Machine Problem 2 – Fat16/32 Write API

Gabriel Groover (gtg3vv)

3/1/18

Writeup:

Comments:

Code:

I was able to successfully implement each of the specified functions with support for both FAT16 and FAT32 volumes.

Problem

The purpose of this assignment was to write the counterpart to our read-only api for FAT16/32 volumes by implementing each of the functions, mkdir, rmdir, creat, rm, and write. The library should be able to be used in conjunction with a working solution to machine problem one. As with the read-only api, it was important to be careful of memory management and disk access to ensure the program ran correctly and efficiently.

Approach

The code for this assignment was written in C++ in order to take advantage of standard libraries and to be compatible with the existing read api. Each of the structs and method headers in the provided myfat.h file were included, and the write functionality was added in the same fatdriver.cpp file from the first assignment.

As part of the creation of a FAT write api, each of the following components were developed:

FAT 16/32 Read Api – Each of the components of the original read api were used in this assignment, with minor modifications that will be discussed later. The read api handled FAT volume initialization,

directory navigation and file reading. The write api also takes advantage of several helper functions that were implemented as part of the read api solution.

OS_mkdir(const char *path) – This function first extracts the desired directory name from the remainder of the path and calls OS_readdir on its parent. This allows the function both to determine if the first part of the path is valid, and to operate on the dirEnts in the result of the read operation. This pattern will hold true for each of OS_rmdir, OS_creat and OS_rm as they all require locating an element of the path in its parent directory. The function then iterates through the entries in the result of the OS_readdir and returns the appropriate failure conditions if the path is invalid or the desired directory name exists.

If none of the failure conditions are met, the function will construct a new directory entry that points to an open cluster in the FAT. Dot and Dot Dot directories are then written into that cluster along with an end of directory marker to ensure the user can correctly navigate through the new directory. If the new directory causes the original parent directory to be full, a new cluster is allocated to that directory and is added to the existing cluster chain in the FAT. Additions can still be made to a full cluster of directory entries without allocating a new cluster if any entry in that cluster is marked as open. The new directory will take the place of the open entry.

OS_creat(const char *path) – This function is almost identical to OS_mkdir with the notable exceptions that it does not label the newly created directory entry as a directory, and does not construct Dot and Dot Dot directories for the allocated cluster. Otherwise, it performs the same operations as OS_mkdir and fails under the same conditions.

OS_rmdir(const char *path) – After locating the final element of the path using the approach described above, this function checks to see if that element is an empty directory by calling OS_readdir a second time on the complete path and iterating through the results. The function next iterates through

the cluster chain for the directory and marks each cluster as free in the FAT. Finally it writes the open directory marker back to the directory name so that future directory entries can overwrite its position in the cluster.

OS_rm(const char *path) – This function is almost identical to OS_rmdir, because it performs the exact same operation, but doesn't need to check if the file is empty. Otherwise, it has the same failure conditions as OS_rmdir.

OS_write(int fildes, const void *buf, int nbyte, int offset) – Writing a file required first checking if the file had previously been opened by reading the entry for the given file descriptor from the open file table. The function then finds the correct start cluster for the write operation by determining how far the provided offset is from the start of the file, and following the cluster chain in the FAT until reaching the correct cluster number.

The following algorithm is then used to complete the write:

- Keep track of the number of bytes written, and the number left to write
- If the bytes left to write is less than the space left in the current cluster, write those from the buffer onto disk and stop
- Otherwise, write up to the end of the cluster, and repeat with the next cluster
- If there are no allocated clusters remaining, allocate a new cluster and add it to the cluster chain

Before returning the number of bytes written, the function updates the filesize of the directory entry that corresponds to the open file to reflect the new length in bytes. This required modifying the open file table to keep track of the directory entry offset on disk so that the function could write back to the correct location.

Helper Functions

dirEnt* buildNewDirEnt(std::string fileName, int type) – This function simplifies the creation of directory entries and is employed multiple times by OS_mkdir and OS_creat.

unsigned int findFreeFATEntry() - This function iterates through the FAT (beginning at the offset specified in fs_info for fat32) and locates the first free cluster marker.

unsigned int setFATEntry(int row, int val) – This function is the logical counterpart to the getFATEntry function written for the read api. It allows the user to update the FAT to reflect new clusters allocated in a chain and is often used in conjunction with findFreeFATEntry.

