Machine Problem 1 – Fat16/32 Read API

Gabriel Groover (gtg3vv)

2/12/18

Problem

The goal of this assignment is to help us understand the FAT filesystem by implementing a read-only api for FAT16/32 volumes. We will be providing functions that allow the user to read and switch between directories and files. In doing so, we will learn how files are organized on disk and abstracted into file systems that allow the user to manage data. The functions we implement will provide a foundation for our write api that we will implement in the next assignment.

I was able to implement each of the functions, read, write, open, close, cd and readdir successfully for both FAT16 and FAT32. The major performance limitations include memory management and disk access. I chose not to read the entire FAT into memory, and instead read each individual entry from the disk. If any allocated memory was not properly freed after use, the program could eventually fail.

Approach

I chose to implement this assignment in C++ in order to take advantage of standard libraries and easier memory management. I included each of the structs and method headers in the provided myfat.h file as well adding several new ones of my own. I also chose to include support for both FAT16/32 file systems. I implemented all of my methods in fatdriver.cpp and compiled them into libFAT16/32.so accordingly.

I broke the assignment down into parts and developed each of the following components:

FAT Volume Initialization – Before performing any actual operations on the FAT file system, I needed to initialize the FAT volume from the boot sector. I used the existing Fat32 struct and read

directly from the start of the FAT volume into that struct because the values were packed. I chose to keep this struct as a global pointer to avoid having to read from disk and initalize the volume multiple times.

There are several essential calculations I performed in this section. The first of these are cluster size and first data sector. Cluster size will be used throughout the operation of the file system to read in chunks of data and locate directory entries on disk. The first data sector will be used to determine byte offsets for data on the disk. I also calculated the total count of clusters on the disk according to the formula in the FAT specifications. This result was used to distinguish FAT16/32 volumes so that I could support both. After determining the FAT type, I was able to pull the root directory cluster number directly from the boot block for FAT32 volumes, and calculate the cluster number based on the byte offset for FAT16 volumes.

OS_readDir – This was the first piece of functionality I wrote. I created several helper functions to assist in path manipulation and FAT access. The In order to limit disk accesses, I read in full clusters at a time and used getFATEntry to determine if the next cluster was part of the same directory. During each iteration, I kept track of the current cluster number and if there was a continuation cluster stored in the FAT. I also iterated through each of the directory entries in each cluster to determine if they matched the portion of the path I was currently searching for. If I hit an end of directory marker without finding the next portion of my path, I knew to return a null pointer. I handled relative paths by first checking if the supplied path began with a "/." If it did not, I could be sure the path was relative and prepend the current working path. Handling relative paths in this manner meant that I could treat all paths as absolute and just work from the root directory downwards.

OS_cd – A correct implementation of read directory allows the implementation of change directory to be extremely straight forward. I first call readDir on the path passed into cd to check if that path

actually exists and is to a directory. The function is then just string manipulation. If the given path was absolute, I overwrote the existing working path. If it was relative, I prepended the current working path to the beginning of the new path.

OS_open/close – For the purposes of a read-only api, I chose to implement my open file table in a minimal way to reduce overhead. I created an array of 128 int pointers to keep track of the cluster number for an open file and used the array index to represent the file descriptor. In order to open a file, I first separated the filename from its containing directory based on the last path delimiter. I could then use readDir to verify that the directory exists and searched through its results for the filename. If I was able to locate the file, I searched through my open file table for an open space and placed the cluster number of that file into the table. I closed each file by checking to see if the entry in the array for that file descriptor was non-zero, and resetting it if it was.

OS_read – Reading a file required first checking if the had previously been opened by reading the entry for the given file descriptor from the open file table. I next found the correct start cluster for my read by determining the number of clusters the provided offset was from the start of the file and following the cluster chain in the FAT until I had reached the correct cluster number. I then followed a simple algorithm to complete my read.

