PROJECT 6

Disks and File Systems

You are free to code your solutions using any C/C++ tools or operating system you choose, but be aware that your project will be graded using OS X 10.9 and/or Ubuntu 12.04. You are responsible for ensuring that your code compiles and runs correctly on these operating systems using standard APIs.

PART ONE

(200 points)

The purpose of this project is to simulate a file system on a disk. You will be simulating the FAT12 file system using a 3.5" floppy disk image as the storage media, much like you did in Project 2. The specifications for a standard double-sided, high-density, 3.5-inch floppy disk drive are: 80 tracks per side, 18 sectors per track, 512 bytes per sector for a total of 1,474,560 bytes or 1.44 MB.

DESIGN CONSTRAINTS

1. **Disk Structure**. The structure of a floppy disk consists of four major sections: the boot sector, FAT tables, the root directory, and the data area.

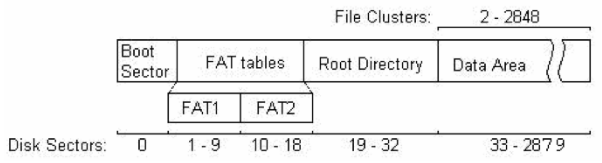

Figure 1: Disk organization of the FAT12 file system[1]

2. **Startup**. When your program starts up, you will initialize the array in memory that simulates your floppy disk with the contents of the 3.5" 1.44MB floppy disk image that you created and turned in for Project 2. You will name this file **`fdd.flp`**.
3. **Boot Record**. The first sector (sector 0) will be the master boot record, as described in Project 1. Recall that it is 512 bytes, the first 446 bytes are the boot code, the next 64 bytes contain partition information, and the MBR ends in the two-byte signature 0x55AA. You never need to write to this area once it is created/loaded into memory.
4. **FAT Tables**. The FAT tables (there are two identical copies for backup/redundancy) follow the boot sector.
   a. The FAT is a simple data structure that maps the data sectors of the disk. It is similar to an array, and each entry in the FAT corresponds to a sector of data on the disk. FAT tables contain pointers to every sector on disk, indicate the number of the next sector in the current chain, the end of the chain, whether a sector is empty or has errors. The FAT tables are the only method of finding the location of files and directories on the rest of the disk. There are typically two redundant copies of the FAT table on disk for data security and recovery purposes.
   b. FAT entries are comprised of 12-bits. This is because 12 bits are needed to address all the sectors on a 1.44MB floppy (2812 sectors requires 12 bits: $2^{11} < 2812 < 2^{12}$).
   c. FAT table entries signify the following:
      0x00 – unused
      0xFF0-0xFF6 – reserved sector.
      0xFF7 – bad sector.
      0xFF8-0xFFF – last sector in a file.
      (anything else) – number of the next sector in the file.
   d. The FAT numbers sectors using a "logical" data sector value that starts at index 0 in the FAT table, but this does NOT correspond to the number of the first usable sector on the physical disk. To determine the physical sector that corresponds to a logical sector, two items must be taken into account:
      i. Recall that the first 33 sectors of the disk are reserved. Therefore, the first sector of user data is at sector "33", since we count starting at sector 0.
      ii. Entries 0 and 1 of the FAT are reserved. Entry 2 of the FAT actually contains the description for physical sector 33. Therefore:
         `physical sector number = 33 + FAT entry number – 2`.
      so entry 5 of the FAT corresponds to the physical sector on the disk numbered 36.

[1] Source: Brigham Young University.

e. Since computers like to do things in 8-bit bytes, a 12-bit FAT entry is annoying. This requires "packing" two FAT entries into every three bytes if you write it to disk. For the purposes of this assignment, you can start with a simple implementation by simulating the FATs in memory as 2-byte entries (i.e. unsigned shorts) in an array. However, for the maximum credit you may attempt implementing the 12-bit, packed FAT entries correctly in memory. This is of course also necessary should you attempt writing to and from the disk image (see below).

5. **Directories**. Directories (such as the root directory) exist like files on the disk, in that they occupy one or more sectors. Each sector (512 bytes) of a directory contains 16 directory entries (each of which is 32 bytes long). Each directory entry describes and points to some file or subdirectory on the disk. Thus, the collection of directory entries for a directory specify the files and subdirectories of that directory.
   a. Each directory entry is 32 bytes and contains the following information about the file or subdirectory to which it points. The structure of a directory entry is as follows:

Byte	**0-7**	**8-10**	**11**	**12-13**	**14-15**	**16-17**	**18-19**	**20-21**	**22-23**	**24-25**	**26-27**	**28-31**
Purpose	Filename (8 captial chars, pad with spaces)	Extension (3 capital chars)	Attrib -utes	Reser -ved	Create time	Create date	Last access date	Ignore	Last write time	Last write date	First logical sector	File size (bytes)
Notes: If the first byte of the filename is 0xE5, then the directory entry is "empty" or "free". If the first byte is 0x00, then this directory entry and all entries that follow are free.												

