Project 3: FAT32 File System Implementation

a.k.a How OpSys ruined my Easter

Project 3: Goals

- Understand basic file system design and implementation
- Have an idea on file system testing
- Know how to do data serialization/de-serialization

Project 3: Outline

- Background
  - Environment Setup
  - Image file mounting
- Project 3
  - Specification
  - Downloading and testing file system image
  - General FAT32 data structures
  - Endian-ness and how to deal with it

Project 3: Environment

- Need to develop it in Linux Environment with root access
- Must compile in the lab machines (runs on Linux Mint)
- Project will be graded in a similar environment

Project 3: System Req.

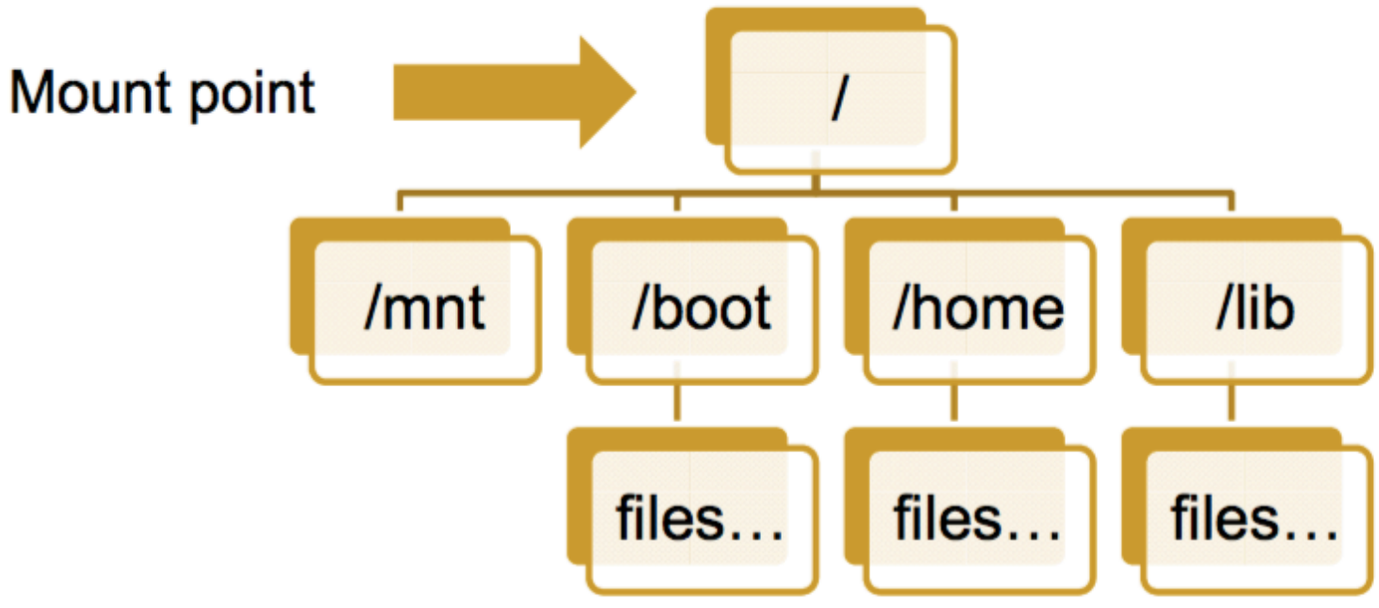
- Need to develop it in Linux Environment with root access
- Must compile in the lab machines (runs on Linux Mint)
- Project will be graded in a similar environment
- Whole project needs to be in C
- At least 64 MB free (for the FAT32 image file) + tax (source code memory)
- Use $df -h to see how much room you have left in your machine

Mounting the image file

- All files accessible in a Unix system are arranged in one big tree
    - Also called the file hierarchy
    - Tree is rooted (starts) at /
- These files can be spread out over several devices
- The mount command serves to attach the file system found on some device to the big file tree

Mounting the image file

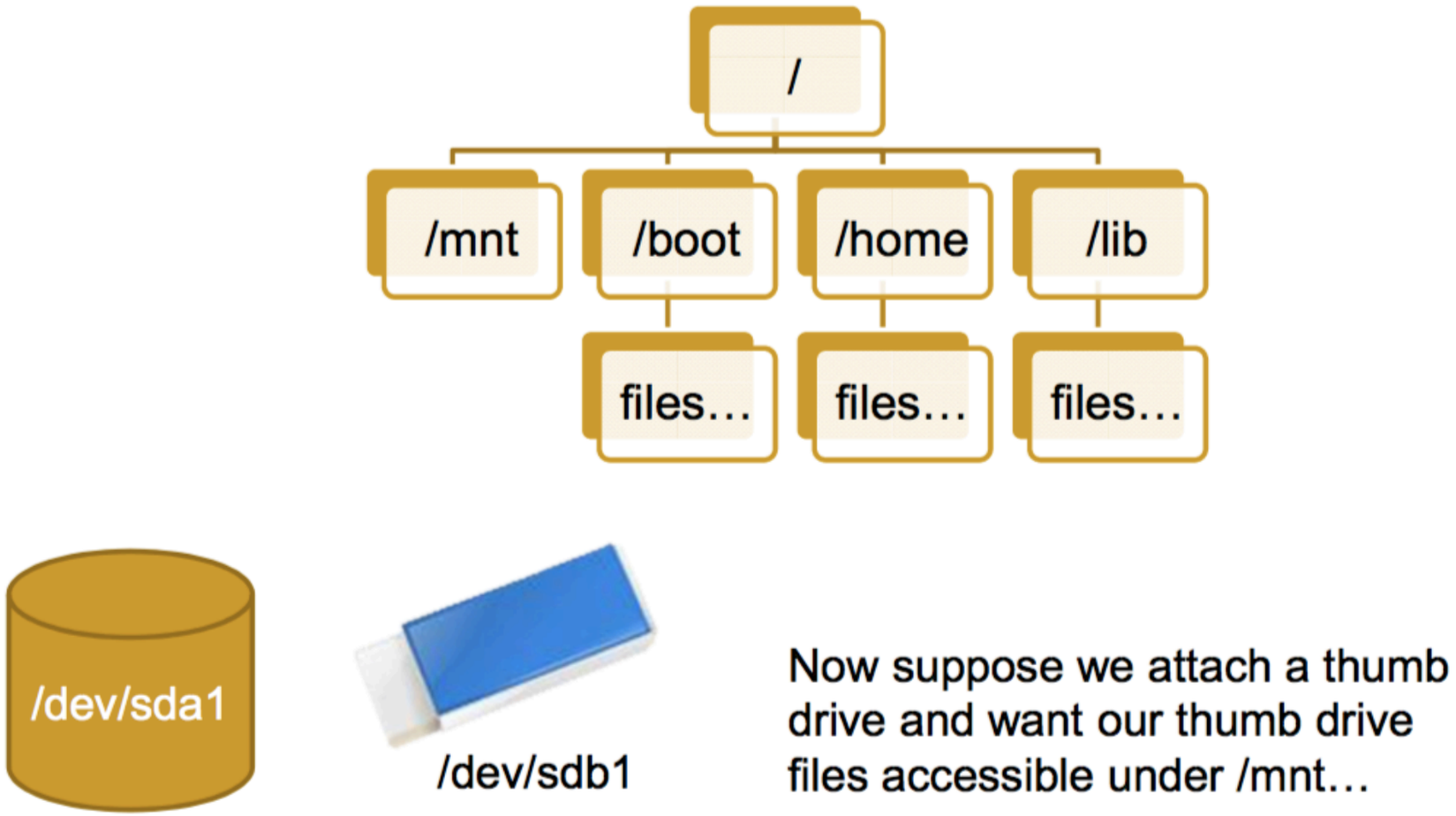
- 2 commands possible:
  $ mount (shows what is mounted and where)



The device sda partition 1 is mounted at “/”. All files and dirs below “/” come from this device.

