COMP 3430 Winter 2013 Assignment 3

Due Date: Wednesday, Apr 10, 11:59 PM

Notes and Style

- must be written in C
- your assignment code must be handed in electronically using the D2L drop box. Try early and re-submit later. Include a Makefile. Confusion over D2L at the last minute is your own fault.
- Use common-sense commenting. Provide function headers, comment regions, etc. No particular format or style is required.
- Error checking is extremely important in real world OS work. However, it is complex and can get in the way when trying to learn these concepts. You can assume extreme cases will not happen such as lack of memory or disk failure (and thus can assume that, e.g., read and write will never fail). Error check for things that you should report to the user or recover from, e.g., missing file.
- **PARTICULARLY FOR THIS ASSIGNMENT** error checking can be a proactive anti-bug strategy to help you catch things immediately

FAT32 library

You will implement a library to read directly from a FAT32 file system. This assignment will require you to have a USB stick (any one will do… I tested on a Transcend 4GB) that is formatted fat32. Both windows and linux (the machines in the linux lab) let you do this, and probably MAC as well. Your program will work as follows, assuming your program is called `fat32`:

```
./fat32 /dev/somedevice
```

where `somedevice` represents your USB device.

The Linux lab in E2-468 (Combo 592874) is set up such that you will have full read / write permissions to raw devices. Further, it is set up such that when you plug in a USB stick it will automatically create a link from `/dev/usbstick` to the actual device. You will use `/dev/usbstick` for your assignment.

When you plug the usb stick in, your linux system will automatically mount the disk so that you can use it from your desktop. This is generally a bad thing as, while the drive is mounted, linux may cache changes to the disk and you should not access it raw at the same time. Before using it it is strongly recommended that you unmount the disk

```
umount /dev/usbstick
```

For initial development only, you have been provided with a sample disk image that you can use for initial development – this is a partial image of an 8GB disk with several files on it. If you work on the writing extension, do not use this file as it is only partial. You must fully test your program on an actual USB stick.

Once started, your program will display a prompt "`>`" and will respond to the commands `info`, `dir`, `cd`, and `get`. `info` prints various stats, `dir` lists current directory, `cd` changes directory, and `get` gets a file to the local disk. An EOF signal (ctrl+D) ends the loop.

```
./fat32 /dev/sdc1                    |    >info
```

```
---- Device Info ----
 OEM Name: MSDOS5.0
 Label: NO NAME
 File System Type: FAT32
 Media Type: 0xF8 (fixed)
 Size: 4051681280 bytes (4051MB, 4.052GB)
 Drive Number: 0 (floppy)

--- Geometry ---
 Bytes per Sector: 512
 Sectors per Cluster: 8
 Total Sectors: 7913440

--- FS Info ---
 Volume ID: Transcend
 Version: 0:0
 Reserved Sectors: 38
 Number of FATs: 2
 FAT Size: 7713
 Mirrored FAT: 0 (yes)
>dir

DIRECTORY LISTING
VOL_ID: Transcend

FAT32.H                 4115
JIM                     4
<SOMEFO~1>              0
<TMP1>                  0
<TEO>                   0
---Bytes Free: 1612959744
---DONE
>cd SOMEFO~1
>dir

DIRECTORY LISTING
VOL_ID: Transcend
.
<.>                     0
<..>                    0
DSCN0231.JPG            6660953
IMG_8026.JPG            3463254
<OLD>                   0
---Bytes Free: 1612959744
---DONE
>cd ..
>cd tmp1
>dir

DIRECTORY LISTING
VOL_ID: Transcend

<.>                     0
<..>                    0
---Bytes Free: 1612959744
---DONE
>cd ..
>dir

DIRECTORY LISTING
VOL_ID: Transcend

FAT32.H                 4115
JIM                     4
<SOMEFO~1>              0
<TMP1>                  0
<TEO>                   0
---Bytes Free: 1612959744
---DONE
>get fat32.h

Done.
>
Exited..
```

You have been provided with a Microsoft white paper on their FAT file systems (Fat12/16/32). You will only worry about the FAT32 implementation so you should use your discretion on reading. Also, pick your battles carefully in this document as not everything pertains to this assignment.

