CprE 308 Laboratory 7 & 8: The FAT-12 Filesystem

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

Submit the following items on Canvas:

- A brief summary of what you learned during this lab, no more than two paragraphs **20 pts**
- Your source code along with a makefile or other compilation scripts (160 pts)

This lab assignment will have total of 200 points since it combines two lab assignments. Attendance is worth 20 points.

Due Date: Two weeks after your lab session, in the week of December 2-6.

2 Introduction

In this lab, you will gain hands-on experience reading a FAT-12 filesystem image. In Part I, you will learn to decode the boot sector by hand, and then write a program to do this automatically. In Part II, you will write a program to perform the `ls` function with a system image.

2.1 FAT-12 Filesystem Layout

Boot Sector	FAT	2nd FAT	Root Directory	Data
0	1	X	Y	Z

FAT-12 filesystems have logical sectors starting at 0. This is where you'll find the boot sector. The first FAT always starts at logical sector 1, but after that you have to calculate the starting positions using information from the boot sector. The formulas for finding the sector numbers for the 2nd FAT, the root directory, and the data area are as follows:
$X = 1 + sectors/FAT$
$Y = X + sectors/FAT$
$Z = Y + roundup512(32 \cdot directoryEntries)$ /* have to start at the next open block */
Note: roundup512() rounds up to the next multiple of 512; this is because data must start at an open block The last formula is because each directory entry takes 32 bytes of space.

2.2 Little Endian numbers

Before we proceed with the lab assignment, let's first understand "Little Endian numbers". Numbers in x86-based systems are stored in the little endian format. This means that the most significant byte comes last. For example, if the bytes "`34 12`" were stored at offset `0xF0` in the file, then we would know that `0x34` is stored at `0xF0`, and `0x12` is stored at `0xF1`. If we interpret this as a "short" value (2 bytes) starting at address `0xF0`, then the value appears to be `0x3412`, but it represents `0x1234`. Similarly, if we were storing an int (4 bytes), the bytes would be reverse order, from least to most significant byte.

Suppose that you had an array of short values in memory. What would they look like? The array is stored forward in memory, so we would reverse the first two bytes to get the first short. We would reverse the next two bytes to get the second short, and so on. What about a character array? Because a character is only one byte, there is no reversal. An array of ASCII characters should not be reversed.

3 Part I: Decoding the Boot Sector

3.1 Boot Sector Layout

A **Sector** is the smallest unit of transfer; It's also called a block. There are 512 bytes per sector on a floppy disk. The **Boot sector** stores vital information about the filesystem. The boot sector is laid out in the following way, starting at the beginning of the disk (logical sector 0, byte 0):

All values are stored as **unsigned little-endian** numbers unless otherwise specified.

Table 1: The Layout of FAT-12 Boot Sector

Offset	Length	Contents	Display Format
0x00	3	Binary offset of boot loader	hex
0x03	8	Volume Label (ASCII, null padded)	ASCII
0x0B	2	Bytes per sector	decimal
0x0D	1	Sectors per cluster	decimal
0x0E	2	Reserved sectors	decimal
0x10	1	Number of FATs (generally 2)	decimal
0x11	2	Max Number of Root Directory entries	decimal
0x13	2	Number of logical sectors	decimal
0x15	1	Medium descriptor	hex
0x16	2	Sectors per FAT	decimal
0x18	2	Sectors per track	decimal
0x1A	2	Number of heads	decimal
0x1C	2	Number of hidden sectors	decimal

3.2 Exercise 1

Download files `image` and `bytedump.c` from Canvas. The file "image" is a FAT-12 system image, and "bytedump.c" is a program to read binary values from the image. Table 1 has shown us how the bits are laid out in the FAT-12 boot sector. You are going to decode a sample system image and find out the information stored in its boot sector. To help get you started, we're going to decode a few fields manually using the bytedump program.