   b. Realize that the directory entry specifies where the file or subdirectory starts (first sector) and how long it is (file size), but files are NOT stored sequentially. In fact, parts of files can be scattered all over the disk. You have to go to the FAT to find the entry for the first sector and then follow the links to get the rest of the sectors containing data for the file.
   c. The root directory is the primary directory of the disk. Unlike other directories, the root directory has a finite size (14 sectors, 16 directory entries per sector).
   d. You do NOT have to support subdirectories in this project, only the root directory.
   e. In reality, times and dates in a directory entry are stored in 2-byte values with a non-trivial encoding. For this project, you may implement a simple time and date encoding as follows:
      i. Time values: hour (first byte, in 24-hour format) and minute (second byte).
      ii. Date values: month (first byte) and year (second byte), all assumed to be in year 2013.

6. **Required Operations**. Your project must support the following operations:
   a. **List Directory**. You must be able to list the directory of the disk. Try to make this look like the MS-DOS directory listing that you emulated in Project 1 (see sample output below).
   b. **Copy**. You must be able to copy a file from the local project directory onto the simulated disk. Assume that files to be copied are in the same directory as your source code. For example, if I had a text file that contained the Gettysburg Address (it's the 150th anniversary, so you should read it by the way!), then I would want that file copied to the disk image, the appropriate directory entry made, and the correct FAT links created.
      i. You need to ensure that you don't overwrite any used sectors of the disk, thus corrupting other files.
      ii. The algorithm that you use to figure out where to put the file is up to you (First-fit, best-fit, worst-fit, for example), but you should endeavor to fit a file in a continuous sequential series of sectors to reduce fragmentation.
      iii. If file fragmentation cannot be avoided, then try to fit files in the fewest number of fragments.
      iv. You do NOT have to worry about "compaction" or "defragmenting."
      v. You DO need to worry about if a file cannot fit on the disk.
   c. **Delete**. You must be able to delete a file from the disk. This is actually very simple, and you don't need to erase or zero out the sectors assigned to the file or even the FAT entries for it. (Think about why not!)
   d. **Rename**. Renaming a file on the disk.
   e. **Disk Usage Map**. Your program must print a map of all sectors of the disk, indicating which are free and which are in use.
   f. **Directory Dump**. Your program must print out the HEX (and ASCII) representation of the root directory structure on demand.
   g. **FAT Dump**. Your program must print out the HEX representation of the FATs on demand.
   h. **FAT Chain**. Your program must print - in order - the physical sectors allocated to any file the user chooses.
   i. **Sector Dump**. Your program must print out the HEX and ASCII representation of any sector on demand.

7. **Write to Disk Image**. For full credit, you must also write all changes to the simulated disk in memory to and from the actual `fdd.flp` image file as they occur.

HINTS:

1. It is HIGHLY SUGGESTED that you utilize C-style structs to exactly define the structure of FAT entries, directory entries, etc. This will allow you to store and write entire entries as a single variable of a perfectly sized and precisely organized data type that matches the physical layout of individual bytes expected in memory and/or on disk.

