Background

Computer users think of the files on a computer system arranged in a tree-like arrangement of directories and files. Another, lower-level, view considers the file system to be a sequential arrangement of bytes stored on some permanent media. Various operating systems use different schemes for storing this data. In this exercise, we will look at some of the details of FAT-12, the method first used in Microsoft's DOS. Although FAT-12 was a victim of hardware improvements, similar systems are still in wide use today on USB memory devices, memories of digital cameras, MP3 players, etc.

It is convenient to think of the contents of a diskette as a sequence of fixed-length "blocks" or "sectors." The first of these, commonly known as the "boot sector," stores a variety of facts about the file system. The table below shows the location and meaning of some of the bytes in the boot sector.

Offset	**Length**	**Meaning**
0x03	8	Identifier
0x0b	2	BytesPerSector
0x0d	1	SectorsPerCluster
0x10	1	NumberOfFATs
0x11	2	MaxRootEntries
0x13	2	NumberOfSectors
0x15	1	MediaDescriptor
0x16	2	SectorsPerFAT
0x18	2	SectorsPerHead
0x1a	2	HeadsPerCylinder

All offsets are given as hexadecimal quantities and the length specifies the number of bytes. Within a sector, offsets are numbered starting with 0. It is important to note that two-byte quantities are stored in "little-endian" fashion: to interpret them as unsigned integer quantities, it is necessary to first swap the order of the bytes.

The FAT system is described in great detail in a "hardware white paper" that Microsoft makes available—see the course website. It serves as a useful reference and is ultimately the final authority on how the file system is organized.

Exercise 1. The file `floppy-image` is an exact byte-by-byte copy of a $3\frac{1}{2}$ inch high-density diskette. Using the `ls` command with an appropriate selection of command-line switches, determine the exact number of bytes and blocks in this file. (In this context, a block is 512 bytes.) If there are m blocks and n bytes, verify that $512m = n$.

What command did you use? Place this command and the values of m and n in a text file named `README`.

Exercise 2. The first block of this file has a special significance — it is known as the **boot sector**. To inspect the boot sector of an arbitrary file, we can use the **hexdump** command:

```
hexdump -v -C -n 512 -s 0b floppy-image
```

Try this out to see the type of output it produces. Explain what each of the command-line switches above specifies. Include your answers in your `README` file.

Exercise 3. Using the information in the boot sector of the provided diskette image, complete the table below. With the exception of the 8-byte identifier and the 1-byte media descriptor, all values should be given as unsigned, decimal integers. You will need to convert a few hexadecimal values into decimal[1].

Offset	**Length**	**Meaning**	**Value**
0x03	8	Identifier	
0x0b	2	BytesPerSector	$B =$
0x0d	1	SectorsPerCluster	
0x10	1	NumberOfFATs	
0x11	2	MaxRootEntries	
0x13	2	NumberOfSectors	$CHS =$
0x15	1	MediaDescriptor	0xf0
0x16	2	SectorsPerFAT	
0x18	2	SectorsPerHead	$S =$
0x1a	2	HeadsPerCylinder	$H =$

Some of the information in the boot sector refers to the *geometry* of the disk. All data on the disk is stored in a collection of *sectors*. Each sector on the disk is uniquely identified by an ordered triple (c, h, s)—c indicates the cylinder number, h denotes the head, and s is the sector number. If there are C cylinders, H heads, and S sectors per head, then the total number of sectors on the disk is the product CHS. If each sector stores B bytes, the total capacity is $CHSB$ bytes. When you have completed the table, determine the following:

- the total capacity, in bytes
- the number of cylinders

Put the contents of your completed table and the two calculated values in your `README` file.

Exercise 4. Implement a shell script, named `sd` (sector dump), which will display the contents of a specified sector of a diskette. The heart of this script will use `hexdump` to display the desired sector on the standard output device.

The `sd` script takes one argument, the name of the disk image, and one optional switch, the sector number. For example, to display the 13th sector of the image stored in `floppy-image`, we would enter the command:

```
sd -13 floppy-image
```

Omitting the optional switch is equivalent to asking for sector 0, the boot sector.

Your shell script should do thorough "sanity" tests. At a minimum, your script should test for the correct number of command-line arguments and switches, incorrect file name, and non-numeric and/or out-of-range sector numbers.

[1] Don't forget: use the little-endian convention.

Exercise 5. Suppose we wish to implement the script `sd` as a C program. The files `sdump.c`, `diskio.h`, and `diskio.c` provide a partial solution. As a first step, review this code to see how it is constructed. The `DisketteWrite()` function is not needed, but is nonetheless included for completeness. The provided `Makefile` has a target named `sdump`; as a result, the command

```
make sdump
```

will compile the appropriate files, producing an executable named `sdump`. As the code now stands, there are three deficiencies:

- No command-line arguments are used; sector 0 is always produced and the image file is assumed to be `floppy-image`.
- No sector offsets are displayed.
- No ASCII characters appear.