Results/Analysis

All of the write operations were implemented successfully for both FAT16 and FAT32. Most of the differences between the two are only relevant in the initialization of various parameters when setting up the volume. Each operation is dependent on the read portion the the api developed for homework 1. Several edge cases such as reading directories that were larger than a cluster were difficult to test before adding write functionality. The creatnfile function provided by the tester revealed that the original implementation of OS_readdir had incorrect pointer arithmetic and never returned more than a single cluster of a directory.

The test shell also revealed some unusual behavior with rm operations. The operation would hang when trying to remove directories or files that were longer than a cluster because the first part of the cluster chain was being cleared in the FAT before saving the later values. Creatnfile would also fail when overflowing a directory by writing to the wrong cluster and overwriting the current Dot directory. Each of these issues were simple to fix.

Overall, the code performed much better than expected. There were substantially more disk accesses required than in the read portion of the homework. Each function would read from the disk

and then write the result back to the FAT and the containing directory. Even with all of these disk accesses, there was no noticeable slowdown. The problem would become far worse with files larger than those in the sample volumes that require more FAT accesses and cluster reads. Helper functions that performed these steps made the code much simpler.

Unlike the first assignment, most of the problems in development were not from discomfort in C. Developing the read api reinforced many of the concepts required for the write portion of the assignment. Instead, the difficulty came from the sheer amount of code required. It quickly became apparent how many of the write functions relied on similar steps. Files and directories are represented very similarly on FAT file systems, and write functions manipulate both of them the same way. This made the design of the library very straight forward, but meant that there was a lot of unnecessary code recycling. More of this repeated functionality should have been broken out into helper functions if time allowed. This would have substantially shortened the code base and made it easier to debug each individual function.

Conclusion

Developing the write portion of the FAT api was difficult mainly because of the scale of the assignment. The code became extremely long and interdependent. Any errors in functions like OS_readdir could break several other functions. It was extremely important to divide up functionality, comment thoroughly, and test iteratively.

