COMP 3413 Winter 2021 Assignment 4

DUE: Dec 9 (last day of class)

Notes and Style

- must be written in C. No C++
- The prof has provided a working demo in d2l (fat32demo)
- your assignment code must be handed in electronically using the D2L drop box. Try early and re-submit later.
- Include a Makefile. If "make" doesn't compile and produce an executable, your assignment **will not be marked**
- Your program must run. Programs that do not run, or segfault before working at all, **will not be marked**. Save early and often, and use version control. Submit often, only submit working versions.
- Your program must compile with –Wall, with no warnings showing. You will lose marks for warnings
- Include a "listing" file, a single .txt file with all your source files concatenated into it. Good header text in EACH file is imperative here. NO LISTING FILE NO MARKS.
- Use common-sense commenting. Provide function headers, comment regions, etc. No particular format or style is required, but code should be readable with good style (e.g., no magic numbers, smaller focused functions, readable code with good variable names).
- Error checking is extremely important in real world OS work, and you must do it rigorously. Error handing, however, is hard in C. Try to handle gracefully when you can, but for hard errors (out of memory, etc.), hard fail (exit) is OK.
- Use gcc, make sure your Makefile works on FCS computers.
- Internet resources may be used for reference but NOT copied. Please do the work yourself. If you use reference code, make sure you understand it. Especially in this assignment, fixing bugs with code you don't understand will hurt you more than just writing it yourself.
- Your assignment will be tested and marked on FCS computers. Test on them before handing

FAT32 library

You will implement a library to read directly from a FAT32 file system. I have provided a real FAT32, well formatted disk image called diskimage on D2L. Your program will work as follows, assuming your program is called `fat32`:

```
./fat32 diskimage
```

Assuming diskimage is in the same directory as your executable.

This is a partial image of an 8GB disk with several files on it for your development purposes. If you work on the writing bonus, do not use this file as it is only partial. You'll have to initialize a FAT yourself.

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

Here is a sample execution with that disk image. You should print out similar information in a similar (or identical) format.

```
[fcsvmuser@fcs-vm-linux-lab solution]$ ./fat32 diskimage
>info
---- Device Info ----
 OEM Name: MSDOS5.0
 Label: NO NAME
 File System Type: FAT32
 Media Type: 0xF8 (fixed)
 Size: 8102624256 bytes (8102MB, 8.103GB)
 Drive Number: 128 (hard disk)

--- Geometry ---
 Bytes per Sector: 512
 Sectors per Cluster: 8
 Total Sectors: 15825438
 Geom: Sectors per Track: 63
 Geom: Heads: 255
 Hidden Sectors: 62

--- FS Info ---
 Volume ID: COMP3430
 Version: 0:0
 Reserved Sectors: 1922
 Number of FATs: 2
 FAT Size: 15423
 Mirrored FAT: 0 (yes)
 Boot Sector Backup Sector No: 6
>
Exited...
[fcsvmuser@fcs-vm-linux-lab solution]$ ./fat32 diskimage
>info
---- Device Info ----
 OEM Name: MSDOS5.0
 Label: NO NAME
 File System Type: FAT32
 Media Type: 0xF8 (fixed)
 Size: 8102624256 bytes (8102MB, 8.103GB)
 Drive Number: 128 (hard disk)

--- Geometry ---
 Bytes per Sector: 512
 Sectors per Cluster: 8
 Total Sectors: 15825438
 Geom: Sectors per Track: 63
 Geom: Heads: 255
 Hidden Sectors: 62

--- FS Info ---
 Volume ID: COMP3430
 Version: 0:0
 Reserved Sectors: 1922
 Number of FATs: 2
 FAT Size: 15423
 Mirrored FAT: 0 (yes)
 Boot Sector Backup Sector No: 6
>dir
.
```

```
>dir

DIRECTORY LISTING
VOL_ID: COMP3430

<LOLCATS>               0
1.JPG           147313
STORY.TXT               112280
JIM.C           408
---Bytes Free: 8085499904
---DONE
>cd LOLCATS
>dir

DIRECTORY LISTING
VOL_ID: COMP3430

<.>             0
<..>            0
FOOD.JPG                36553
TRAVEL.JPG              32398
---Bytes Free: 8085499904
---DONE
>cd ..
>dir

DIRECTORY LISTING
VOL_ID: COMP3430

<LOLCATS>               0
1.JPG           147313
STORY.TXT               112280
JIM.C           408
---Bytes Free: 8085499904
---DONE
>JIM.C

Command not found
>get JIM.C

Done.
>
Exited...
[fcsvmuser@fcs-vm-linux-lab solution]$
```