Mounting the image file

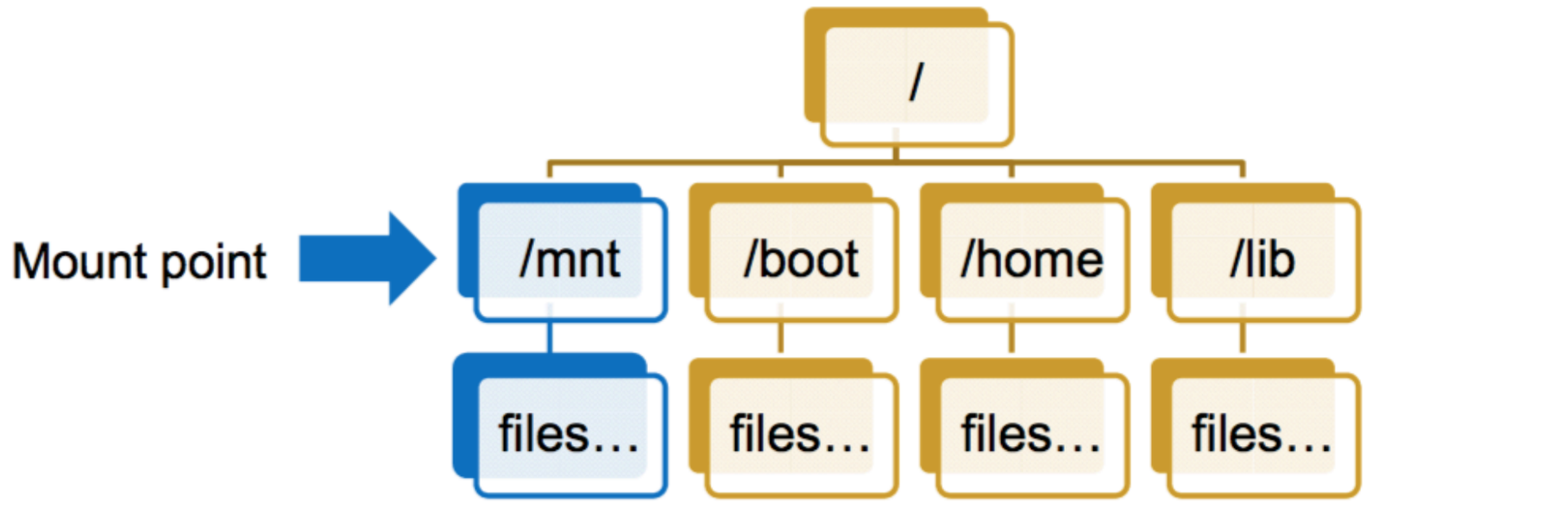
- 2nd type of mount command possible:
  $ mount <device> <mount dir>
  (mounts <device> in <mount dir>)

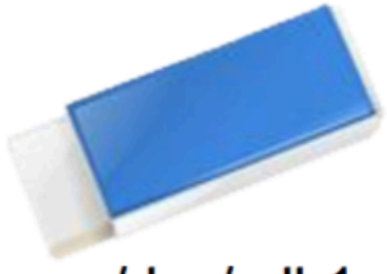


$sudo mount /dev/sdb1 /mnt

Mounting the image file

- What happens if we use the command in prev slide?





/dev/sdb1

Files from the thumb drive are now accessible under /mnt

Mounting the image file

- To unmount disk image use “umount <dir>”
- See what disks mounted at boot, use “cat /etc/fstab”
- You can also use the GUI to mount the image file
- Just double click on the image file/right click on image file and click mount

The big FAT image file

- Manipulation utility will work on a pre- configured FAT32 file system image (basically a file)

- File system image will have raw FAT32 data structures inside (you'll literally be looking at the raw bites stored inside the disk)

- Code needs to open this image file (use fopen function, with rb+)

- Also needs to interpret the raw bytes inside so that the commands like 'ls' to work

Terms and Conditions

- **Byte** – 8 bits of data, the smallest addressable unit in modern processors

- **Sector** – Smallest addressable unit on a storage device. Usually this is 512 bytes

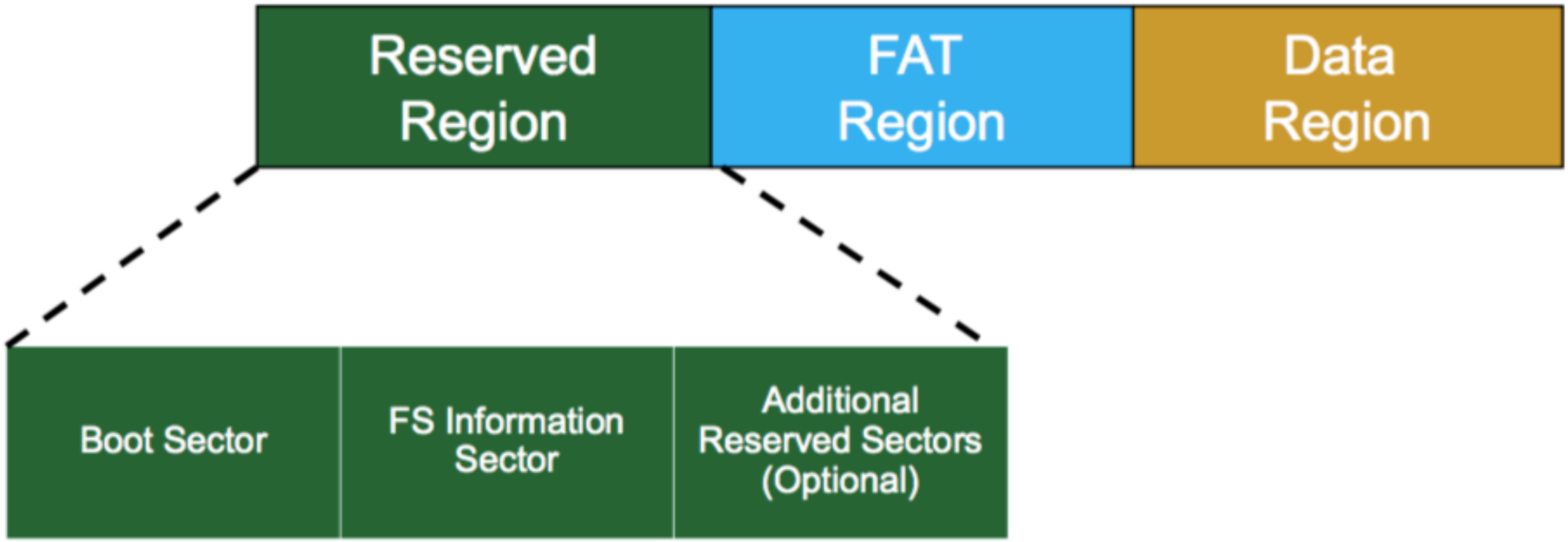
- **Cluster** – FAT32-specific term. A group of sectors representing a chunk of data