SAMPLE OUTPUT

```
MENU:
1) List directory
2) Copy file to disk
3) Delete file
4) Rename a file
5) Usage map
6) Directory dump
7) FAT dump
8) FAT chain
9) Sector Dump
> 1

Volume Serial Number is 0859-1A04
Directory of C:\

IO       SYS     13454 11-11-91   5:00a
GETTYSBU TXT      1287 11-19-13   2:15p
WHALE    TXT   1193405 11-17-13   3:33p
        3 file(s)    1208146 bytes
                      231424 bytes free
(note that free bytes isn't simple subtraction, as bytes can be lost due to internal fragmentation because of sectors)

MENU: (menu omitted here to save space)
> 6

ROOT DIRECTORY:
|-----FILENAME-----|-EXTN-|AT|RESV|CRTM|CRDT|LADT|IGNR|LWTM|LWDT|FRST|--SIZE--|
494f 2020 2020 2020 5359 5327 0000 0000 0000 0000 0000 621b 1d00 169f 0000 348e IO       SYS
4745 5454 5953 4255 5458 5400 0000 0d0e 0b14 0b14 0000 0d0e 0b14 0000 0000 0507 GETTYSBU TXT
5748 414c 4520 2020 5458 5400 0000 0f21 0b12 0b13 0000 0f21 0b13 0003 0012 35BD WHALE    TXT
... continues with more entries.

MENU: (menu omitted here to save space)
> 5

CAPACITY: 1,474,560b   USED: 346,452b (xx.x%)   FREE: 1,128,108 (xx.x%)
SECTORS: 2,880         USED: 560 (xx.x%)        FREE: 2,320 (xx.x%)
FILES: 46      SECTORS/FILE: 12.17     LARGEST: 72s     SMALLEST: 1s

DISK USAGE BY SECTOR: (sample data for illustration purposes only)
           |----+----|----+----|----+----|----+----|----+----|----+----|----+----|----+----
0000-0079: BFFFFFFFFFFFFFFFFFFRRRRRRRRRRRRRRXXX...XXXXXX...................................
0080-0159: ...X...XXXXX............XXXXXXXXXXXXXXX.....................XXXXX.........XXXXXX
0160-0239: XXXXXXXXXXXX...XXXXXXX..........................................................
0240-0319: ......................XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.........
0320-0399: ................................................................................
 (rows 05 to 34 here)
2800-2879: ................................................................................

MENU: (menu omitted here to save space)
> 7 (note that this is really big and a lot of it will be zeros)
PRIMARY FAT TABLE:
0000-0019: 000 000 002 003 fff 000 000 000 008 009 010 011 012 013 fff 000 000 000 000 000
0020-0039: 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0040-0059: 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
... (continues on and on here)
2800-2879: 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000

SECONDARY FAT TABLE CONSISTENCY CHECK:
The secondary FAT table DOES/DOES NOT match the primary FAT table.

MENU: (menu omitted here to save space)
> 9 (note that this dumps physical sectors, not logical ones)
Select physical sector to display: 33
000: 46 6F 75 72 20 73 63 6F 72 65 20 61 6E 64 20 73 65 76 65 6E Four score and seven
020: 20 79 65 61 72 73 20 61 67 6F 20 6F 75 72 20 66 61 74 68 65  years ago our fathe
040: 72 73 20 62 72 6F 75 67 68 74 20 66 6F 72 74 68 2C 20 75 70 rs brought forth, up
(Gettysburg address continues until the 512 byte sector is fully printed.)
500: 69 6E 20 61 20 6C 61 72 67 65 72 20                         in a larger
```

```
MENU: (menu omitted here to save space)
> 8
Filename for which to list allocated sectors: GETTYSBU.TXT
Logical:  000 001 002 003
Physical: 033 034 035 036

MENU: (menu omitted here to save space)
> 2
Filename to copy to the simulated disk: SMALL.TXT (simulator then copies the file to the disk)

MENU: (menu omitted here to save space)
> 3
Filename to delete: SMALL.TXT

MENU: (menu omitted here to save space)
> 4
Filename to rename: GETTYSBU.TXT
New name: LINCOLN.TXT
```

Grading

Ensure that the file that contains your main method is called **partone.cpp**. If you have a large number of classes or files, a **makefile** is highly recommended. Remember, a happy grader leads to a happy grade! Points will be generally awarded for each component of the project you complete, with some points being reserved for correct compilation, nice output, etc.

10 points	Correctly simulate the raw bytes of an FDD in memory and reading the image into that memory at start.
25 points	Correctly simulate writing files to and from the simulated disk image in memory.
10 points	Correctly print out the usage map of the disk.
20 points	Correctly simulate the two FATs in memory. *(can be "unpacked" 16-bit entries for basic credit)*
15 points	Correctly simulate a "packed" FAT of 12-bit entries in memory. *(points in addition to the basic credit above)*
10 points	Correctly print out the FAT map.
20 points	Correctly simulate the root directory entries and update them as needed.
10 points	Correctly list the directory of the disk (in user readable format, e.g. MS-DOS's DIR command).
10 points	Correctly dump the raw contents of the root directory structure.
10 points	Correctly simulate deleting files from the disk and updating directory entries as needed.
10 points	Correctly dump the contents of any sector that is requested.
40 points	Correctly write to and from the physical disk image (stored in the file `fdd.flp`).
10 points	Correct compilation, good output, free of errors, good commenting, and otherwise nice work.

Bonus:

1. The MBR contains four 16-byte partition records at the end. For a bonus, create a partition signature for the disk, following the standards set forth in the FAT12 specification (Google it!). The 16-byte partition entry is comprised of a boot flag, a Begin code, a type code, an End code, the start location of the partition, and number of sectors in it.
2. The attributes attribute (not a typo!) of a directory entry contain information about files, such as read-only status, hidden status, system status, etc. (Google it!) Implement the attributes attribute for a bonus.
3. Implement the correct date and time encoding specified in the FAT12 standard for directory entries and adjust the printout of the directory to accurately reflect it. (Google it!) The encoding is roughly as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Day of month (1-31)					Month of year (1-12)						Years from 1980 (0-127)					0000h
Note that the range of valid years is only from 1980 until 2107.																

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Hours (0-23)					Minutes (0-59)						Seconds (0-29)					0000h
Note that seconds are counted with a 2 second interval, so a value of 29 in this field gives 58 seconds.																

PROJECT SUBMISSION

1. ALL SOURCE CODE YOU TURN IN MUST CONTAIN THE FOLLOWING AT THE TOP:
```
    // CS3242 Operating Systems
    // Fall 2013
    // Project 6: Disks and File Systems
    // John S. Doe and Bob A. Smith
    // Date: 9/23/2013
    // File: partone.cpp
```
2. Zip ALL source code files for your project into a single ZIP file named "DOE_SMITH.ZIP" (where Doe and Smith are the surnames of the two students) and upload that as your submission in Dropbox on D2L by the posted deadline.