- Keep track of the number of bytes read, and the number left to read
- If the bytes left to read is less than the space left in the current cluster, read those into the buffer and stop
- Otherwise, read up to the end of the cluster, and repeat with the next cluster

Results/Analysis

I was able to implement both FAT16 and FAT32. For the most part, they are identical. They differ only in their initialization and a few constants used during the file system's operation. After implementing FAT32, adding support for FAT16 turned out to be trivial. I successfully linked with the tester and each of the functions performed as I expected. Testing my code in the shell did reveal some unexpected edge cases such as using relative paths back into the root directory. The root directory does not have /. or /.. directories, and all relative directories that point to the root have cluster numbers of 0. I had to handle each of the edge cases manually.

The shell also revealed some unusual behavior that seemed to be intended. The provided shell kept track of opened files on the back end and associated them with the path used to open them. Our api just returns a file descriptor that could then be used to reference a file for a read, but the shell doesn't allow the user to directly manipulate the descriptor. Consequently, the same file opened from different paths will create two different file descriptors. The user also cannot read from a file that is open using a different path than the one used to open it.

My code actually performed much better than expected. I tried to limit the amount of times I accessed disk by reading in whole clusters at a time. I only accessed the disk in order to retrieve FAT entries. I chose not to read something as large as the FAT into memory. All of the commands I tried to run were seemingly instantaneous. However, the test file system we used had very few files or directories that were actually more than a cluster long. My program wasn't forced to query the disk very many times, regardless of the operation I was performing. In a context with much larger files that require more FAT accesses and cluster reads, the performance would be a lot worse.

Most of the problems that I encountered developing this library were from discomfort in C or difficulty with an assignment of this scale. Each of the functions we had to develop were dependent on

the other functions in the library. I found that when I made an error in one function, it would break all of the others. I also found that I had to plan my implementation very carefully. We had complete flexibility in our implementation, but certain design decisions early on made later features more difficult. I originally handled cd by storing a directory pointer in memory and using that as the starting point for my read, but this made handling relative paths extremely difficult. My simplified open file table will also not be adequate for the write api we will have to be implement and I should have considered a different approach.

It is also important to consider things like performance and memory management early on in the development process. I was careful to free all of the memory I allocated, but I did this later than I should have and it was difficult to go back and add. If I missed any it would affect my api's ability to operate for a prolonged period.

Conclusion

Developing the read portion of the FAT api was difficult because of the scale of the assignment and because of my unfamiliarity with C. Throughout the course of the assignment I became much more comfortable with memory management and string manipulation in C. I also gained a much better understanding of the FAT file system, including the layout of the volume and how cluster chains are really just linked lists. I will be able to use this to plan and implement my write api more effectively in the next assignment.

On my honor I have not given or received aid on this assignment or report.

fatdriver.cpp

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW1 - FileSystem  Read API
 *
 */

//User head files
#include "myfat.h"

//C++ libraries
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <queue>
#include <locale>
#include <algorithm>

//Sys call dependent includes
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

//Initialize file system from boot block
void initFS()
{
  const char *path = std::getenv("FAT_FS_PATH");
  fatFd = open(path, O_RDONLY);
  lseek(fatFd,0,SEEK_SET);
  bB = (bpbFat32*) malloc(sizeof(bpbFat32));
  read(fatFd, bB, sizeof(bpbFat32));
  calcRootDir();
}

//Calculate Root Dir location and initalize other boot block parameters
void calcRootDir()
{
  RootDirSectors = round(((bB->bpb_rootEntCnt * 32) + (bB->bpb_bytesPerSec - 1)) / bB->bpb_bytesPerSec);

  clusSize = bB->bpb_secPerClus * bB->bpb_bytesPerSec;

  //Determine FAT size
  uint32_t FATSz;
  if (bB->bpb_FATSz16 != 0)
    FATSz = bB->bpb_FATSz16;
  else
    FATSz = bB->bpb_FATSz32;