Compile "bytedump.c". Given a filename and an offset (in decimal notation), the program will print out 32 bytes of hex, decimal and ASCII values. For example, `./bytedump image 1536` will produce the following output with the given image:

```
$./bytedump image 1536
Trying to read 32 bytes in "image" starting from offset 1536.
Actually read 32 bytes:
================================
Address  Hex    Decimal  ASCII
0x0600   0x31   49       1
0x0601   0x36   54       6
0x0602   0x53   83       S
0x0603   0x45   69       E
0x0604   0x43   67       C
0x0605   0x20   32
...
...
0x061e   0x00   0
0x061f   0x00   0
```

The first column is the address (in hexadecimal), and the second is the data at that address (also in hexadecimal). The third column is the decimal value of the data, while the fourth column is the ASCII value of the data.

Using the bytedump program and the offsets in Table 1, find and decode the values for each of the following fields in the boot sector (remember that it starts at offset 0):

Remember that data is stored in the little endian format.

	Hex	Decimal
Bytes per sector		
Sectors per cluster		
Max Number of Root directory entries		
Sectors per FAT		

NOTE: No submission is needed for this exercise. Have the lab instructor check your answers. You will use these values for debugging the next part.

3.3 Exercise 2

Download "bsdump-template.c" and fill in the missing code marked by comment "TODO". The completed program should be able to read and then print information from the boot sector of the image. The starting offset and size of each field can be found in Table 1.

An example output of the completed program would be:

```
$./bsdump image

                    Name:   mkdosfs
            Bytes/Sector:   512
         Sectors/Cluster:   16
        Reserved Sectors:   1
          Number of FATs:   2
  Root Directory entries:   224
         Logical sectors:   2880
       Medium descriptor:   0x00f0
             Sectors/FAT:   1
           Sectors/Track:   18
         Number of heads:   2
Number of Hidden Sectors:   0
```

4 Part II: Decoding the Root Directory

In this Part of lab session, you will write a program to decode the root directory portion of the filesystem image, and print out the information of all the valid root directory entries.

4.1 Root Directory Layout

The root directory is an array of directory entries (the number of such entries is given in the boot sector). Each directory entry is 32 bytes and is laid out according to the table below:

Offset	Length	Description
0x00	8	Filename
0x08	3	Extension
0x0b	1	Attributes
0x0c	10	Reserved
0x16	2	Last Access Time
0x18	2	Last Access Date
0x1a	2	First Cluster
0x1c	4	Size

Table 2: Directory Entry

Notes on interpreting a directory entry:

If the filename starts with 0x00, it's not a valid entry. And all the entries following this entries are invalid. If it starts with `0xE5`, it's also an invalid entry, meaning it has been released/deleted. If it starts with anything else, assume that it's a valid file entry.

Filename in DOS, the naming convention is 8.3. That is, files have 8 character names and 3 character extensions, such as `FILENAME.EXT`. We will not cover the long filename extensions in this lab.

Attributes a file's attributes include read/write permissions, among other things. There are 8 bits, but we're only interested in a few of them for this lab. If a bit is set, it indicates the property is true. 0 is the least significant (rightmost) bit.

Bit	Attribute
0	Read-only
1	Hidden
2	System
5	Archive

Date and time These fields are each 16 bits long. The bits are laid out as follows:

The time is encoded using 5 bits for the hour, 6 bits for the minutes, and 5 bits for the seconds. One catch - Since 5 bits is not enough for 60 seconds, only every other second is counted. In other words, if your counter is 16, that indicates a value of 32 seconds.

The date is encoded in a yyyy/mm/dd format. It uses 7 bits for the years since 1980, so add 1980 to the counter to get the current year. There are 4 bits for the month, and 5 bits for the day.

4.2 Exercise 3

Download `fat12ls-template.c` from Canvas, and fill in the missing code marked with "TODO". The completed program should do what `ls` would do: print out the filename, attributes, last access time, last access date, and size of the file for each valid entry in the root directory. No need to sort the files – simply list them in disk order.

For function **decodeBootSector**, you can reuse the code from Exercise 2.

Here is an output example of the completed program:

```
File_Name    Attr    Time        Date          Size
16SEC.TXT    RHS     15:22:50    2002/11/06    331
...
...
BIG.LOG      HS      00:00:58    2009/05/03    62559
(R)ead Only (H)idden (S)ystem (A)rchive
```