CPSC 313: Computer Hardware and Operating Systems
Assignment #4, due Thursday, April 9th, 2015 at 22:00

NOTE: Late assignments are penalized at 33.333% per day pro rated by the minute. With this being the end of term, late assignments will not be accepted after 8:00 AM Saturday April 11, 2015

## Introduction and Objectives

This assignment is all about file systems. While completing this assignment,

1. You will gain additional practical experience writing code in the C programming language.

2. You will be exposed to the source code management through git and stash.

3. You will gain a better understanding of the implementation of at least one file system (specifically, the MS-DOS file system that is used by most digital cameras and MP3 players).

The assignment is divided into three parts. The division is meant to provide guidelines to help you allocate time when you are working on the assignment, and to help you make steady progress on it instead of leaving it to the last minute. Although the assignment is broken into three parts you are not required to handin the intermediate parts and will handin only the completed assignment. However, you are strongly encouraged to push things to Stash on a regular basis. Parts of the assignment can be challenging so you should get started early. If you want you can work with one other person. Groups of three or more people are not permitted.

The program you are to write, named **fatinfo** is to be written from scratch using C and it must compile and run on the undergraduate Linux machines provided by the department. If you are you using your own machine make sure to test that it compiles and runs as expected before handing it in. Keep in mind that the department machines are all running a 64 bit version of Linux and as a result there could be differences in the function prototypes provided by different include files and of course there are always the potential issues that arise when porting from a machine with one native word size to another with a different size. Also, you are not allowed to use C++ to complete this assignment. You are only allowed to use functions from the standard C library. In particular, this means that you are *not* allowed to use a library of functions that deal with FAT file systems from somewhere on the Web.

When writing this program you are to use good software design practices for the naming of functions, variables, and the organization of functions into separate files. In particular this means that you are not to put all of your functions into a single file. Instead, put related functions into the same file. For example, putting functions dealing with block I/O together, and then putting directory functions together would make a lot of sense. To deal with compiling these separate files into a single executable you will want to use a makefile.

As with past assignments you will need to register at the provided link to get a copy of the source code and file structure to use as a starting point. Unlike the previous assignments you are being provided with very little initial code.

The provided repository contains a suggested collection of files to place your code, function prototypes, structure definitions etc. into, along with a Makefile. These files are just suggestions and you are free to delete, rename, or add to the collection of files that comprise this assignment, however you must keep your git repository upto date and it must always be the case that the following make commands work:

- *make clean* - this will always clean out the .o files and the executbles
- *make depend* - this will rewrite the makefile to build a proper list of dependencies. As long as you keep the SRC line in the Makefule upto date this should work.
- *make* or *make fatinfo* - this will cause the fatinfo program to be built. NOTE your program must build and compile without producing any warnings. You are not permitted to add compiler directives or directives into the C code to suppress warnings. However, you make get warnings when you do a *make depend* and that is acceptable.

## Part 1

You should aim at completing this part of the assignment by Monday March 30, 2014. The goal of this part is to help you become comfortable using random access techniques to access the data in a file and then interpreting some of the files contents. On connect, in the resources section associated with this assignment, are several files that contain images of various FAT file systems. The FAT file system comes in various formats and your code is required to work with both FAT12 and FAT32 versions. The primary difference is in the way information is represented in the FAT table and the fact that FAT32 uses 32 bit addresses while FAT12 uses 12 bit addresses.

The task for this first part is to write code to output the following information about this file system:

- its sector size.
- its cluster size in sectors.
- the number of entries in its root directory. - This is only meaningful for FAT12. For FAT32 the root directory is like all other directories and does not have a fixed size whereas the FAT12 root directories have their size fixed when the file system is initialized. Consequently for FAT32 you will print 0.
- the number of sectors per file allocation table.
- the number of reserved sectors on the disk.
- the number of hidden sectors on the disk.

• the sector number of the first copy of the file allocation table.

• the sector number of the first sector of the root directory. - The way to determine this varies between FAT12 and FAT32.

• the sector number of the first sector of the first usable data cluster. Keep in mind that in both systems the first 2 clusters (0 and 1) are "reserved" and are not used. (The FAT table will have entries for these clusters but the values in the FAT table have a special meaning.) Also in FAT12 cluster 2 starts right after the root directory whereas in FAT32 it starts right after the FATs.