  //Locate beginning of data
  FirstDataSector = bB->bpb_rsvdSecCnt + (bB->bpb_numFATs * FATSz) + RootDirSectors;
```

```
  //FAT Type Determination
  uint32_t TotSec;
  if (bB->bpb_totSec16 != 0)
    TotSec = bB->bpb_totSec16;
  else
    TotSec = bB->bpb_totSec32;

  uint32_t DataSec = TotSec - (bB->bpb_rsvdSecCnt + (bB->bpb_numFATs * FATSz) + RootDirSectors);
  uint32_t CountofClusters = floor(DataSec / bB->bpb_secPerClus);
  if (CountofClusters < 4085) {/* FAT 12 */}
  else if (CountofClusters < 65525) {
    std::cout << "Volume is fat16" << std::endl;
    fatType = 16;
  }
  else {
    std::cout << "Volume is fat32" << std::endl;
    fatType = 32;
  }

  //Initialize Values based  on fat type
  if (fatType == 16)
  {
    rootClusNum = 2;
    eofVal = 0xFFF8;
    badClus = 0xFFF7;

    //Calculate root cluster based on offset from cluster formula
    int rootOffset = (FirstDataSector - RootDirSectors) * bB->bpb_bytesPerSec;
    int clusOffset = (((-2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec;
    rootClusNum = (rootOffset - clusOffset) / clusSize;
  } else if (fatType == 32)
  {
    rootClusNum = bB->bpb_RootClus;
    eofVal = 0x0FFFFFF8;
    badClus = 0x0FFFFFF7;
  }

  //Set default path to root dir
  cwdPath = (char*) malloc(2);
  cwdPath[0] = '/';
  cwdPath[1] = '\0';
}

//Read a single directory from a path
dirEnt * OS_readDir(const char *dirname)
{
  //Check initialization
  if (!initialized)
  {
    initFS();
    initialized = true;
  }
```

```
  std::string pathName(dirname);
  std::string desiredEntry = getNextPathName(pathName);
  uint32_t currentClus = rootClusNum;
  bool hasNextCluster = true;
  bool endOfDirectory = false;
  bool rootDir = false;

  //Check for relative path and add cwd if necessary
  if (dirname[0] != '/')
  {
    if (!cwdPath)
      return nullptr;

    std::string currentPath(cwdPath);
    if (currentPath[currentPath.size()-1] != '/')
      pathName.insert(0, "/");
    pathName.insert(0,  currentPath);
    return OS_readDir(pathName.c_str());

  } else if (stringCompare(pathName, "/") != 1)
      pathName = pathName.substr(desiredEntry.size()+1,pathName.size());
  else
    rootDir = true;

  //Continue iterating until reached the end of a directory
  while (hasNextCluster && !endOfDirectory)
  {
    dirEnt *dir = (dirEnt*) malloc(clusSize);
    lseek(fatFd, (((currentClus - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec ,
SEEK_SET);
    read(fatFd, dir, clusSize);

    hasNextCluster = getFATEntry(currentClus) < eofVal;
    currentClus = getFATEntry(currentClus);

    //std::cout << desiredEntry << " " <<  pathName << std::endl;

    for (int i = 0; i < 64; i ++)
    {
      std::string entryName = strip(dir[i].dir_name);
     // std::cout << entryName << std::endl;

      //End of directory marker
      if (dir[i].dir_name[0] == 0)
      {
            //entry not found
            endOfDirectory = true;

            if (rootDir)
              return dir;
            free(dir);
            return nullptr;
            break;
```

```
      }
      //If found desired entry and it is actually a directory
      if (stringCompare(entryName,desiredEntry) == 1 && (dir[i].dir_attr & 0x10))
      {
            std::cout << "Located " << desiredEntry << std::endl;

            unsigned int highWord = (unsigned int) dir[i].dir_fstClusHI << 16;
            unsigned int lowWord = (unsigned int) dir[i].dir_fstClusLO;

            currentClus = highWord | lowWord;
            hasNextCluster = true;