Make the appropriate modifications to the provided C code to remove these shortcomings. Upon completion, your extended `sdump` program will have the functionality of the `sd` shell script.

Exercise 6. In Exercise 3, we manually inspected various fields within the boot sector. Now do the same thing by writing a C program, displaying the contents of the following fields:

SectorsPerCluster
NumberOfSectors
SectorsPerHead
HeadsPerCylinder

Modify the `Makefile`, adding a target named `bdump` which will produce an executable named `bdump`.

What to Submit

Before you submit your work, verify that your scripts and programs work properly in the Mac OS X environment. In particular, ensure that your work compiles without errors or warnings:

```
make clean
make sdump
make bdump
```

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 each program as a comment somewhere near the top of the file.

Source Code

```
#
# Makefile for Homework 10
#
# MAT 4970
# Bill Slough
#
# Define the file suffixes for C++ source files
.SUFFIXES:
.SUFFIXES: .c $(SUFFIXES)

# Define the command to be used for compilation
CC = gcc

# Define compiler options
OPTS = -ansi -Wall

# Define the object files to be used
OBJS = diskio.o

# Describe how to build the executable file
sdump: sdump.o $(OBJS)
        $(CC) -o sdump sdump.o $(OBJS)

# Describe how to create the object file
.c.o:
        $(CC) $(OPTS) -c $<

# Describe actions necessary to clean the current directory
clean:
        rm -f *.o sdump
```

```
/*
  sdump.c

  Read and display the boot sector of a 1.44 Mb diskette image file.

  MAT 4970
  Bill Slough
*/

#include "diskio.h"
#include <stdlib.h>

int main() {
    /* Read and display the boot sector */
    char buffer[BLOCKSIZE];
    int result = DisketteRead("floppy-image", 0, buffer);
    if (result != -1) {
        DisplaySector(buffer);
        exit(0);
    }
    else
        exit(1);
}
```

```
/*
  diskio.h

  Header file for disk I/O operations.

  MAT 4970
  Bill Slough
*/

#ifndef DISKIO_H
#define DISKIO_H

/* Number of bytes per sector */
#define BLOCKSIZE 512

/* Sector limits */
#define MINSECTOR 0
#define MAXSECTOR 2879

/* Read a sector from the disk */
int DisketteRead(char * filename, unsigned int sector, char * buffer);

/* Write a sector to the disk */
int DisketteWrite(char * filename, unsigned int sector, char * buffer);

/* Display one sector */
void DisplaySector(char * values);

#endif
```

```
/*
  diskio.c

  Basic I/O operations for 1.44 Mb diskettes.

  MAT 4970
  Bill Slough
*/

#include <stdio.h>
#include <stdlib.h>
#include "diskio.h"
#include <fcntl.h>
#include <unistd.h>

int DisketteRead(char * filename, unsigned int sector, char * buffer) {
    /* open the file for reading */
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "No such file: %s\n", filename);
        return -1;
    }

    /* skip the appropriate number of bytes to reach the desired sector */
    off_t position = lseek(fd, sector*BLOCKSIZE, SEEK_SET);
    if (position != sector*BLOCKSIZE) {
        fprintf(stderr, "Sector not found.\n");
        close(fd);
        return -1;
    }

    /* read a block of data */
    ssize_t nread = read(fd, buffer, BLOCKSIZE);
    if (nread != BLOCKSIZE) {
        fprintf(stderr, "Could not read all bytes of the desired sector.\n");
        close(fd);
        return -1;
    }

    /* close the file */
    close(fd);
    return 0;
}


int DisketteWrite(char * filename, unsigned int sector, char * buffer) {
    /* open the file for writing */
    int fd = open(filename, O_WRONLY);
    if (fd == -1) {
        fprintf(stderr, "Could not open the file for writing.\n");
        return -1;
    }

    /* skip the appropriate number of bytes to reach the desired sector */
    off_t position = lseek(fd, sector*BLOCKSIZE, SEEK_SET);
    if (position != sector*BLOCKSIZE) {
        fprintf(stderr, "Sector not found.\n");
        close(fd);
        return -1;
    }

    /* write a block of data */
    ssize_t nwritten = write(fd, buffer, BLOCKSIZE);
    if (nwritten != BLOCKSIZE) {
        fprintf(stderr, "Could not write all bytes of the desired sector.\n");
        close(fd);
        return -1;
    }

    /* close the file */
```

```
71      close(fd);
72      return 0;
73  }
74
75
76  void DisplaySector(char * values) {
77      /* output the bytes, in hexadecimal */
78      const int PerRow = 16;
79      int i;
80      for (i = 0; i < BLOCKSIZE; i++) {
81          /* view the ith byte as an unsigned integer quantity */
82          unsigned char current_byte = (unsigned char) values[i];
83
84          /* output the value of this byte as a two-digit hexadecimal quantity */
85          printf(" %02x", current_byte);
86
87          /* advance to the next row when necessary */
88          if ((i + 1) % PerRow == 0)
89              printf("\n");
90      }
91  }
```