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

```
/*
 * Gabriel  Groover (gtg3vv)
 * HW2 - FileSystem  Write API
 * Due: 3/1/2018
 */

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

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

//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_RDWR);
  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] == 0x00)
   {
        //entry not found
        endOfDirectory = true;

        if (rootDir)
```

```
            {
              lastClusRead = rootClusNum;
              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);
              lastClusRead = currentClus;
              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 / sizeof(dirEnt);
    }
    return dir;
}


//Open a file at the given path
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 startClus = sizeof(dirEnt) * i / clusSize;

        //Iterate clusters to correct part of directory
        while (startClus > 0)
        {
          lastClusRead = getFATEntry(lastClusRead);
          if (lastClusRead >= eofVal)
            return -1;
          startClus--;
        }

        unsigned int offsetForDirEnt = (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB-
>bpb_bytesPerSec + (sizeof(dirEnt) * i % clusSize);


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

//determine start cluster from a dirent offset
unsigned int getClusFromDirOffset(int offset)
{

   lseek(fatFd, offset, SEEK_SET);
```

```
    dirEnt* thisDir = (dirEnt*) malloc(sizeof(dirEnt));
    read(fatFd, thisDir, sizeof(dirEnt));

    unsigned int highWord = (unsigned int) thisDir->dir_fstClusHI << 16;
    unsigned int lowWord = (unsigned int) thisDir->dir_fstClusLO;
    unsigned int fileCluster = highWord | lowWord;

    free(thisDir);
    return fileCluster;
}

//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 = getClusFromDirOffset(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);
      bytesRead += clusSize;
      bytesLeft -= clusSize;
    } else
    {
      read(fatFd, buf+bytesRead, bytesLeft);
      bytesRead += bytesLeft;
      bytesLeft -= 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;
}

//Set a value in the fat
unsigned int setFATEntry(int row, int val)
{
  //Fat offset determination
  uint32_t fatoffset;
  if (fatType == 16)
    fatoffset = row * 2;
  else
    fatoffset = row * 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);

  if (fatType == 16)
    *((unsigned short*) &buffer[fatEntOffset]) = val;
  else {
      val &= 0x0FFFFFFF;
      *((unsigned int *) &buffer[fatEntOffset]) = (*((unsigned int*) &buffer[fatEntOffset])) & 0xF0000000;
```

```
    *((unsigned int*) &buffer[fatEntOffset]) = (*((unsigned int*) &buffer[fatEntOffset])) | val;
  }
  lseek(fatFd, fatSecNum * bB->bpb_bytesPerSec ,SEEK_SET);
  write(fatFd, buffer, bB->bpb_bytesPerSec);

  return val;
}

//Locate an open cluster
unsigned int findFreeFATEntry()
{
  uint16_t infoSector;
  unsigned int next_free;
  if (fatType == 32 && bB->bpb_FSInfo != 0xFFFFFFFF)
  {
    infoSector = bB->bpb_FSInfo;
    unsigned char buffer[4];
    lseek(fatFd, infoSector * bB->bpb_bytesPerSec + 492, SEEK_SET);
    read(fatFd, buffer, 4);

    next_free = *(unsigned int*) &buffer;
  } else
     next_free = 2;

  while (getFATEntry(next_free) != 0)
    next_free++;

  return next_free;
}

//Remove directory at the  specified path
int OS_rmdir(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;
 dirEnt* dir = OS_readDir(pathSlice);
 if (dir)
 {
  int i = 0;
  bool foundDir = false;
  while (dir[i].dir_name[0] != 0x00)
  {
   std::string entryName = strip(dir[i].dir_name);
   //std::cout << strip(dir[i].dir_name) << " " << i << std::endl;
   //Found matching pre-existing directory
   if (stringCompare(entryName,fileName) == 1 && (dir[i].dir_attr & 0x10))
   {
        //std::cout << "Located Directory to remove" << std::endl;
        foundDir = true;
        break;
   }
   i++;
  }
  if (!foundDir)
  {
   free(dir);
   //std::cout << "Coudn't locate desired dir" << std::endl;
   return -2;
  }
  //Store parent cluster for later
  int temp = lastClusRead;


  //Check if dir is empty
  dirEnt* dirToRemove = OS_readDir(path);
  int j = 0;
  int countNotFree = 0;
  while (dirToRemove[j].dir_name[0] != 0x00)
  {
   if (dirToRemove[j].dir_name[0] != 0xE5)
        countNotFree++;
   j++;
  }
  if (countNotFree > 2)
  {
   free(dir); free(dirToRemove);
