Background

To complete this assignment, you need to know how directories and files are stored on a 1.44 Mb diskette, formatted for DOS/Windows. Although the basic idea is fairly simple, there are a number of small details. By sorting out these details, you will obtain a greater appreciation of how file systems can be stored on disks.

The File Allocation Table

We begin our investigation by considering the layout of such a diskette. As we know, the contents of a diskette can be viewed as a sequence of 2,880 512-byte sectors. These sectors are indexed from 0 through 2,879 and store the boot sector, a file allocation table (FAT), a second copy of the FAT, the root directory, and all of the data sectors that constitute subdirectories and files on the diskette, as shown in Figure 1.

0	1	10	19	33 2,879
B	FAT #1	FAT #2	root directory	data

Figure 1: Disk layout for a 1.44 Mb DOS diskette. B is the boot sector.

Conceptually, the FAT is an array of 12-bit quantities. In other words, each entry takes up 1.5 bytes, or 3 hexadecimal digits. Since processors work most easily with bytes, words, and double words, the designers had to make a choice: use two bytes per entry, wasting four bits of space for each and every entry, or pack two entries into three bytes, making use of each bit. Living in an era of relatively small disks and memories, the latter scheme was chosen. To interpret the FAT area of the disk, some unpacking and rearrangement of bits is thus necessary.

Suppose the bytes of the FAT area are

$$B_0, B_1, B_2, \ldots, B_{3k}, B_{3k+1}, B_{3k+2}, \ldots$$

Each byte consists of two hexadecimal digits: the most significant four bits H, and the least significant four bits L. We can think of the FAT area as a sequence of hexadecimal digits—in groups of three bytes—as follows:

$$H_0L_0, H_1L_1, H_2L_2, \ldots, H_{3k}L_{3k}, H_{3k+1}L_{3k+1}, H_{3k+2}L_{3k+2}, \ldots$$

The three bytes $H_{3k}L_{3k}, H_{3k+1}L_{3k+1}, H_{3k+2}L_{3k+2}$ represent entries $2k$ and $2k+1$ in the FAT. These entries are:

$$\text{FAT}_{2k} = L_{3k+1}H_{3k}L_{3k} \text{ and } \text{FAT}_{2k+1} = H_{3k+2}L_{3k+2}H_{3k+1}.$$

As a further detail, the first three bytes of the FAT are reserved: we can safely ignore them.

Consider an example. Suppose the first few bytes of a FAT are as follows:

```
F0 FF FF 03 40 00 FF 6F 00 FF 0F 00 00 00 00
```

Viewed as groups of three, we get

```
F0 FF FF
03 40 00
FF 6F 00
FF 0F 00
00 00 00
```

Ignore the first three bytes. Unpacking and rearranging the remaining hexadecimal digits gives:

```
003   004
FFF   006
FFF   000
000   000
```

Arranged in an array, the first few values are:

0	1	2	3	4	5	6	7	8	9	···
—	—	003	004	FFF	006	FFF	000	000	000	···

The values stored in this table should be viewed as unsigned index (or pointer) values, where `0xFFF` is interpreted as a null pointer. In fact, any value between `0xFF8` and `0xFFF` can be used as a null pointer. In this table, there are two chains: $2 \to 3 \to 4$ and a second, $5 \to 6$.

There is another way to view bit packing which is sometimes helpful. Suppose we wish to find the jth 12-bit entry of the file allocation table. To do so, join bytes $k+1$ and k (in that order), where $k = \lfloor \frac{j}{2} \rfloor + j$. If j is even, the four most-significant bits of this 16-bit result should be removed; if j is odd, the four least-significant bits are removed.

Directories and Subdirectories

Each directory entry occupies 32 contiguous bytes of storage, as shown in Figure 2. Since each sector is 512 bytes, 16 directory entries can be placed in each sector. All directory entries at the root level must appear in the portion of the disk reserved for this purpose, sectors 19 through 32 (See Figure 1).

0	8	11	12	22	24	26	28 31
name	ext	attr		time	date	start sector	size

Figure 2: Layout for a directory entry. "Name" is an 8-character file name and "ext" is the 3-character file name extension. "Attr" provides various attributes of the file; "time" and "date" reflect when the file was created. "Size" is a 4-byte quantity which indicates the number of bytes in the file. "Start sector" indicates where the first sector of the file contents appears. The shaded region is unused and is reserved for a future use.

The attribute field can be further subdivided, as shown in Figure 3. There are six bits which describe various things about the directory entry: if the bit is 1, the file has the specified property, if it is 0 it doesn't have this property. For example, a file with its archive and read-only bits set would have an attribute bit pattern of `0x21`.

7 6	5	4	3	2	1	0
	A	D	V	S	H	R

Figure 3: Attribute bits within a directory entry. A=archive, D=directory, V=volume label, S=system, H=hidden, R=read-only. The most significant two bits in the shaded region are unused.

