        //If there is a cluster chain for the directory
        dirEnt cwd;
        offset = firstClusterSector(cwdCluster) * bpb.bpb_bytesPerSec;
```

```
	fseek(fd, offset, SEEK_SET);
	fread(&cwd, sizeof(dirEnt), 1, fd);

	int length = clusterChainSize(cwd.dir_fstClusLO, 0);
	if(length > 1)
	{
		inc = 0;
		//Loop through each entry (32 bytes long)
		for(inc = 0; inc < bytesPerClus ; inc += 32)
		{
			fseek(fd, offset + inc, SEEK_SET);
			fread(&dir, sizeof(dirEnt), 1, fd);
			if(dir.dir_name[0] == 0x00) //last entry
				break;
			if(dir.dir_name[0] == 0xE5)
				continue;
			if(dir.dir_attr == 8 || dir.dir_attr == 15) //special case
				continue;

			ls[count] = dir;
			count++;
			//printDir(dir);
		}
	}

	//If absolute path was given, set cwdCluster to original
	if(tempCWD != 0)
		cwdCluster = tempCWD;

	//printf("after cwd: %d \n", cwdCluster);

	return ls;
}

//Loads specified file in 'path' as a dirEnt to the global 'openDir'
int OS_open(const char *path)
{
	if(start == 0)
		init();

	//If long relative path (leading to ther directories other than current)
	if(strstr(path, "/"))
	{
		int i, status;
		char * subPath = (char *)malloc(sizeof(char) * 8);
		int index = 0;

		//Parse through path by finding the 'subpath' (.../*THIS_IS_THE_SUBPATH*/...)
		for(i = 0; i < strlen(path); i++)
		{
			if(path[i] == '/') //reached beginning/end of subpath
			{
				//printf("subPath: %s \n", subPath);
				status = OS_cd(subPath);
```

```
                        index = 0;
                        //We are only looking for directories so we don't care about ext
                        subPath = (char *)malloc(sizeof(char) * 8);
                }
                else //keep finding more of the subpath
                {
                        subPath[index] = path[i];
                        index++;
                        //printf("%s \n", subPath);
                }
        }

        path = subPath;
        //printf("%s \n", path);
}

dirEnt * dir = OS_readDir(".");
int i, j;
int terminate = 0;
char * realPath = (char *)malloc(sizeof(char) * 8);

//See cd
//tldr; Reparse the path to get rid of garbage at the end if the short name is smaller than
the max allowed
for(i = 0; i < 11; i++)
{
        if(terminate == 1)
        {
                realPath[i] = 32;
                continue;
        }

        if(path[i] == 0)
        {
                realPath[i] = 32;
                terminate = 1;
        }
        else
                realPath[i] = path[i];
}

//Loop through all the directories in cwd
for(i = 0; i < 128; i++)
{
        if(dir[i].dir_name[0] == 0x00)
        {
                break;
        }

        //printf("dirName: %s \n", dirName(dir[i], 1));
        if(strcmp(realPath, dirName(dir[i], 1)) == 0)
        {

                //Linear search through the global 'openDir' for first open space
```

```
                        for(j = 0; j < 128; j++)
                        {
                                if(openDir[j].dir_name[0] == 0x00)
                                {
                                        openDir[j] = dir[i];
                                        return j;
                                }
                        }
                        //If code gets here, then there were no empty spaces left in the array
                        //That means user attempted to load 128 files into memory without
freeing up space
                        //which isn't a good idea anyway

                        printf("Too many files opened. Please close a file\n");
                        break;
                }
        }

        return -1;
}

//Replaces openDir[fd] with a blank dir that can be used to load new files into memory
int OS_close(int fd)
{
        if(start == 0)
                init();

        dirEnt dir = openDir[fd];
        if(dir.dir_name[0] == 0x00) //If it is already empty, nothing to do
                return -1;
        dirEnt *emptyDir = (dirEnt*)malloc(sizeof(dirEnt)); //empty dirEnt
        openDir[fd] = *emptyDir;

        return 1;
}

//Reads the file specifed by the file descriptor fildes
//The file to be read has its dirEnt stored in openDir
//The file MUST HAVE BEEN PREVIOUSLY OPENED USING OS_open!!!!!
//
//Reads the file starting at the offset and up to nbytes into buf
int OS_read(int fildes, void *buf, int nbyte, int offset)
{
        if(start == 0)
                init();

        dirEnt dir = openDir[fildes];
        if(dir.dir_name[0] == 0x00) //If file location is empty, throw shell error
                return -1;

        //Get the cluster chain and its length
        unsigned int * chain = clusterChain(dir.dir_fstClusLO);
        int length = clusterChainSize(dir.dir_fstClusLO, 0);
```

```
	//If the offset or nbytes to be read is larger than the size (in bytes)
	//the cluster link occupies, send signal to throw error
	if(nbyte > (length * bytesPerClus) || offset > (length * bytesPerClus))
		return -1;

	int bytesRead = 0;
	int firstCluster = offset / bytesPerClus;
	int firstClusterOffset = offset % bytesPerClus;
	int bytesToRead = nbyte; //var used to hold specifically how many bytes to read

	//If there are more bytes to read than what a cluster holds, only read
	//up to the end of the cluster
	if(nbyte > bytesPerClus - firstClusterOffset)
		bytesToRead = bytesPerClus - firstClusterOffset;

	/*printf("firstCluster: %d \n", firstCluster);
	printf("firstClusterOffset: %d \n", firstClusterOffset);
	printf("bytesToRead: %d \n", bytesToRead);
	printf("firstChainCluster: 0x%x \n", (int)chain[firstCluster]);*/

	//Read the data at the offset of the first cluster (after adding in the offset) relative to the
file
	fseek(fd, (firstClusterSector((int)chain[firstCluster]) * bpb.bpb_bytesPerSec) +
firstClusterOffset, SEEK_SET);
	fread(buf, bytesToRead, 1, fd);
	//Increment local variables to continue reading (if needed) the next cluster(s) in the chain
	bytesRead += bytesToRead;
	firstCluster++;
	bytesToRead = nbyte - bytesRead;

	//while there are still bytes left to be read
	while(bytesRead < nbyte || bytesToRead > 0)
	{
		//Make sure we aren't reading in more bytes there are in a cluster again
		if(bytesToRead > bytesPerClus)
		{
			bytesToRead = bytesPerClus;
		}

		//read the data of that cluster into buffer
		fseek(fd, firstClusterSector((int)chain[firstCluster]) * bpb.bpb_bytesPerSec,
SEEK_SET);
		fread(buf + bytesRead, bytesToRead, 1, fd);
		bytesRead += bytesToRead;
		bytesToRead = nbyte - bytesRead;
		firstCluster++;
	}

	return 1;
}
```

```
//
==========================================================WRITE=========
=======================================
```

```
int OS_rmdir(const char *path)
{
	return -1;
}
int OS_mkdir(const char *path)
{
	return -1;
}
int OS_rm(const char *path)
{
	return -1;
}
int OS_creat(const char *path)
{
	return -1;
}
int OS_write(int fildes, const void *buf, int nbyte, int offset)
{
	return -1;
}
```