To work with the disk / disk image, **use the low-level, unbuffered open, close, read, write, and lseek system calls**. This will save you headaches, buffering, conversions, etc., and let you work on the byte level.

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 (you MUST use them)? 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)

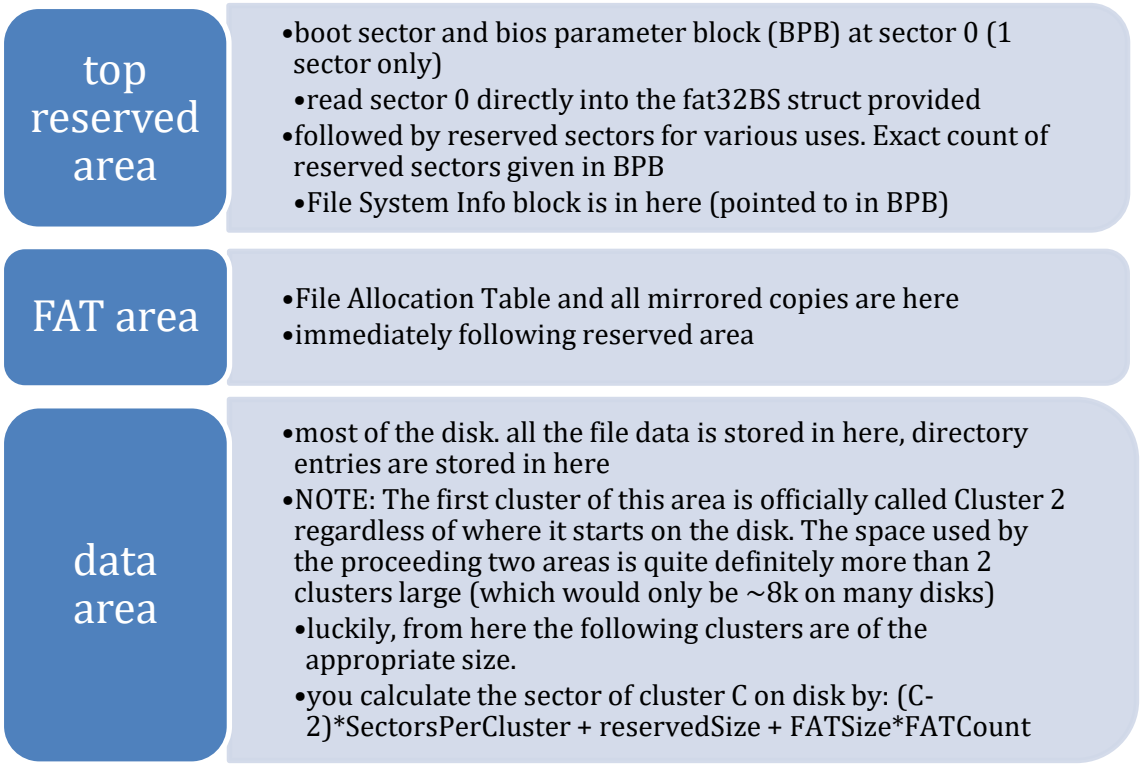
This is a tricky one, and it requires that you have clear understanding of working with bits and bytes, data, memory, and casting in C. Before starting make sure that you are VERY clear on the difference between a byte, a sector, and a cluster, and how they relate. NOTE THAT these values vary by disk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

Feel free to make and test on your own diskimages using various tools (google). 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). At the beginning, focus on working just with the provided diskimage.

