Project 3: FAT32 File System

Project Overview

- FAT32 is a file system
- What does that actually mean?
- A storage device (ie hard drive) is simply a place to store a bunch of bytes
- Given a device, how do you find a specific file in that large bunch of bytes?
- File systems organize the space available on a storage device into an accessible and navigable format

Implementation Overview

- Implemented in C
- No more kernel programming
  - Program will run as a normal executable
- Given an image file containing raw bytes
  - Open image file as a binary file (use ‘b’ flag)
  - Read user input
  - Manipulate image file according to input

Getting Started

- First, explore the provided fat32 image file

2 options:

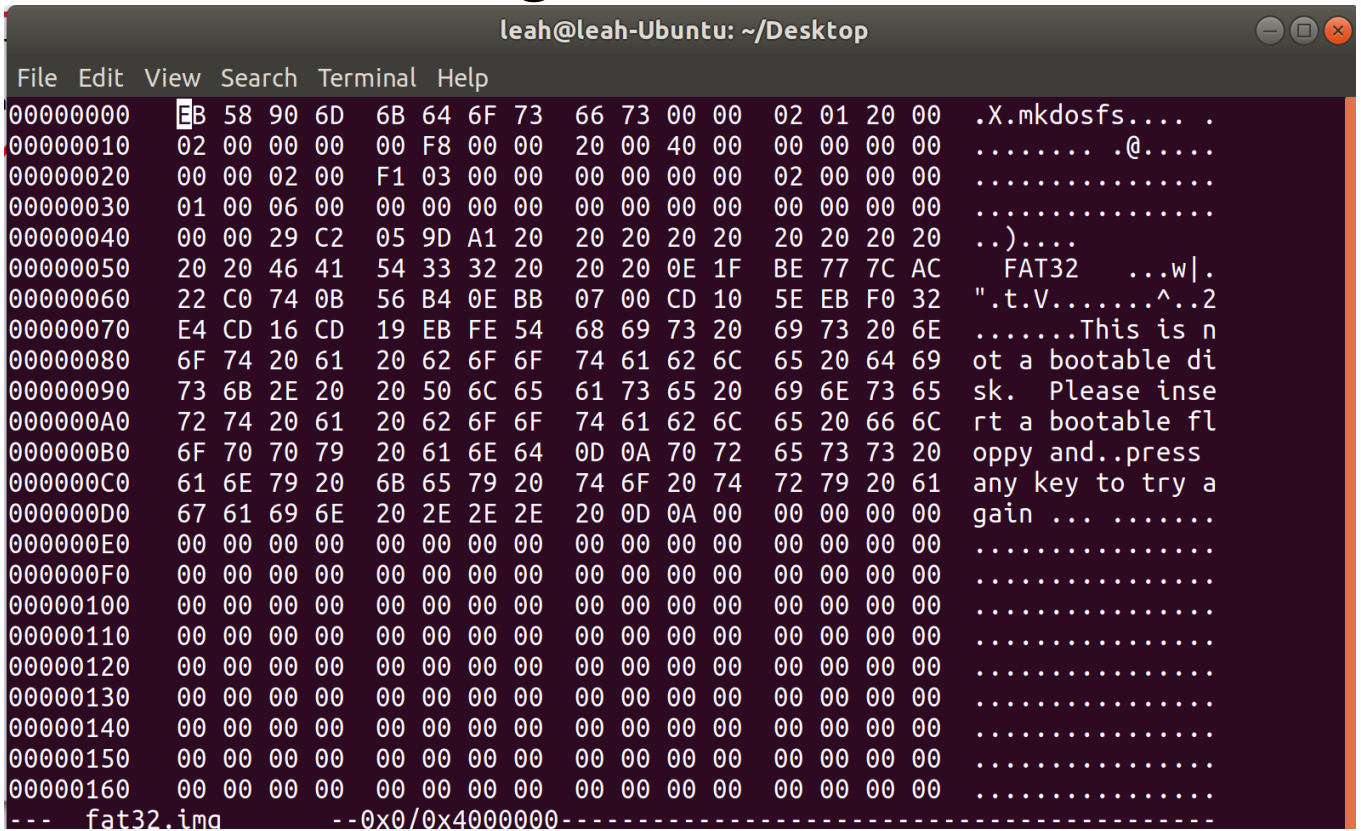
  - Hexedit
  - Mount

Hexedit

- Hexedit often included in Linux distributions
- Free downloads for hex viewers available online
- Allow you to view raw data content of an image file
- Even if you use hexedit, make sure to test your project via mounting before submission and vice versa