- **FAT** – Stands for file allocation table and is a map of files to data

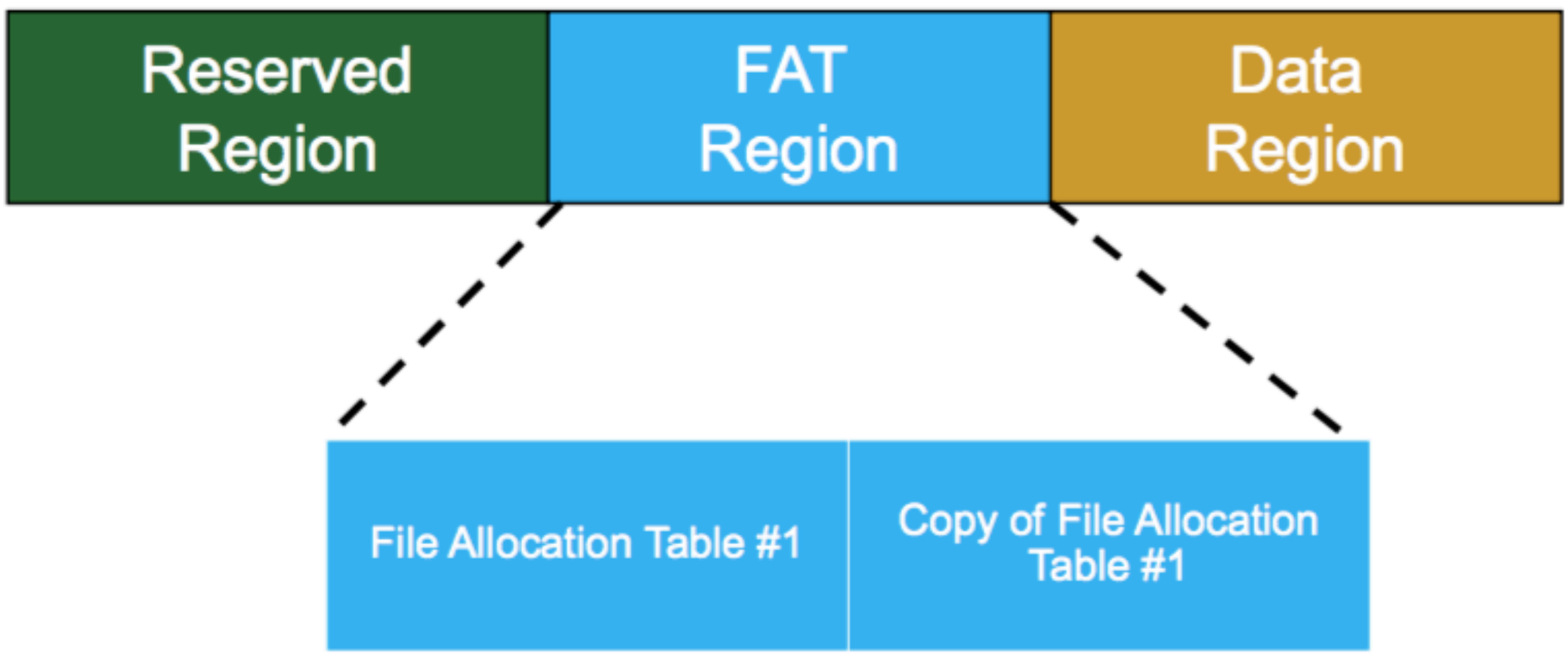
FAT32 Disk Layout

- FAT has 3 main regions - Reserved, FAT, Data
- Reserved Region - Includes the boot sector, the extended boot sector, the file system information sector, and a few other reserved sectors
- FAT Region - Has the FAT: basically a map used to traverse the data region. Contains mappings from cluster locations to cluster locations
- Data Region - Using the addresses from the FAT region, contains actual file/directory data