The above text is 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 that you need to mask out, be careful!). The first two entries (0 and 1) are a signature. Every other entry represents a cluster on disk – entry 2 (first following the signature) represents cluster 2, entry 10 represents cluster 10, entry n represents cluster n. Thus, your FAT has exactly the number of entries as you have clusters on

disk. This is a lot. (note that the sig entries do not waste usable disk since your data area starts at cluster 2)

Entry 2+ are simply pointers. Let's say you know that a file starts on cluster n. You can find cluster n on disk without the FAT – but what if the file is larger than one cluster? The corresponding entry n 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. There is also a mark for an empty file. Thus, reading a file can be like:

> Start at cluster n. Read cluster n into memory.
>
> Look at entry n in the FAT to see if we're done, or, there is another cluster. There is, at cluster m
>
> Read cluster m from disk as the next chunk of your file. Look at entry M in the fat to see.. repeat.

- The FAT can be quite large 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.
- Be very careful with unit types! You will need to work in sectors, clusters, and bytes, at different times. Maybe create functions to convert the types.
- You will 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 within that sector (as per page 15 of the white paper). This is because the FAT is clearly bigger than one sector.

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! Only the last one!
- A directory file can be quite large and span multiple clusters (Check in the FAT!), so don't just stop at one cluster.
- 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

- Use casting, overlaying structs, and pointer math here to save a ton of effort. For example

```
uint8_t cluster[CLUSTER_SIZE_IN_BYTES]; /* has a directory cluster loaded*/
fat32Dir* dir = (fat32Dir*)(&cluster[0]); /* casting to overlay struct on the memory
*/
/* work with directory entry as a struct*/
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 to be sure you have enough memory, 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) START EARLY. Read this document twice. Then slowly go over the white paper. Don't understand something? It's okay! Just keep moving. You can come back to it later if you need to. I provided a converted word doc so you can remove parts that are not relevant (e.g., FAT16 parts). Most of your bugs will come from not implementing things precisely as this white paper specifies. If you don't understand something, ask me!
2) This assignment is much simpler than AS3 (just implement parts of the white paper), but you can suffer from death by a thousand papercuts! Get started early, and work on it little by little, piece by piece. Code SLOWLY and CAREFULLY. Problems? Read the doc. Check your math, make sure your reads are valid.
3) You will most likely get overflows, null pointers, etc. Some bad math and who knows where you'll end up on disk (or in memory!). **Be extremely defensive in your programming.** Check parameters, ranges, and printf meaningful error messages. This will save you debug time
4) First, 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.

5) Second, calculate the number of clusters from the BPB as per the "FAT Type Determination" on P14, and ensure it's in the FAT32 range
6) Third, calculate the location of the FAT and check the FAT Signature
   A) implement methods to read specific fat items for disk clusters.
7) Load the FSInfo and check its signature entries.

BONUS: make a submission note to the marker about the bonus, if you do it.

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 in one bulk read operation. 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.

Marking

[3] C Style. This will be a messier assignment (lots of old-style variable names), but good function design with comments will help yourself a lot.

[5] info works and gives appropriate information (you will lose marks for deliberately leaving out things the demo prints out, and being inaccurate).

[5] directory listing works at root folder (root directory is slightly different from other directories!)

[5] can change directories

[7] can download a large file, clearly more than one cluster. File isn't corrupted.

-MAKE SURE TO OPEN THE FILE AFTER YOU GET IT to test to see if your code works.

If you can only partially finish some components, let markers know in your submission comments (e.g., "reading files doesn't work completely…see incomplete implementation starting from fat32.c line 200" etc.