   //std::cout << "Dir not empty" << std::endl;
   return -3;
  }

  //Check how many clusters long directory is
  lastClusRead = temp;
  int clusNum = i * sizeof(dirEnt) / clusSize;
  while (clusNum > 0)
  {
   lastClusRead = getFATEntry(lastClusRead);
```

```
      //std::cout << clusNum << " " << lastClusRead << std::endl;
      clusNum--;
    }

    //Free cluster chain
    unsigned int highWord = (unsigned int) dir[i].dir_fstClusHI << 16;
    unsigned int lowWord = (unsigned int) dir[i].dir_fstClusLO;
    temp = highWord | lowWord;
    int next = getFATEntry(temp);
    setFATEntry(temp,0);
    while (next < eofVal)
    {
      temp = next;
      setFATEntry(temp,0);
      next = getFATEntry(next);
    }

    //write to new  directory
    dir[i].dir_name[0] = 0xE5;
    int offset = i * sizeof(dirEnt) % clusSize;
    lseek(fatFd, (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec + offset,
SEEK_SET);
    write(fatFd, &dir[i], sizeof(dirEnt));

    free(dir); free(dirToRemove);
    //std::cout << "removed  dir" << std::endl;

  } else return -1; // Path invalid
  return 1;
}

//Make a directory at the specified path
int OS_mkdir(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;
  dirEnt* dir = OS_readDir(pathSlice);
  if (dir)
  {
    int i = 0;
    bool foundOpenDir = false;
    while (dir[i].dir_name[0] != 0x00)
    {
      std::string entryName = strip(dir[i].dir_name);
      //std::cout << strip(dir[i].dir_name) << " " << i << std::endl;
      //Found matching pre-existing directory
      if (stringCompare(entryName,fileName) == 1)
      {
            //std::cout << "Directory already exists" << std::endl;
            free(dir);
            return -2;
      }

      if(dir[i].dir_name[0] == 0xE5)
      {
            foundOpenDir = true;
            break;
      }
      if (!foundOpenDir)
            i++;
    }

    //Check how many clusters long directory is
    int clusNum = i * sizeof(dirEnt) / clusSize;
    while (clusNum > 0)
    {
      lastClusRead = getFATEntry(lastClusRead);
      clusNum--;
    }

    //Build new dirent
    dirEnt* newDir = buildNewDirEnt(fileName, DIRECTORY);
    int x = findFreeFATEntry();
    setFATEntry(x, -1);
    newDir->dir_fstClusHI = (x & 0xFFFF0000) >> 16;
    newDir->dir_fstClusLO = (x & 0x0000FFFF);

    //write to new  directory
    int offset = i * sizeof(dirEnt) % clusSize;
    lseek(fatFd, (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec + offset,
SEEK_SET);
    write(fatFd, newDir, sizeof(dirEnt));

    //Clear new cluster
    char* memsetBuffer = (char*) malloc(clusSize);
    memset(memsetBuffer, 0, clusSize);
```

```
    lseek(fatFd, (((x - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec, SEEK_SET);
    write(fatFd, memsetBuffer, clusSize);

    //Update clusters in fat table
     int dirsPerClus = clusSize / sizeof(dirEnt);
    if ((i % dirsPerClus) + 2 > dirsPerClus)
    {
      int addClus = findFreeFATEntry();
      setFATEntry(lastClusRead, addClus);
      setFATEntry(addClus, -1);
      lastClusRead = addClus;
      offset = 0;
    } else
      offset += sizeof(dirEnt);

    //Write . directory
    dirEnt* newDotDir = buildNewDirEnt(".", DIRECTORY);
    newDotDir->dir_fstClusHI = (x & 0xFFFF0000) >> 16;
    newDotDir->dir_fstClusLO = (x & 0x0000FFFF);
    lseek(fatFd, (((x - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec, SEEK_SET);
    write(fatFd, newDotDir, sizeof(dirEnt));

    //Write .. directory
    dirEnt* newDotDotDir = buildNewDirEnt("..", DIRECTORY);
    newDotDotDir->dir_fstClusHI = (lastClusRead & 0xFFFF0000) >> 16;
    newDotDotDir->dir_fstClusLO = (lastClusRead & 0x0000FFFF);
    lseek(fatFd, (((x - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec + sizeof(dirEnt),
SEEK_SET);
    write(fatFd, newDotDotDir, sizeof(dirEnt));

    //Write new sub-dir end marker
    dirEnt* endSubDir = (dirEnt*) malloc(sizeof(dirEnt));
    endSubDir->dir_name[0] = 0;
    lseek(fatFd, (((x - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec + (2 *
sizeof(dirEnt)), SEEK_SET);
    write(fatFd, endSubDir, sizeof(dirEnt));

    if (!foundOpenDir)
    {
      //Write new endir marker
      dirEnt* endDir = (dirEnt*) malloc(sizeof(dirEnt));
      endDir->dir_name[0] = 0;
      lseek(fatFd, (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec +
offset, SEEK_SET);
      write(fatFd, endDir, sizeof(dirEnt));
      free(endDir);
    }

    //Cleanup