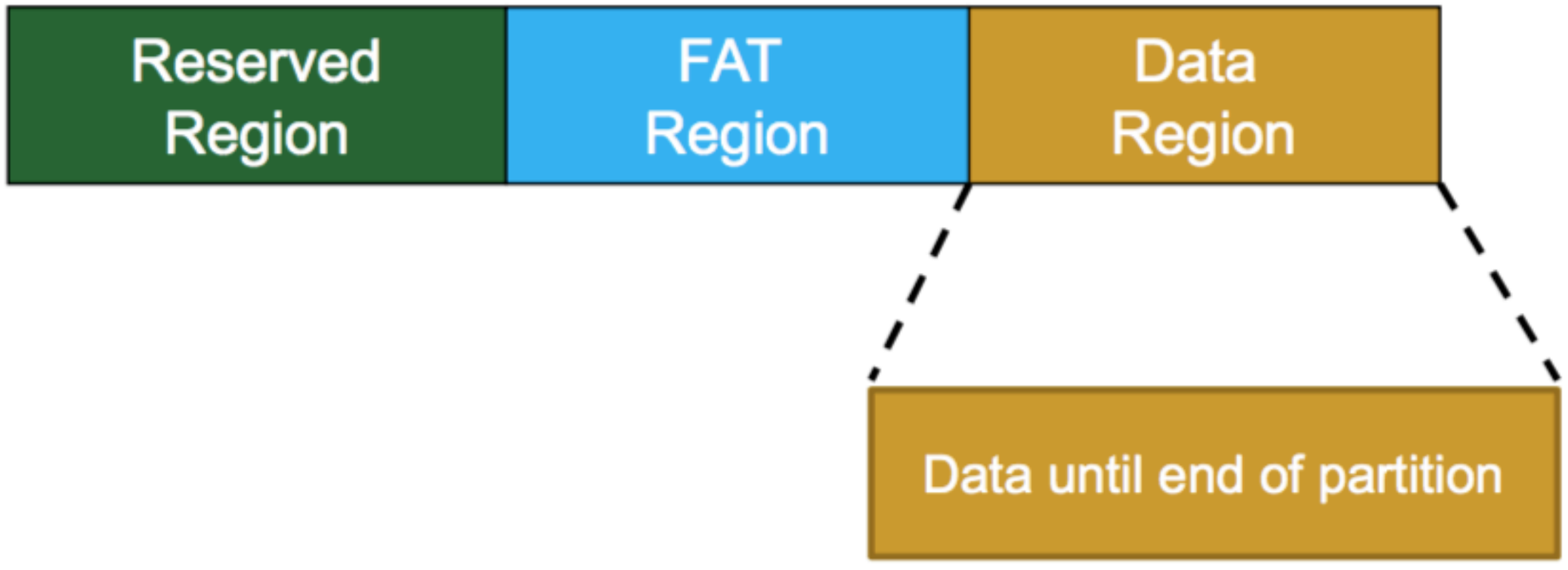
FAT32 Disk Layout



FAT32 Disk Layout



FAT32 Disk Layout



Just Endian Things

- Short value = 15; /* 0x000F */
- char bytes[2];
- memcpy(bytes, &value, sizeof(short));
- In little-endian: bytes[0] = 0x0F ; bytes[1] = 0x00
- In big-endian: bytes[0] = 0x00 ; bytes[1] = 0x0F

Just Endian Things

- Array mapping integer value 13371337 (0x00CC07C9)

index	0	1	2	3
little endian	0xC9	0x07	0xCC	0x00
big endian	0x00	0xCC	0x07	0xC9

FAT32 Boot Sector Parsing

- Boot block is the first 512 bytes (1st sector) of the image disk
- All the vital statistics of the image file contained here
- Fields in this block can be found in the FATSpec.pdf (given to you)

FAT32 Boot Sector Parsing

- Boot block is the first 512 bytes (1st sector) of the image disk
- All the vital statistics of the image file contained here
- Fields in this block can be found in the FATSpec.pdf (given to you) Page 9
- Make a struct consisting of all the fields in that place (THAT ARE RELATED TO FAT32 SYSTEM)

FAT32 Boot Sector Fields

- Size related fields: (BPB_BytesPerSec, BPB_SecPerClus, BPB_RsvdSecCnt, BPB_NumFATS, BPB_FATSz32)

- Cluster number related field: (BPB_RootClus)

- Look at the image to the right

- Do the same for Directory Structures and etc (will come to it later)

```
typedef struct {
    unsigned char jmp[3];
    char oem[8];
    unsigned short sector_size;
    unsigned char sectors_per_cluster;
    unsigned short reserved_sectors;
    unsigned char number_of_fats;
    unsigned short root_dir_entries;
    unsigned short total_sectors_short; // if zero, later field is used
    unsigned char media_descriptor;
    unsigned short fat_size_sectors;
    unsigned short sectors_per_track;
    unsigned short number_of_heads;
    unsigned int hidden_sectors;
    unsigned int total_sectors_long;

    unsigned int bpb_FATz32;
    unsigned short bpb_extflags;
    unsigned short bpb_fsver;
    unsigned int bpb_rootcluster;
    char volume_label[11];
    char fs_type[8];
    char boot_code[436];
    unsigned short boot_sector_signature;
}__attribute((packed)) FAT32BootBlock;
```

Functions to be implemented : exit()

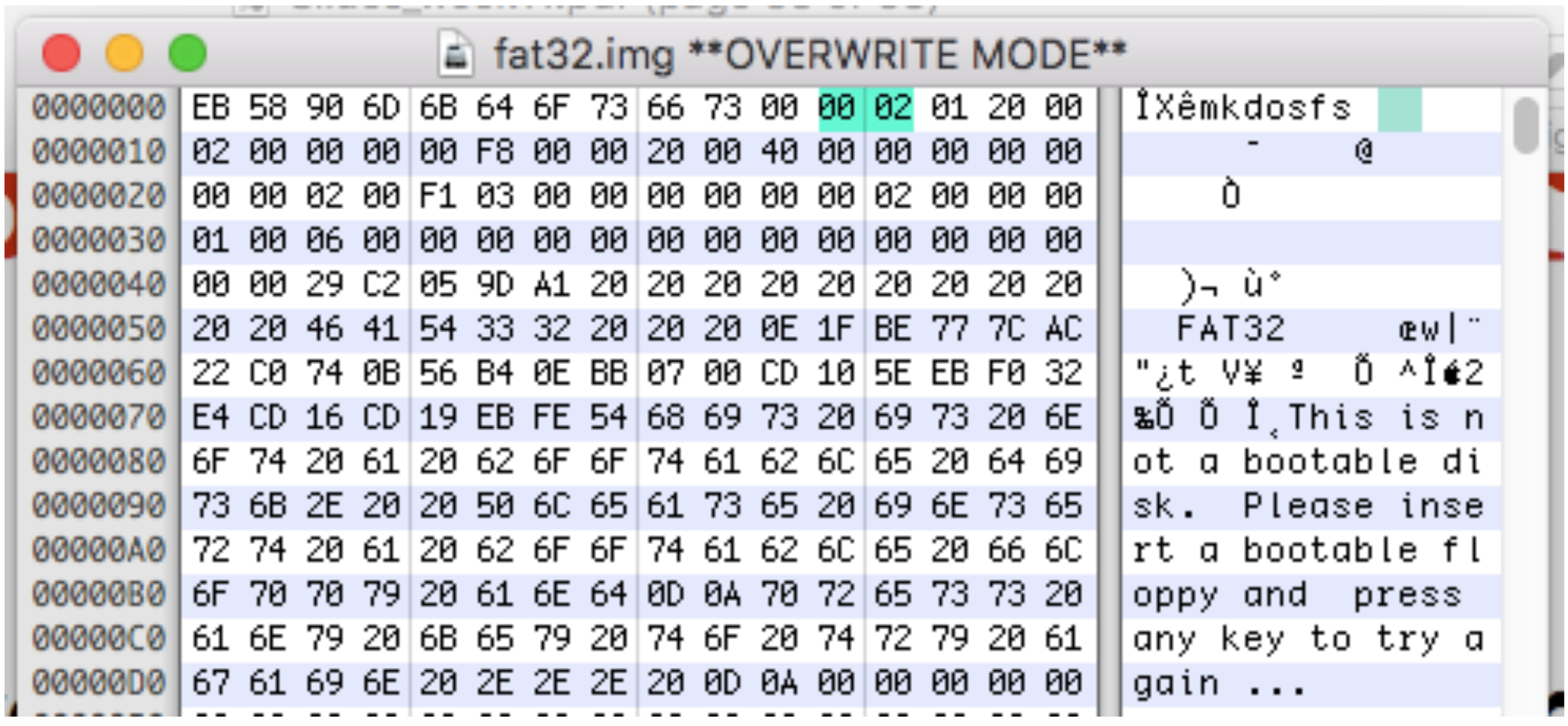
REALLY?????