            //Check root .. case
            if (currentClus == 0)
              currentClus = rootClusNum;

            //If path complete, return directory pointer
            if (pathName.size() <= 0 || (pathName.size() == 1  && pathName[0] == '/'))
            {
              free(dir);
              return readClusForCurrentDir(currentClus);
            }

            desiredEntry = getNextPathName(pathName);
            pathName = pathName.substr(desiredEntry.size()+1,pathName.size());
           break;
      }
    }

    free(dir);
   }
   return nullptr;
}

//Handle reading clusters for final return
dirEnt * readClusForCurrentDir(uint32_t startClus)
{
    std::queue<uint32_t> clusters;
    clusters.push(startClus);

    //Add all clusters for current directory to queue
    while (getFATEntry(startClus) < eofVal)
    {
      startClus = getFATEntry(startClus);
      clusters.push(startClus);
    }

    //Read each cluster from queue into dir*
    unsigned int clusCount = clusters.size();
    dirEnt *dir = (dirEnt*) malloc(clusCount * clusSize);
    unsigned int dirOffset = 0;
    while (!clusters.empty())
    {
```

```
    uint32_t offset = (((clusters.front() - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec;
    clusters.pop();

    lseek(fatFd, offset , SEEK_SET);
    read(fatFd, dir + dirOffset, clusSize);
    dirOffset += clusSize;
   }

   return dir;
}


int OS_open(const char* path)
{
 //Check initialization
 if (!initialized)
 {
  initFS();
  initialized = true;
 }

 //extract filename from path
 std::string s(path);
 if (s[s.size()-1] == '/')
  s = s.substr(0, s.size()-1);
 size_t lastDelim = s.find_last_of('/');
 std::string fileName;
 if (lastDelim != std::string::npos)
  fileName = s.substr(lastDelim+1, s.size());
 else fileName = s;

 char pathSlice[lastDelim+2];
 strncpy(pathSlice, path, lastDelim+1);
 pathSlice[lastDelim+1] = '\0';

 //std::cout << pathSlice << " " << fileName  << std::endl;

 //If directory exists, search it for file
 dirEnt* dir;
 dir = OS_readDir(pathSlice);
 if (dir)
 {
  int i = 0;
  while (1)
  {
   std::string entryName = strip(dir[i].dir_name);
   //std::cout << entryName << "#" << std::endl;

   //End of directory marker found
   if (dir[i].dir_name[0] == 0)
   {
        //entry not found
        std::cout << "last entry in directory" << std::endl;
```

```
            return -1;
            break;
        }
        //Found entry matching  filename that isnt directory
        if (stringCompare(entryName,fileName) == 1 && !(dir[i].dir_attr & 0x10))
        {
            std::cout << "Located " << fileName << std::endl;

            unsigned int highWord = (unsigned int) dir[i].dir_fstClusHI << 16;
            unsigned int lowWord = (unsigned int) dir[i].dir_fstClusLO;
            unsigned int fileCluster = highWord | lowWord;

            //Check file table for open space
            for (int j = 0; j < 127; j ++)
            {
              if (fileTable[j] == 0)
              {
                fileTable[j] = fileCluster;
                return j;
              }
            }
            return -1;
        }
        i++;
      }
    }
    return -1;
}

//Read from an open file
int OS_read(int fildes, void *buf, int nbyte, int offset)
{
    //Handle Initialization
    if (!initialized)
    {
      initFS();
      initialized = true;
    }
    //Check to see file is actually open
    if (fildes < 0 || fildes > 127 || fileTable[fildes] == 0)
      return -1;

    //Determine offset from start cluster of file
    unsigned int startClus = offset / clusSize;
    unsigned int currentClus = fileTable[fildes];
    int bytesRead = 0;
    int bytesLeft = nbyte;