You have been provided with a very minimal fat32.h file that provides a struct data structure for the boot sector. In addition to any other data structures you may need for your implementation, you will need to create similar structs for the FSInfo and fat32Dir types. Note the datatypes used instead of the standard int, etc. Why am I using those? Also note carefully the #pragma commands used and use them for any structures that you tie directly to reading from disk (what do these do and why are they important? http://en.wikipedia.org/wiki/Data_structure_alignment)

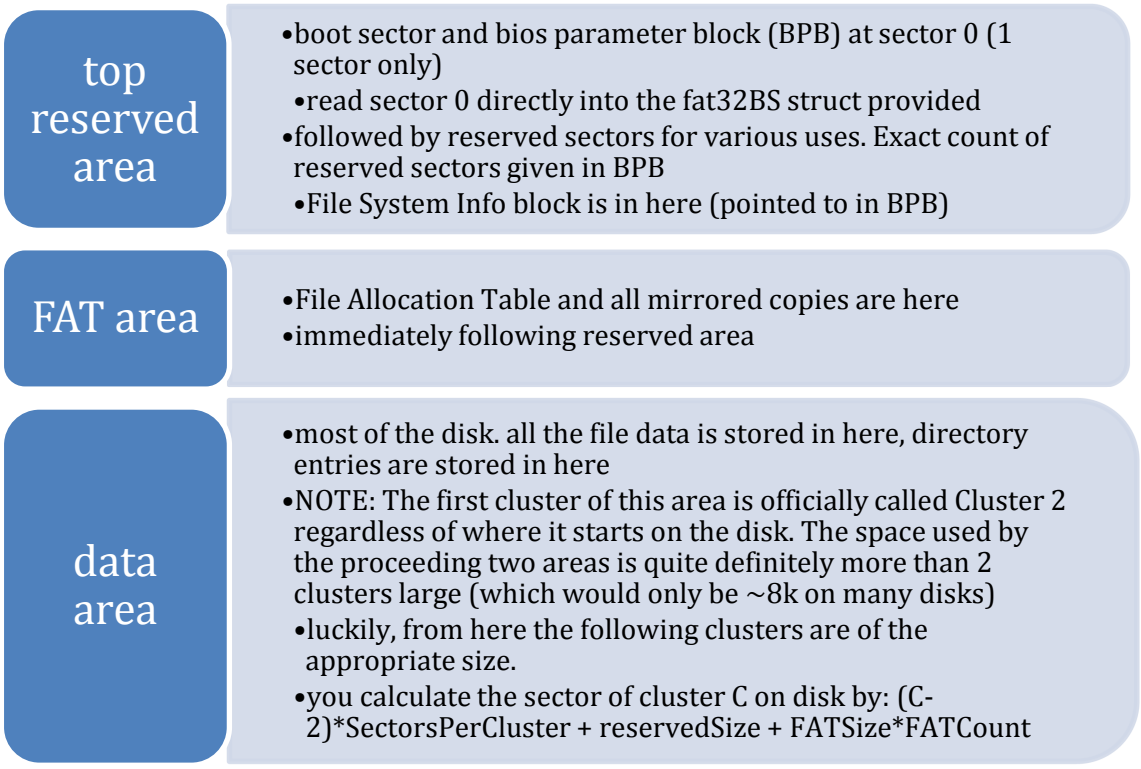
This is a tricky one, and it requires that you have clear understanding of working with bits and bytes, data, memory, and casting in C. Before starting make sure that you are VERY clear on the difference between a byte, a sector, and a cluster, and how they relate. NOTE THAT these values vary by disk and how it was formatted, and **cannot be hard coded**. Don't be shy to ask for help.

Make sure to test your program with large (>1GB) and small (<100b) files, and nested directories. Make sure you test all signatures as you go to ensure correctness (outlined more later).

The above text is pretty much the assignment description and has everything you need to understand what to do. The rest of this document is simply full of additional description, hints and suggestions on how to proceed.