Functions to be implemented : info()

- Seek to the beginning of the image file
- Read the sector, and store it in a FAT32BootBlock variable
- FAT32BootBlock’s attribute((packed)) normally takes care of endianness of the system.
- Print out the values of the fields shown in previous slide (or more fields as you wish)

Functions to be implemented : info()

- For example, info() should read the BPB_BytesPerSector field as highlighted below:
- BPB_BytesPerSector is found at Offset 11, goes for 2 bytes. Returns 0x0200 = 512 (REMEMBER ENDIAN?)



Functions to be implemented : info()

- For example, info() should read the BPB_RootClus field as highlighted below:
- BPB_BytesPerSector is found at Offset 44, goes for 4 bytes. Returns 0x00000002 = 2 (AGAIN, REMEMBER ENDIAN?)

fat32.img **OVERWRITE MODE**

0000000	EB	58	90	6D	6B	64	6F	73	66	73	00	00	02	01	20	00
0000010	02	00	00	00	00	F8	00	00	20	00	40	00	00	00	00	00
0000020	00	00	02	00	F1	03	00	00	00	00	00	00	02	00	00	00
0000030	01	00	06	00	00	00	00	00	00	00	00	00	00	00	00	00
0000040	00	00	29	C2	05	9D	A1	20	20	20	20	20	20	20	20	20
0000050	20	20	46	41	54	33	32	20	20	20	0E	1F	BE	77	7C	AC
0000060	22	C0	74	0B	56	B4	0E	BB	07	00	CD	10	5E	EB	F0	32
0000070	E4	CD	16	CD	19	EB	FE	54	68	69	73	20	69	73	20	6E
0000080	6F	74	20	61	20	62	6F	6F	74	61	62	6C	65	20	64	69
0000090	73	6B	2E	20	20	50	6C	65	61	73	65	20	69	6E	73	65
00000A0	72	74	20	61	20	62	6F	6F	74	61	62	6C	65	20	66	6C
00000B0	6F	70	70	79	20	61	6E	64	0D	0A	70	72	65	73	73	20
00000C0	61	6E	79	20	6B	65	79	20	74	6F	20	74	72	79	20	61
00000D0	67	61	69	6E	20	2E	2E	2E	20	0D	0A	00	00	00	00	00

```
ÎXêmkdosfs
      -     @
    Ò
  )¬ ù°
  FAT32      œw|¨
"¿t V¥ º  Ő ^Î¢2
‰Ő Ő Î¸This is n
ot a bootable di
sk.  Please inse
rt a bootable fl
oppy and  press
any key to try a
gain ...
```

Now What?

- You have the root directory cluster number from the boot sector.
- Use it to go to the root directory
- FirstDataSector = BPB_ResvdSectCnt + (BPB_NumFATs*BPB_FatSz32)
- FirstSectorofCluster = ((N - 2)*BPB_SecPerClus) + FirstDataSector)
- N is the Cluster Number. FirstSectorofCluster denotes where the particular cluster starts
- FirstDataSector is the Sector where data region starts (should be something around 100400 in hex)
- For root directory, N = BPB_RootClus (= 2 normally)

FAT32 Directory Structure

- Open the FATSpec.pdf file given to you and go to page 23
- Do the same thing that you did with the Boot Sector
- (THAT IS) Make a struct, and put in the fields in order, with memory for each of the field corresponding to their size in bytes

FAT32 Directory Structure

- However, there is more to it than just the struct you made just now
- You also have to take into account the long directory structure preceding the structure you saw
- Go to page 29 of FATSpec.pdf to look at the long directory structure
- In case of situations where you don’t have a long directory entry, the 1st long entry is absent
- Thankfully, for the purposes of this project, you don’t have to deal with long entries, and hence, directory structure consists of 2nd long directory entry, and the short entry (as per page 29)

FAT32 Directory Structure

- The first 2 lines in green correspond to 2nd long entry. The last 2 lines in green is the short entry

Storage of a Long-Name Within Long Directory Entries

A long name can consist of more characters than can fit in a single long directory entry. When this occurs the name is stored in more than one long entry. In any event, the name fields themselves within the long entries are disjoint. The following example is provided to illustrate how a long name is stored across several long directory entries. Names are also NUL terminated and padded with 0xFFFF characters in order to detect corruption of long name fields by errant disk utilities. A name that fits exactly in a n long directory entries (i.e. is an integer multiple of 13) is not NUL terminated and not padded with 0xFFFFs.

Suppose a file is created with the name: "The quick brown.fox". The following example illustrates how the name is packed into long and short directory entries. Most fields in the directory entries are also filled in as well.

2nd long entry → (and last)	42h	w	n	.	f	o	0Fh	00h	chk-sum	x				
	0000h	FFFFh	FFFFh	FFFFh	FFFFh	0000h	FFFFh	FFFFh						
1st long entry →	NOT PRESENT													
Short entry →	T	H	E	Q	U	I	~	1	F	O	X	20h	NT	Rsvd	Created Time
	Created Date	Last Access Date	0000h	Last Modified Time	Last Modified Date	First Cluster	File Size							

0100490	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF				
01004A0	43	20	20	20	20	20	20	20	20	20	20	20	00	00	9D	8D	C			ùç
01004B0	6F	3E	57	45	00	00	9D	8D	6F	3E	12	00	0B	00	00	00	o>WE		ùço>	
01004C0	41	72	00	65	00	64	00	00	00	FF	FF	0F	00	37	FF	FF	Ar e d		ˇˇ	7ˇˇ
01004D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	ˇˇˇˇˇˇˇˇˇˇ			ˇˇˇˇ
01004E0	52	45	44	20	20	20	20	20	20	20	20	10	00	00	E2	8E	RED			,é
01004F0	6F	3E	6F	3E	00	00	E2	8E	6F	3E	03	00	00	00	00	00	o>o>	,éo>		
0100500	41	67	00	72	00	65	00	65	00	6E	00	0F	00	42	00	00	Ag r e e n		B	
0100510	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	ˇˇˇˇˇˇˇˇˇˇ		ˇˇˇˇ	
0100520	47	52	45	45	4E	20	20	20	20	20	20	10	00	00	8A	8D	GREEN			äç
0100530	6F	3E	6F	3E	00	00	8A	8D	6F	3E	04	00	00	00	00	00	o>o>	äço>		
0100540	41	62	00	6C	00	75	00	65	00	00	00	0F	00	55	FF	FF	Ab l u e		U˘ˇ	
0100550	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	ˇˇˇˇˇˇˇˇˇˇ		ˇˇˇˇ	
0100560	42	4C	55	45	20	20	20	20	20	20	20	10	00	64	A8	8D	BLUE			d®ç
0100570	6F	3E	6F	3E	00	00	A8	8D	6F	3E	05	00	00	00	00	00	o>o>	®ço>		
0100580	41	66	00	61	00	74	00	73	00	70	00	0F	00	44	65	00	Af a t s p			De
0100590	63	00	2E	00	70	00	64	00	66	00	00	00	00	00	FF	FF	c . p d f			ˇˇ
01005A0	46	41	54	53	50	45	43	20	50	44	46	20	00	64	38	8F	FATSPEC PDF			d8è
01005B0	6F	3E	6E	4C	00	00	38	8F	6F	3E	0C	15	8A	4E	05	00	o>nL	8èo>		äN