    free(dir);free(newDir);free(newDotDir);free(newDotDotDir);free(endSubDir);free(memsetBuffer);

    }
  return 1;
```

```
}

//Construct a directory entry to be written to the disk
dirEnt* buildNewDirEnt(std::string fileName, int type)
{
   dirEnt* newDir = (dirEnt*) malloc(sizeof(dirEnt));

   if (type == DIRECTORY)
   {
     newDir->dir_attr = 0x10;
     //Set new filename
     std::locale loc;
     for (int i = 0; i < 11; i++)
     {
          if (i < fileName.size())
            newDir->dir_name[i] = std::toupper(fileName[i],loc);
          else
            newDir->dir_name[i] = 32;
     }
   }
   else
   {
     newDir->dir_attr = 0x00;
     //Set new filename
     std::locale loc;
     size_t nextDelim = fileName.find_last_of('.');

     for (int i = 0; i < 8; i++)
     {
          if (i < fileName.size() && (i < nextDelim || nextDelim == std::string::npos))
            newDir->dir_name[i] = std::toupper(fileName[i],loc);
          else
            newDir->dir_name[i] = 32;
     }
     if (nextDelim != std::string::npos)
     {
          for (int i = 0; i < 3; i++)
          {
            if (nextDelim + i < fileName.size())
              newDir->dir_name[i+8] = std::toupper(fileName[nextDelim+i+1],loc);
            else
              newDir->dir_name[i+8] = 32;
          }
     }
     else
     {
          for (int i = 8; i < 11; i++)
          {
            newDir->dir_name[i] = 32;
          }
     }
   }
```

```
    newDir->dir_fileSize = 0;

    //Not supporting optional fields
    newDir->dir_crtTimeTenth = 0;
    newDir->dir_crtTime = 0;
    newDir->dir_crtDate = 0;
    newDir->dir_lstAccDate = 0;

    //Time stuff
    std::time_t ttime = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
    struct tm* cur_time = localtime(&ttime);


    //now need to add . and .. directories
    //std::cout << "returning dir " << strip(newDir->dir_name) << std::endl;
    return newDir;
}

//Remove a file at  the given destination
int OS_rm(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;
  dirEnt* dir = OS_readDir(pathSlice);
  if (dir)
  {
    int i = 0;
    bool foundDir = false;
    while (dir[i].dir_name[0] != 0x00)
    {
      std::string entryName = strip(dir[i].dir_name);
      //std::cout << strip(dir[i].dir_name) << " " << i << std::endl;
      //Found matching pre-existing directory
```

```
      if (stringCompare(entryName,fileName) == 1 && !(dir[i].dir_attr & 0x10))
      {
            //std::cout << "Located file to remove" << std::endl;
            foundDir = true;
            break;
      }
      i++;
    }
    if (!foundDir)
    {
      free(dir);
      //std::cout << "Coudn't locate desired dir" << std::endl;
      return -2;
    }

    //Check how many clusters long directory is
    int clusNum = i * sizeof(dirEnt) / clusSize;
    while (clusNum > 0)
    {
      lastClusRead = getFATEntry(lastClusRead);
      clusNum--;
    }

    //Free cluster chain
    unsigned int highWord = (unsigned int) dir[i].dir_fstClusHI << 16;
    unsigned int lowWord = (unsigned int) dir[i].dir_fstClusLO;
    int temp = highWord | lowWord;
    int next = getFATEntry(temp);
    setFATEntry(temp,0);
    while (next < eofVal)
    {
      temp = next;
      next = getFATEntry(next);
      setFATEntry(temp,0);
    }

    //write to new  directory
    dir[i].dir_name[0] = 0xE5;
    int offset = i * sizeof(dirEnt) % clusSize;
    lseek(fatFd, (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec + offset,
SEEK_SET);
    write(fatFd, &dir[i], sizeof(dirEnt));

    free(dir);
    //std::cout << "removed  file" << std::endl;

  } else return -1; // Path invalid
  return 1;
}

//Create a file at the given destination
int OS_creat(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;
dirEnt* dir = OS_readDir(pathSlice);
if (dir)
{
  int i = 0;
  bool foundOpenDir = false;
  while (dir[i].dir_name[0] != 0x00)
  {
    std::string entryName = strip(dir[i].dir_name);
    //std::cout << strip(dir[i].dir_name) << " " << i << std::endl;
    //Found matching pre-existing directory
    if (stringCompare(entryName,fileName) == 1)
    {
        //std::cout << "File already exists" << std::endl;
        free(dir);
        return -2;
    }

    if(dir[i].dir_name[0] == 0xE5)
    {
        foundOpenDir = true;
        break;
    }
    if (!foundOpenDir)
        i++;
  }

  //Check how many clusters long directory is
  int clusNum = i * sizeof(dirEnt) / clusSize;
  while (clusNum > 0)
  {
    lastClusRead = getFATEntry(lastClusRead);