Disk Layout

Here is a quick diagram of the file system layout on disk:



The FAT itself is simply a long list of 32-bit entries (actually, 28 bit+4 reserved bits, be careful!). The first two entries (0 and 1) are a signature. Entry 2+ are simply pointers where each entry represents a cluster on disk (note that the sig entries do not waste usable disk since your data area starts at cluster 2). Let's say you know that a file starts on cluster 2, then the corresponding entry 2 in the FAT will have a pointer to the next cluster in the file. If the file only takes up that one cluster then there is a special END_OF_CLUSTER mark, and there is also a mark for an empty file.

- The FAT can be quite large (multiple MB) so do not load the entire FAT into memory at once. What I did was to create an array of pointers, one entry per sector of the FAT, and load each sector on demand only.
- You may find it useful to create helper functions that take a given sector on disk and calculate which FAT sector and which FAT entry it maps to (as per page 15 of the white paper)

Directories (or folders, lists of files and other folders) are simply normal files on disk with a special format. This format is detailed in the white paper, p23. There are some important points to realize

- The first byte of an entry tells you things such as whether the entry is empty or if it's the last entry.
- You will find empty entries interspersed through the directory. Do not stop searching on an empty entry!
- A directory file can be quite large and span multiple clusters (Check in the FAT!)
- The most confusing part here is that you will find what appear to be garbage entries. Fat12/16 only supported 8-character file names with a 3-character extension. Fat32 added long filename support but did it in a way that is backward compatible – old Fat16/12 systems would only see an 8 char version. E.g., file "somelargefile" would show up as "SOMELA~1" – you may have seen such files up until the mid 2000's. The full filename is *hidden* across several extra added directory entries. For this assignment we will use the short names only (much simpler) and can skip the filler entries. Note that you can filter out these unwanted long-name additional entries by filtering against the ATTR_LONG_NAME attribute.
- An additional side effect is that everything in FAT12/16 was uppercase, so you should stick to uppercase
- directory entries inside a directory point to the cluster where another directory file is stored – look in that file to see what is in the directory
- Being comfortable with casting here can save you a ton of effort. For example

```
uint8_t cluster[CLUSTER_SIZE_IN_BYTES] /* has a directory cluster loaded*/
fat32Dir* dir = (fat32Dir*)(&cluster[0]);
/* work with directory entry*/
dir++; /* move to next directory entry..*/
```

-

Memory and Data Types

Integer overflows and memory layout and alignment are serious issues in this assignment. Note my #pragma comment above. When dealing with addressing specific bytes of a file (or disk..) and you seek using the `lseek` command, you can easily get bytes addressed well out of range of a standard 32 bit integer, particularly if it is signed. When dealing with addresses use the explicit `uint32_t` family of types, and use the `off_t` type (file offset, as in `man lseek`) to be safe.

In addition, you will want to add the following line to the TOP of your fat32.c file before you load the system libraries:

```
#define _FILE_OFFSET_BITS 64
```

This tells the compiler to make the off_t 64 bits wide instead of 32 bits.

Further, it is recommended that you think of reading and writing to disk in units of sectors and clusters. Not only for efficiency reasons, but it will help you conceptualize the assignment a lot easier. Try to work in sectors and clusters as much as possible and only convert to bytes when necessary (e.g., when doing a seek and read)

How to get started and suggestions…

1) load and parse the boot sector / bpb into the struct and write your "printInfo" function as much as possible. **CHECK THE SIGNATURE BYTES**. This tells you if you loaded it correctly.
   A) Check that your FAT16 descriptors are indeed 0.
2) calculate the number of clusters from the BPB as per the "FAT Type Determination" on P14, and ensure it's in the FAT32 range
3) calculate the location of the FAT and check the FAT Signature
   A) implement methods to read specific fat items for disk clusters. Using my above suggested array means these methods load sectors on demand.
4) Load the FSInfo and check its signature entries.

Bonus 1 (5%): Implement long filename support properly.

Bonus 2 (5%): optimize your code so that sets of sequential clusters of a file are read at once. You can identify these cases by inspecting the FAT when you read.

Bonus 3 (10%): implement write support for files (not new directories). All that you need to do is to a) store the file in empty clusters as marked in the FAT, b) update the FAT to point to the new entries, c) generate a directory entry and store it in a directory.