Signed Int | little | (select less data)

64 bytes selected at offset 1049856 out of 67108864 bytes

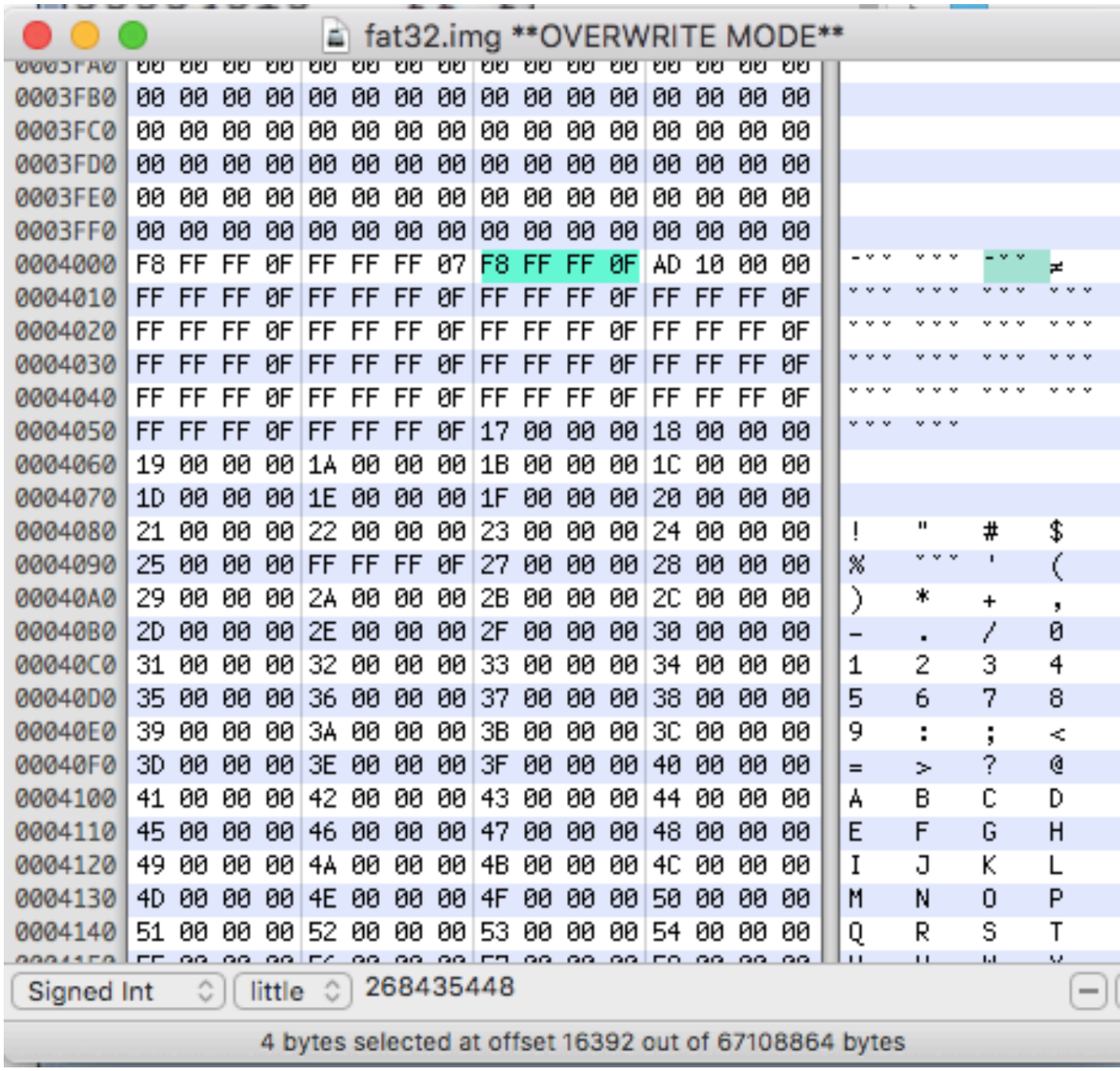
FAT32 Clusters and Directories

- A directory/file may span more than 1 cluster.
- In such a case, you need to look which cluster it goes to next.
- Look up FAT[current_cluster_number]
- If the value is 0x0FFFFFF8 or 0x0FFFFFFF, then this cluster is the last cluster of the directory (the directory ends here)
- Otherwise, FAT[current_cluster_number] will be the cluster_number where the directory continues.
- If FAT[current_cluster_number] == 0, you are in an empty cluster

FAT32 Clusters and Directories

- In case of the root directory, the current_cluster_numbe r is = 2, hence we check FAT[2]

- We see FAT[2] = 0x0FFFFF8 -> the directory does not continue to a new cluster



FAT32 Clusters and Directories

- On the other hand, inside the directory entry, you can find where the first cluster number of the directory