Hexedit

- > hexedit fat32.img



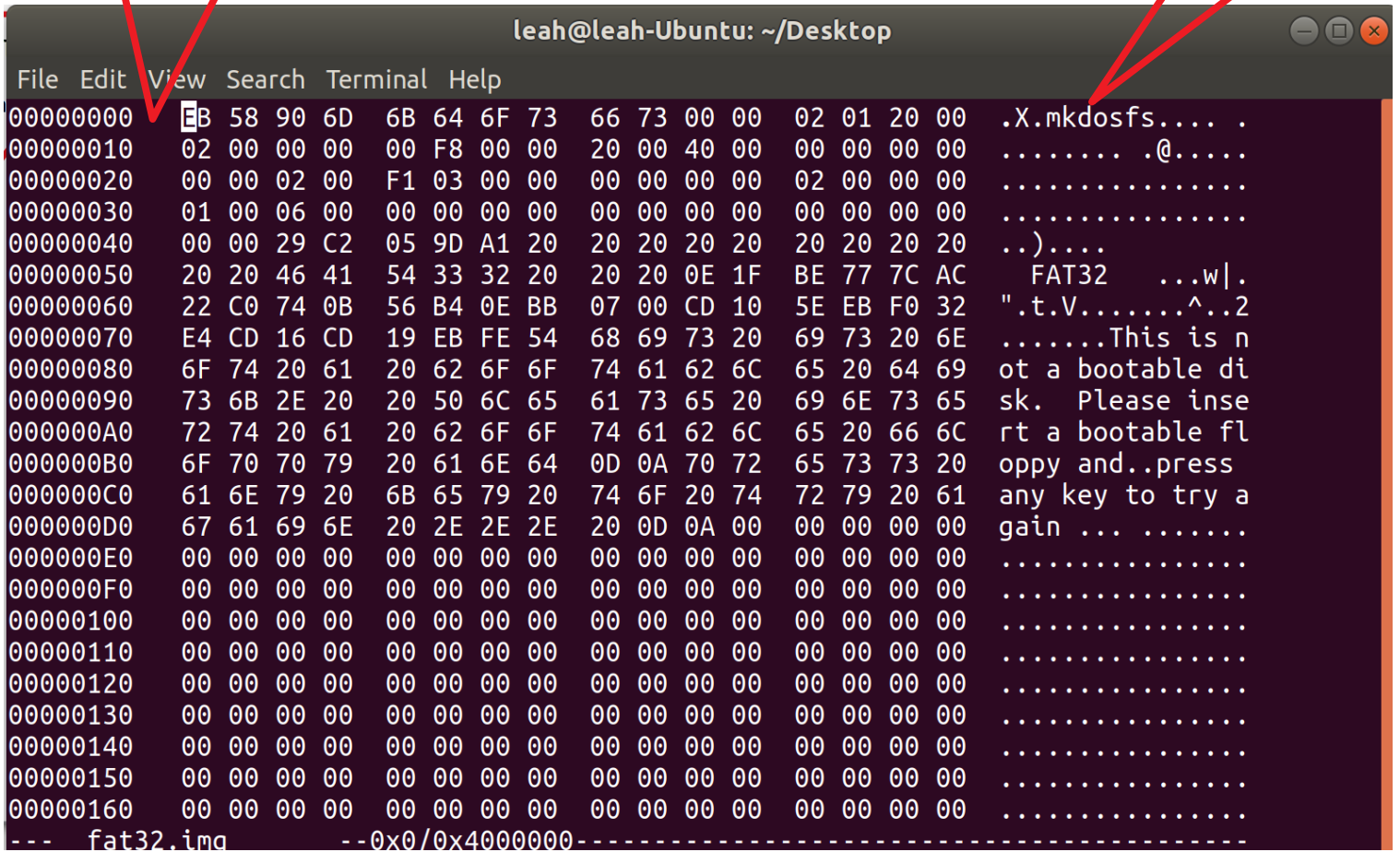
Hexedit

- Data and line numbers shown in hexadecimal
- When possible, the ASCII representation of the data will be shown on the right