```

```
      //std::cout << clusNum << " " << lastClusRead << std::endl;
      clusNum--;
    }

    //Build new dirent
    dirEnt* newDir = buildNewDirEnt(fileName, FILE_T);
    int x = findFreeFATEntry();
    setFATEntry(x, -1);
    newDir->dir_fstClusHI = (x & 0xFFFF0000) >> 16;
    newDir->dir_fstClusLO = (x & 0x0000FFFF);


    //write to new  directory
    int offset = i * sizeof(dirEnt) % clusSize;
    //std::cout << "overwriting " << i << " at offset " << offset << "for clus" << lastClusRead << std::endl;
    lseek(fatFd, (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec + offset,
SEEK_SET);
    write(fatFd, newDir, sizeof(dirEnt));

    //Clear new cluster
    char* memsetBuffer = (char*) malloc(clusSize);
    memset(memsetBuffer, 0, clusSize);
    lseek(fatFd, (((x - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec, SEEK_SET);
    //std::cout << "writing 0s to " << x  << std::endl;
    write(fatFd, memsetBuffer, clusSize);

    //Allocate additional cluster if dir full
    int dirsPerClus = clusSize / sizeof(dirEnt);
    //std::cout << i << " " << dirsPerClus << " " <<  i % dirsPerClus << std::endl;
    if ((i % dirsPerClus) + 2 > dirsPerClus)
    {
      if (fatType == 16 && lastClusRead == rootClusNum) //can't expand root cluster in fat 16
            return -1;
      int addClus = findFreeFATEntry();
      setFATEntry(lastClusRead, addClus);
      setFATEntry(addClus, -1);
      lastClusRead = addClus;
      offset = 0;
    } else
      offset += sizeof(dirEnt);

    if (!foundOpenDir)
    {
      //Write new endir marker
      dirEnt* endDir = (dirEnt*) malloc(sizeof(dirEnt));
      endDir->dir_name[0] = 0;
      lseek(fatFd, (((lastClusRead - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec +
offset, SEEK_SET);
      write(fatFd, endDir, sizeof(dirEnt));
      free(endDir);
    }

    //Cleanup
```

```
    free(dir);free(newDir);free(memsetBuffer);

    //std::cout << "Successfully created " << fileName << std::endl;
    } else return -1;
  return 1;
}

//Write to a file with the given filedescriptor
int OS_write(int fildes, const 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;

  dirEnt* thisDir = (dirEnt*) malloc(sizeof(dirEnt));
  lseek(fatFd, fileTable[fildes], SEEK_SET);
  read(fatFd, thisDir, sizeof(dirEnt));

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

  //Iterate clusters to correct part of file
  while (startClus > 0)
  {
    currentClus = getFATEntry(currentClus);
    if (currentClus >= eofVal)
      return -1;
    startClus--;
  }

  //std::cout << "filename: " << strip(thisDir->dir_name) << std::endl;
  //std::cout << "writing to file at " << currentClus << std::endl;

  thisDir->dir_fileSize = std::max((offset+nbyte), (int) thisDir->dir_fileSize);

  //Check to see if there is enough space in the cluster to write remaining bytes
  offset = offset % clusSize;
  lseek(fatFd, ((((currentClus - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec) + offset ,
SEEK_SET);
  if (clusSize - offset > bytesLeft)
  {
    write(fatFd, buf, nbyte);
    lseek(fatFd, fileTable[fildes], SEEK_SET);
```

```
  write(fatFd, thisDir, sizeof(dirEnt));
  free(thisDir);
  return nbyte;
 }
 else
 {
  write(fatFd, buf, clusSize - offset);
  bytesWritten = clusSize - offset;
  bytesLeft -= bytesWritten;
 }

 //Continue reading next cluster until all bytes read, or end of file reached
 while(bytesLeft > 0)
 {
  unsigned int temp = getFATEntry(currentClus);
  if (temp >= eofVal)
  {
        int newClus = findFreeFATEntry();
        setFATEntry(currentClus, newClus);
        setFATEntry(newClus, eofVal);

      //Clear new cluster
        char* memsetBuffer = (char*) malloc(clusSize);
        memset(memsetBuffer, 0, clusSize);
        lseek(fatFd, (((newClus - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec,
SEEK_SET);
        write(fatFd, memsetBuffer, clusSize);

        currentClus = newClus;
  } else currentClus = temp;

  lseek(fatFd, ((((currentClus - 2) * bB->bpb_secPerClus) + FirstDataSector) * bB->bpb_bytesPerSec),
SEEK_SET);
  if (bytesLeft >= clusSize)
  {
   write(fatFd, buf+bytesWritten, clusSize);
   bytesLeft -= clusSize;
   bytesWritten += clusSize;
  } else
  {
   write(fatFd, buf+bytesWritten, bytesLeft);
   bytesLeft -= bytesLeft;
   bytesWritten += bytesLeft;
  }
 }
 lseek(fatFd, fileTable[fildes], SEEK_SET);
 write(fatFd, thisDir, sizeof(dirEnt));
 free(thisDir);
 return bytesWritten;
}
```