- dir_first_cluster_number = dir.FstClusHI*0x100 + dir.FstClusLO

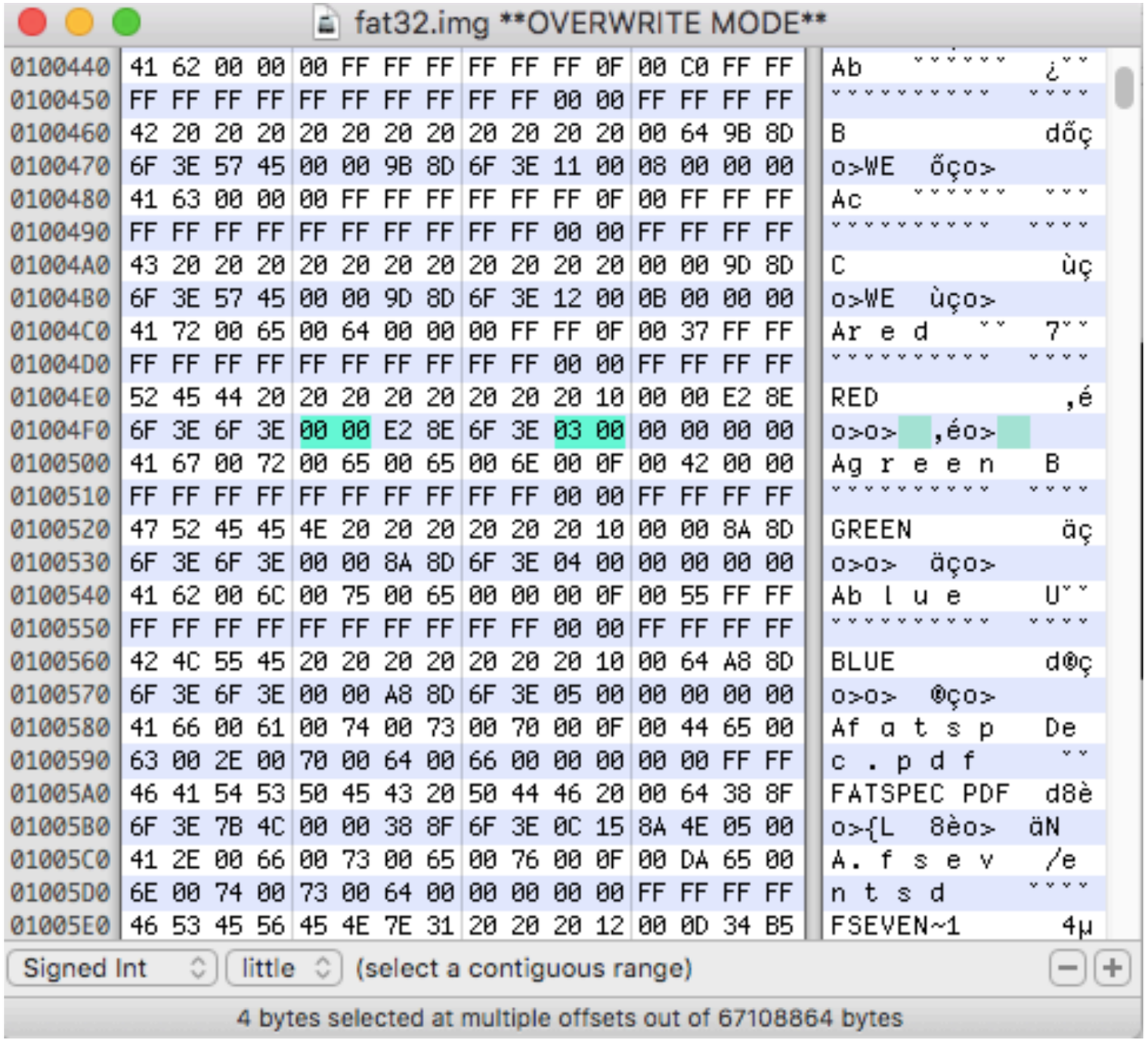
- Also, as both files and directories have directory entries, we differentiate these 2 is by looking at DIR_Attr

- If DIR_Attr == 0x10, then it is a directory, otherwise it is a file

- Other important attributes like Read Only, are given in the FatSpec.pdf (seriously, read that; it has a lot of useful info)

FAT32 Clusters and Directories

- Let’s say we want to check out the “RED” directory
- The first green part is DIR_FstClusHI, the 2nd part is your DIR_FstClusLO
- Hence, RED directory is present at cluster_number 3.
- Also, note the 12th byte of the short entry of RED (line 1004E0, 12th entry in that line). 10 means it is a directory (that entry is the DIR_Attr)



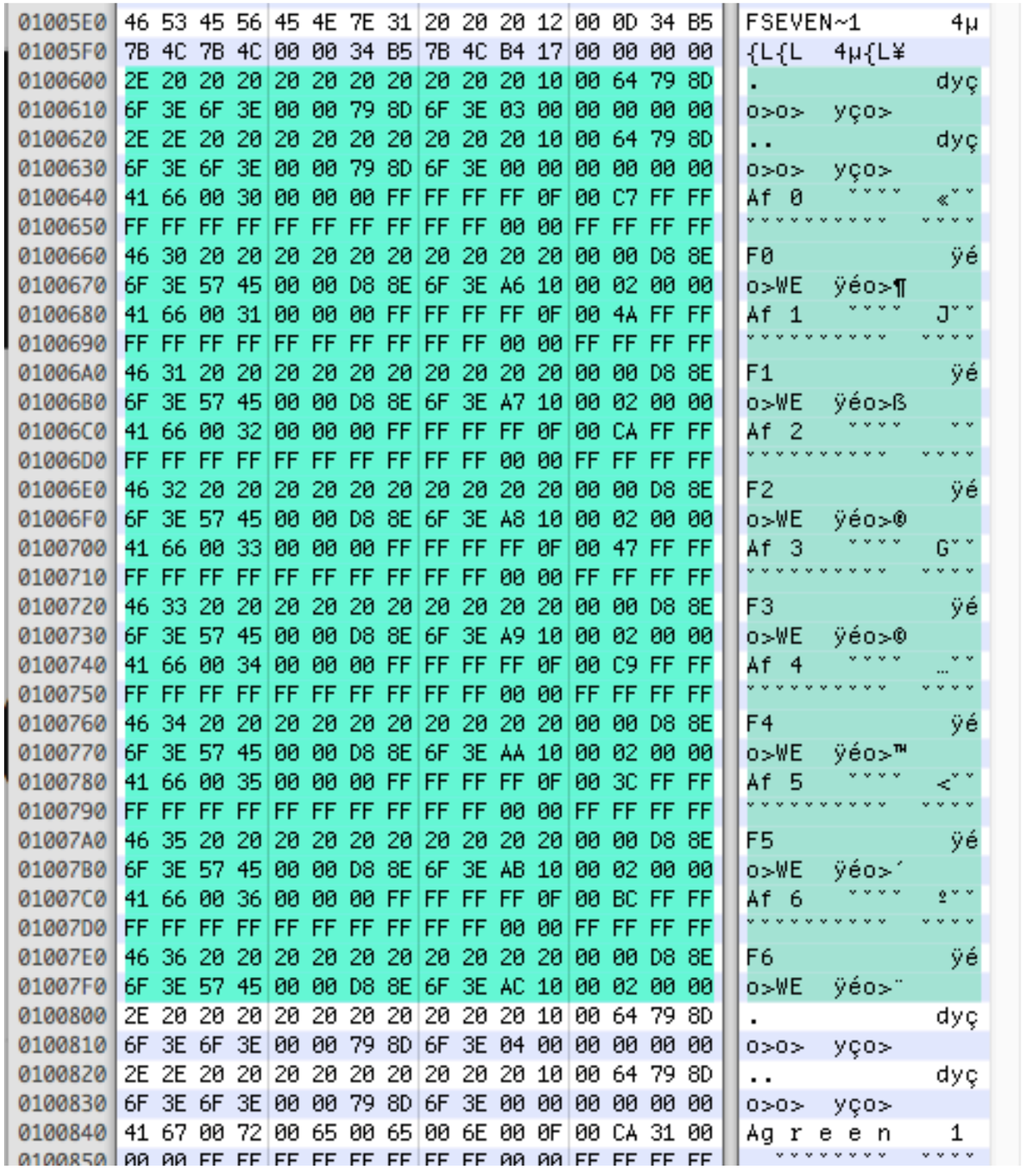
FAT32 Clusters and Directories

- Plug 3 in the FirstSectorOfCluster Equation

- Then seek to that particular part of the file (using fseek preferably)