Other important facts about directory entries include the following:

- File names are stored with upper-case ASCII letters.
- File names less than eight characters are padded with blanks on the right to fill the eight-character field.
- The first character of the name can have a special value:
  - `0x00`: the entry does not refer to a file or subdirectory.
  - `0xE5`: the entry was once a file or subdirectory, but has been deleted.
- The time field encodes hour (H), minute (M), and seconds (S) as an unsigned 2-byte quantity, using the formula $2048H + 32M + S/2$.
- The date field encodes month (m), day (d), and year (y) as an unsigned 2-byte quantity, using the formula $512(y - 1980) + 32m + d$.
- The start sector refers to the location of the first sector of the file. All subsequent sectors are determined from the chain of sectors stored in the FAT.
- All multibyte quantities use the little-endian convention and must be reversed before interpretation. For example, the date field `0x72 0x2d` is actually `0x2d72`. Similarly, the size field `0x10 0x23 0x45 0x00` is actually `0x00452310`.

An example of a directory entry, corresponding to a file named **LAZY1.TXT** is as follows:

```
4c 41 5a 59 31 20 20 20 54 58 54 20 00 5e b2 44
72 2d 72 2d 00 00 d9 41 72 2d 40 03 2e 00 00 00
```

Exercise 1. Copy or move `floppy-diskette` to a convenient place for your work in this current assignment. Using the **sd** script from the previous assignment, display the contents of the first sector of the FAT region of this disk. Based on the values of the sector dump, fill in the missing entries below with 12-bit entries (i.e., three hexadecimal digits) taking care to properly unpack the bits.

0	1	2	3	4	5	6	7	8	9	···
▬	▬									···

Make a copy of this table in a file named `README`.

Exercise 2. A program, named `fatdump`, is desired which will assist with the inspection of the FAT stored on the diskette image `floppy-image`. This program expects two command-line arguments, representing a range of index values. For example,

```
./fatdump 10 12
```

is a request to display entries 10 through 12 of the FAT. Your program should output the specified range of values in decimal. Typical results from `fatdump` might appear as follows:

```
FAT[10] = 4095
FAT[11] = 12
FAT[12] = 4095
```

Your program should verify the requested interval is in the appropriate range for a diskette; if inappropriate, `fatdump` should respond with an error message.

The `filesys.h` header file provides several constants and function prototypes which should be used for the development of `fatdump`. To complete this exercise, you need to create `filesys.c` and `fatdump.c`. Here are some things to keep in mind:

- `filesys.c` will contain the implementation of the function prototypes in `filesys.h`.
- `filesys.c` should declare an array of `char` just large enough to hold all of the bytes of the FAT. To determine the size, use an appropriate expression consisting of constants declared in `filesys.h`.
- To build the `fatdump` executable, you will need `filesys.o`, `fatdump.o` and `diskio.o`. Modify the `Makefile` from the previous assignment, adding a target for `fatdump`.

Exercise 3. Somewhere in the root directory of the floppy disk image, there is a file named `LAZY8.TXT`. Using `sd`, find this entry. What are the values of the corresponding 32 bytes? Show how to interpret these bytes. In particular, determine the following:

- All attribute values
- Time and date stamps
- Size, in bytes
- Chain of sector addresses

Put your answers — and supporting justification — in your `README` file. (Other than the sector addresses, your interpretations should make sense to an end user of this file system. Don't simply give the bytes as hexadecimal values.)

What to Submit

Before you submit your work, verify that your program compiles and works properly in the Mac OS X environment. Your `Makefile` should, at a minimum, support the following targets:

```
make clean
make fatdump
```

Put copies of your work — all source code, your `Makefile`, a copy of `floppy-diskette` and your `README` in a folder, then drag this folder onto the EIU submit icon. Be sure that your name appears in every file you create.

Source Code

```
1  /*
2    File: filesys.h
3
4    Header file for FAT-12 based files systems, using 1.44 Mb diskettes
5    MAT 4970
6
7  */
8
9  #ifndef FILESYS_H
10 #define FILESYS_H
11
12 /* Number of FAT sectors */
13 #define FAT_SECTORS 9
14
15 /* Range for first FAT copy */
16 #define LOW_FAT_SECTOR 1
17 #define HIGH_FAT_SECTOR 9
18
19 /* Range for second FAT copy */
20 #define LOW_AUX_FAT_SECTOR 10
21 #define HIGH_AUX_FAT_SECTOR 18
22
23 /* Range for root directory */
24 #define LOW_ROOT_SECTOR 19
25 #define HIGH_ROOT_SECTOR 32
26
27 /* Directory entry information */
28 #define DIRECTORY_ENTRIES_PER_SECTOR 16
29 #define BYTES_PER_ENTRY 32
30
31 /* Read the on-disk FAT into an in-memory copy */
32 int ReadFat();
33
34 /* Get a 12-bit entry of the FAT */
35 int GetFatEntry(int j);
36
37 /* Display a portion of the FAT */
38 void DisplayFat(unsigned int low, unsigned int high);
39
40 #endif
```