    //Iterate clusters to correct part of file
    while (startClus > 0)
    {
      currentClus = getFATEntry(currentClus);
      if (currentClus >= eofVal)
```

```
      return -1;
    startClus--;
  }

  //Check to see if there is enough space in the cluster to read remaining bytes
  offset = offset % clusSize;
  lseek(fatFd, ((((currentClus - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec) + offset ,
SEEK_SET);
  if (clusSize - offset > bytesLeft)
  {
    read(fatFd, buf, nbyte);
    return nbyte;
  }
  else
  {
    read(fatFd, buf, clusSize - offset);
    bytesRead = clusSize - offset;
    bytesLeft -= bytesRead;
    currentClus = getFATEntry(currentClus);
  }

  //Continue reading next cluster until all bytes read, or end of file reached
  while(bytesLeft > 0)
  {
    if (currentClus >= eofVal)
      return bytesRead;

    lseek(fatFd, ((((currentClus - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec),
SEEK_SET);
    if (bytesLeft >= clusSize)
    {
      read(fatFd, buf+bytesRead, clusSize);
      bytesLeft -= clusSize;
      bytesRead += clusSize;
    } else
    {
      read(fatFd, buf+bytesRead, bytesLeft);
      bytesLeft -= bytesLeft;
      bytesRead += bytesLeft;
    }
  }
  return bytesRead;
}

//Close an existing file descriptor
int OS_close(int fd)
{
  //Check initialization
  if (!initialized)
  {
    initFS();
    initialized = true;
  }
```

```
  //Close file if it exists
  if (fileTable[fd] != 0)
  {
    fileTable[fd] = 0;
    return 1;
  } else
    return -1;
}

//Compare filesystem name (s1) to user supplied path (s2)
int stringCompare(std::string s1, std::string s2)
{
  //Remove . if not relative path, uppercase all dirent names
  s1.erase(std::remove(s1.begin(), s1.end(), ' '), s1.end());
  if (s2[0] != '.')
    s2.erase(std::remove(s2.begin(), s2.end(), '.'), s2.end());

  std::locale loc;

  //Compare sizes and check remaining chars for equality
  if (s1.size() != s2.size())
    return -1;
  for (int i = 0; i < s1.size(); i++)
  {
    if (std::toupper(s2[i],loc) != s1[i])
      return -1;
  }
  return 1;
}

//Change current working directory to path
int OS_cd(const char *path)
{
  //Check initialization
  if (!initialized)
  {
    initFS();
    initialized = true;
  }

  //Check directory exists
  void* dir = OS_readDir(path);
  if (dir)
  {
    free(dir);
    //Check if path  is absolute and ovewrite cwd
    if (path[0] == '/')
    {
      if (cwdPath)
            free(cwdPath);
      cwdPath = (char*) malloc(strlen(path)+1);
      strcpy(cwdPath, path);
```

```
      cwdPath[strlen(path)] = '\0';

      std::cout << "New Working Dir: " << cwdPath << std::endl;
      return 1;
    }
    //If path is relative, append to end of cwdPath
    else
    {
      if (!cwdPath)
            return -1;
      //Add trailing  / to cwdPath
      if (cwdPath[strlen(cwdPath)-1] != '/')
      {
            char* tempPath = (char*) malloc(strlen(path) + strlen(cwdPath) + 2);
            strcpy(tempPath+strlen(cwdPath)+1, path);
            strcpy(tempPath, cwdPath);
            tempPath[strlen(cwdPath)] = '/';
            tempPath[strlen(path) + strlen(cwdPath) + 1] = '\0';

            free(cwdPath);
            cwdPath = tempPath;
            std::cout << "New Working Dir: " << cwdPath << std::endl;
            return 1;
      }
      //cwdPath already has trailing /
      else
      {
            char* tempPath = (char*) malloc(strlen(path) + strlen(cwdPath) + 1);
            strcpy(tempPath, cwdPath);
            strcpy(tempPath+strlen(cwdPath), path);
            tempPath[strlen(path) + strlen(cwdPath)] = '\0';

            free(cwdPath);
            cwdPath = tempPath;
            std::cout << "New Working Dir: " << cwdPath << std::endl;
            return 1;
      }
    }
    return 1;
  } else return -1;
}