All of this information can be computed from the information stored in the first (BOOT) sector of the file system. You may assume that if bytes 17-18 are 0 then you are dealing with a FAT32 system otherwise you are dealing with a FAT12. Other than figuring out some of the basic information about the file system the other major area these two systems differ in is how the FAT is used. Many of the details about the MS-DOS file systems can be found here:

http://www.win.tue.nl/∼aeb/linux/fs/fat/fat-1.html

(Additional information is available in the section of Connect devoted to this assignment.) One of the sentences on this page is somewhat confusing. Under the heading "FAT12", you will find the statement:

> Since 12 bits is not an integral number of bytes, we have to specify how these are arranged. Two FAT12 entries are stored into three bytes; if these bytes are $uv$, $wx$, $yz$ then the entries are $xuv$ and $yzw$.

Here is a simpler interpretation of the second sentence: Suppose you have the bytes $b_1$, $b_2$ and $b_3$ in that order. You can extract the two FAT12 entries by treating these three bytes as a single little-endian 24 bit unsigned integer $x$. To get the FAT12 entry that occupies the low-order 12 bits *and* $x$ with 0x0FFF. To the get the FAT12 entry that occupies bits 12 to 23 bits shift the $x$ value to the right by 12 so that bits 12 to 23 are now bits 0 to 11. Unless you are 100% sure that the top order 8 bits were 0, *and* this value with 0x0FFF to ensure that only the 12 bits of interest are captured.

Once you have extracted the information from the first sector you are to print the information out in the exact format shown in the sample output.

When writing your code for part 1, be sure to make it general enough to use again for parts 2 and 3. Note that you are not allowed to use any I/O functions to read the contents of the disk images. You must use *open()* to get a file descriptor to the file and then use *mmap()* to map the file into the process's address space. *mmap()* returns you a pointer to a location in memory where the contents of the file are. You can then use pointer arithmetic to compute the addresses needed to access all parts of the disk image.

To modularize your code you might find it useful to define a `struct` like `filesystem_info` that contains most of the information about the file system that part 1 asks you to determine. It is strongly recommend that you write functions to:

• Copy a 512 byte sector into a buffer in memory to work with, or to just work with the mapped file directly.

• Read a single cluster (In the DOS world the term cluster corresponds to the term block that we have used in class) into a buffer in memory.

• Extract an unsigned integer value from two (or three, or four) specific bytes treated as a little-endian value.

You may find the manual pages for the C library functions `open`, and `mmap()` helpful. To access one of them, simply type

man 2 open

Your test program must take a single argument, the name of the file containing the file system it is supposed to interpret. That is, your `main` function should look at its parameters `argc` and `argv` to determine which file should be used.

## Part 2

You should aim at completing this part of the assignment by Friday April 3, 2015. The goal of this part is to be able to deal with directory entries, and extract information about the files a directory contains. More specifically, you should add to your code from part 1 so it can output the following information about each file in the root directory:

• Its name, including the extension if one is present.

• Whether or not this file is a directory.

• The number of the first cluster containing its data.

• Its size (note that directories have a size of 0).

This part of the assignment should be relatively easy once you have completed part 1. Note that the format for the directory entries is exactly the same for FAT12 and FAT32. One thing to be aware of is that the parent of a subdirectory is the root directory then the cluster number for that directory is sometimes given 0 so you will have to be careful about that in the next section.

## Part 3

To complete this part, the program is to print out the information from part 2 about every file in the file system, **including files in subdirectories**. The name printed for each file is to include its full path name (for instance, `ASS2/MCHEME.TXT`). The program is also to print out the list of clusters used for each file using the same format as provided in the sample output.

This part of the assignment will likely be the most challenging because it deals directly with the File Allocation Table. It is recommended that you write functions that:

- determine if a given cluster number is the last cluster in its file.
- retrieve the next cluster number in a file, given the current cluster number.

## Deliverables

You are to submit the assignment using git, just like you have for all previous assignments in this course. The files to submit for this part are:

1. All of your C source and include files. They `must` be commented appropriately, and marks will be taken off it they do not contain enough comments to allow the TAs to figure out what your code is doing.

2. A makefile such that when make is typed in the directory containing your solution the program **fatinfo** is built. This program takes a single argument, the name of the file containing the sample file system the program is to decode. Your code must compile by typing make and there must be no errors or warnings. You are not allowed to change the gcc options in the makefile to alter how and when any warning messages are printed.

3. The filled in version of coverpage.txt. Follow the directions in the coverpage.txt file on how to fill-in and submit the file.

## Makefiles

In the batch of files for this assignment is included a Makefile to be used when building the program. If you add any new files you will need to edit the Makefile so that the line that starts **"SRC ="** contains the name of the new .c file. After modifying and saving the makefile you will need to run makedepend to update the makefile with any .h dependencies. Any time you change any includes in any of your .h or .c you need to rerun makedepend. A more detailed description on how to use make and how makefiles work can be found at:

http://www.ugrad.cs.ubc.ca/~cs219/CourseNotes/Make/intro.html