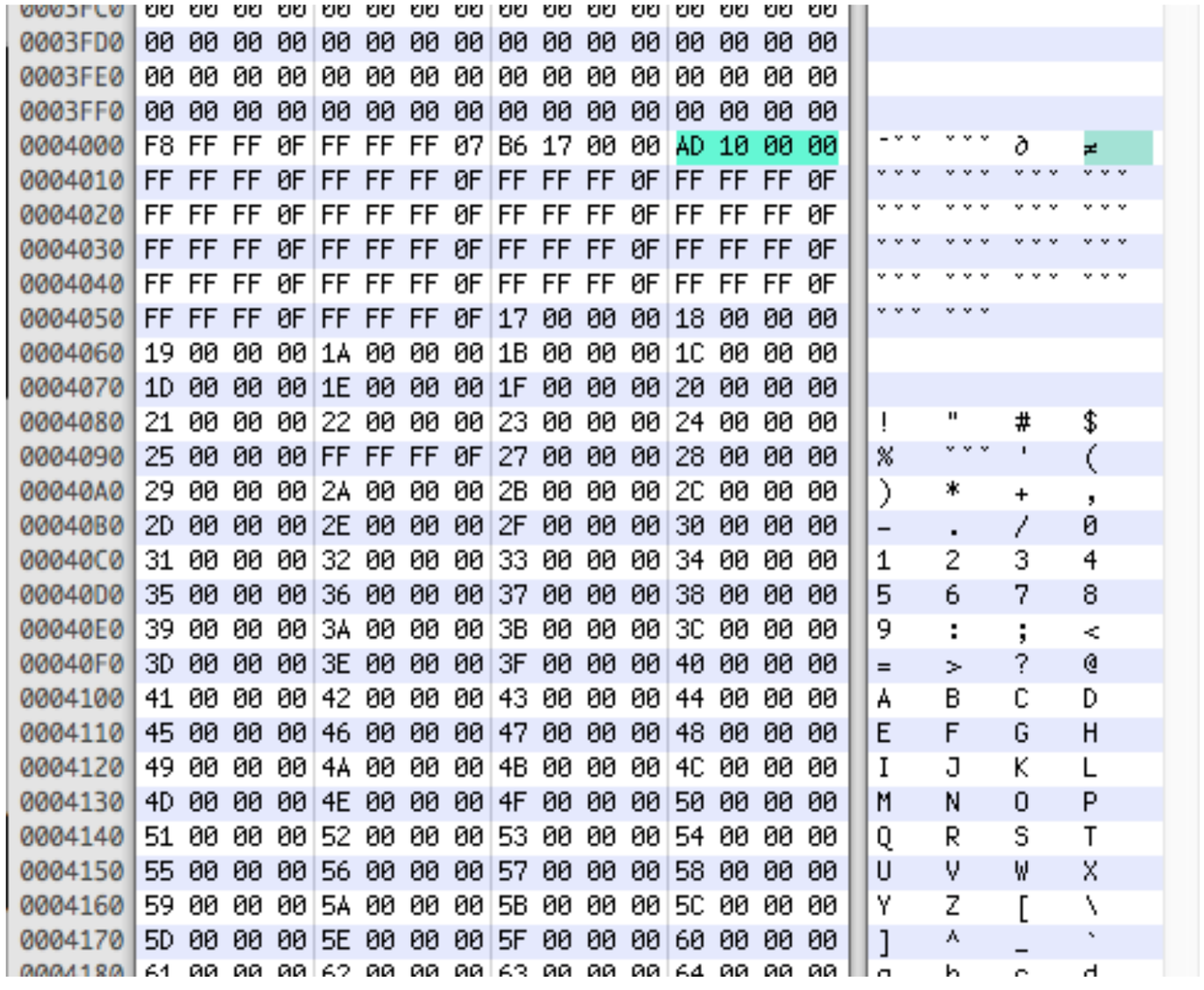
- If you have done it correctly, the program should reach a place highlighted in green (line 100600)

- How do we check if this cluster ends here or not?



FAT32 Clusters and Directories

- Go to FAT[3]
- It has a value of 0x000010AD
- Thats where the directory continues
- Plug that value in N for FirstSectorOfCluster equation
- You should reach the next cluster where RED directory continues



Functions to be implemented: ls

- Make a directory structure like the FAT32DirectoryBlock (remember to reserve some space for the long entry)

- Call an ls function ls(int current_cluster_number {should be the first_cluster_number of a directory})

- Function looks up all directories inside the current directory (fseek, and i*FAT32DirectoryStructureCreatedByYou, where ‘i’ is a counter)

- Iterate the above step while the i*FAT32DirectoryStructureCreatedByYou < sector_size

- When that happens, lookup FAT[current_cluster_number]

- If FAT[current_cluster_number] != 0x0FFFFFF8 or 0x0FFFFFFF or 0x00000000, then current_cluster_number = FAT[current_cluster_number]. Do step 3 - 5 by resetting i

- Else, break

Functions to be implemented: cd DIRNAME

- Make a cd(int current_directory's_first_cluster_number, string DIRNAME)

- Iterate the current directory (using the current_cluster_number) just like before

- Difference is, for every directory entry in the current directory, compare DIR_Name with the DIRNAME string

- If they match AND DIR_attr = 0x10, return DIR_FstClusHI*0x100 + DIR_FstClusLO (this is the first_cluster_number of the Directory remember?)

- If you reach the end of the directory (current_directory's_first_cluster_number = 0x0FFFFFF8 or 0x0FFFFFFF) then there is NO such file with that name in this directory.

Functions to be implemented: ls DIRNAME

- Call cd (current_directory's_first_cluster_number, DIRNAME) function
- Call ls (cluster_number_returned_by_cd)
- Call cd (cluster_number_returned_by_cd, "..")
- DONE

Functions to be implemented: size DIRNAME

- Most of the things are EXACTLY the same as cd

- However, on match, instead of returning the cluster number from the directory entry, return the DIR_Size

- DONE

Project 3: To Do

- Look through the FAT32 Image (using HexEdit or HexFiend)
- Try implementing the 3 functions given
- Think about HOW you would implement the next 4 (mkdir, creat, rmdir, rm) functions

Project 3: Additional Info

- Should be done in groups of 1-3
- One member per group should mail the group names and their email addresses to me by THIS Sunday (1st April, 2018)
- Deadline for the Project - (29th April, 2018 upto 11:59:59 PM)
- Late Submission for Project 3 will have the same policy as Project 2 (10 points off for each day after due date, submissions post 5 days receive 0 points)
- Grading Policy and Grading Appeals same as that given in syllabus
- Deliverables are as given in syllabus (README, Makefile, Project3SourceCode.c)

Project 3: More Additional Info

- You WILL feel Code Rage
- You WILL feel like using the Crucio curse on me
- Won’t work. You aren’t a wizard Harry
- You WILL feel like running a car over me 50 times
- Please Don’t

MAY THE CODE

BE WITH YOU