//Get next / delimited element of path
std::string getNextPathName(const std::string& s)
{
  size_t firstDelim = s.find_first_of('/');
  size_t nextDelim = s.find_first_of('/',firstDelim+1);
  if (nextDelim == std::string::npos)
    return s.substr(firstDelim+1, s.size() - firstDelim);
  return s.substr(firstDelim+1, nextDelim - firstDelim - 1);
}
```

```
//Remove trailing  whitespace
std::string strip(uint8_t* dirName)
{
  char* str;
  for (int i = 10; i >= 0; i--)
  {
    if (dirName[i] != 32)
    {
      str = (char*) malloc(i+2);
      strncpy(str, (const char*)dirName, i+1);
      str[i+1] = 0;
      break;
    }
  }
  std::string s(str);
  free(str);

  return s;
}


//Get entry in fat table for a given index
unsigned int getFATEntry(int n)
{
  //Fat offset determination
  uint32_t fatoffset;
  if (fatType == 16)
    fatoffset = n * 2;
  else
    fatoffset = n * 4;

  uint32_t fatSecNum = bB->bpb_rsvdSecCnt + (fatoffset / bB->bpb_bytesPerSec);
  uint32_t fatEntOffset = fatoffset % bB->bpb_bytesPerSec;

  //Read fat entry
  lseek(fatFd, fatSecNum * bB->bpb_bytesPerSec ,SEEK_SET);
  unsigned char buffer[bB->bpb_bytesPerSec];
  read(fatFd, buffer, bB->bpb_bytesPerSec);

  //fat 16 extraction
  if (fatType == 16)
    unsigned short table_value = *(unsigned short*) &buffer[fatEntOffset];
  //fat 32 extraction
  else
    unsigned int table_value = *(unsigned int*) &buffer[fatEntOffset] & 0x0FFFFFFF;
}

//Pre-defined functions

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;
}
```

myfat.h

```
#ifndef __MYFAT_H__
#define __MYFAT_H__
#include "stdint.h"
#include <string>
#include <vector>

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 NT
	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 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 of 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 "C" int OS_cd(const char *path);
extern "C" int OS_open(const char *path);
extern "C" int OS_close(int fd);
extern "C" int OS_read(int fildes, void *buf, int nbyte, int offset);
extern "C" dirEnt * OS_readDir(const char *dirname);
extern "C" int OS_rmdir(const char *path);
extern "C" int OS_mkdir(const char *path);
extern "C" int OS_rm(const char *path);
extern "C" int OS_creat(const char *path);
extern "C" int OS_write(int fildes, const void *buf, int nbyte, int offset);

void initFS(); //Initialize file system boot block
void calcRootDir(); //Calculate root dir location from boot block
unsigned int  getFATEntry(int n); //Determine offset for fat entry n
std::string strip(uint8_t *dirName); //strip trailing whitespace
std::string getNextPathName(const std::string& s); //get next element of path
int stringCompare(std::string s1, std::string s2);
dirEnt * readClusForCurrentDir(uint32_t startClus); //Return pointer to final directory

char *cwdPath;		// current working dir name
dirEnt *currentDir;
int fdCount;
bool initialized;
int fatType;
int fatFd; //file descriptor for opened fat volume
bpbFat32* bB; //boot block values
uint32_t FirstDataSector; //first data sector of  volume relative  to 0
uint32_t RootDirSectors; //num sectors  of root directory
```

```
uint32_t clusSize; //cluster size in bytes
uint32_t rootClusNum; //clusternum of root directory
uint32_t eofVal; //end of file value for each fat type
uint32_t badClus; //bad cluster value

unsigned int fileTable[128];


#endif
```