Hexedit

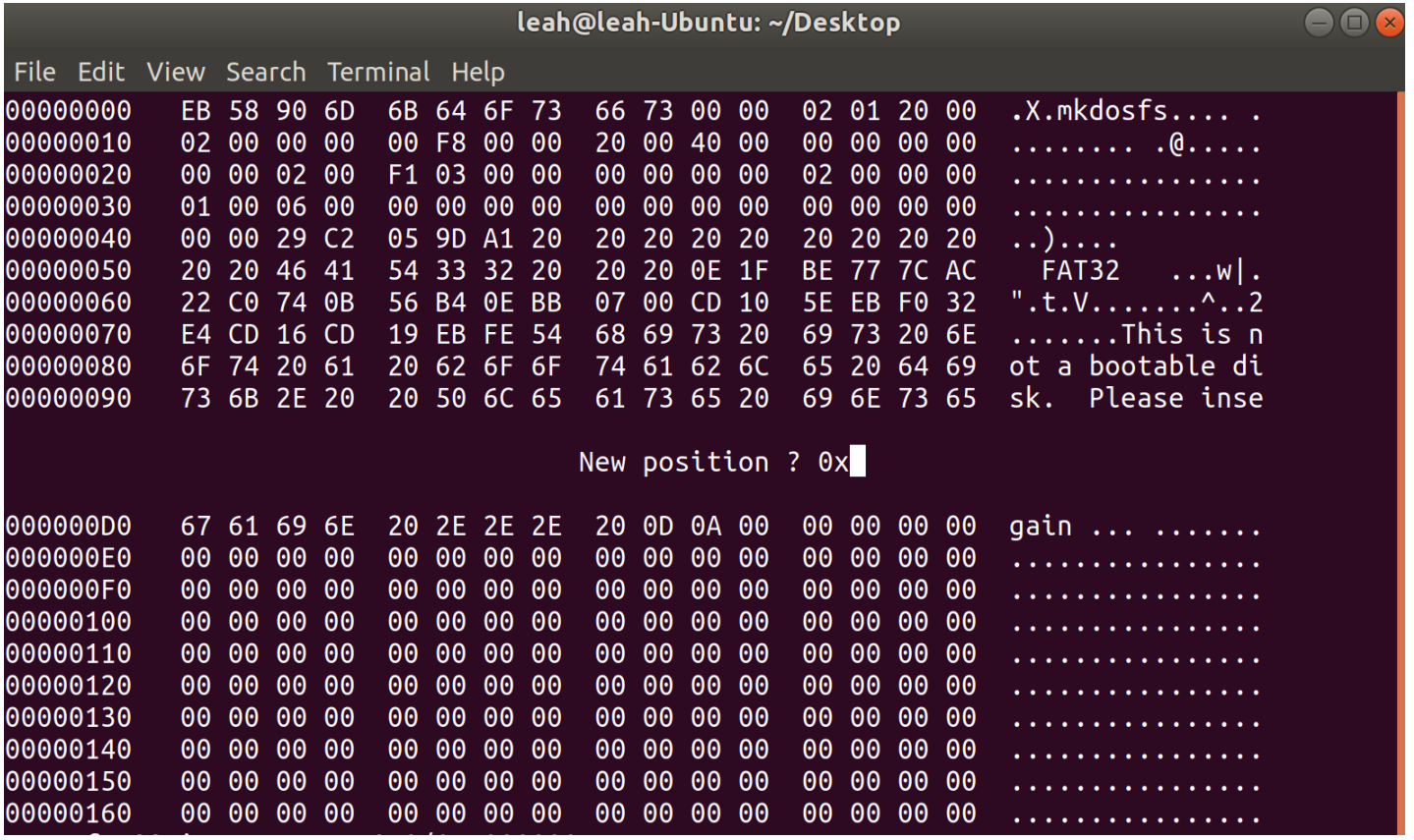
Line numbers and data shown in hex

ASCII representation



Hexedit

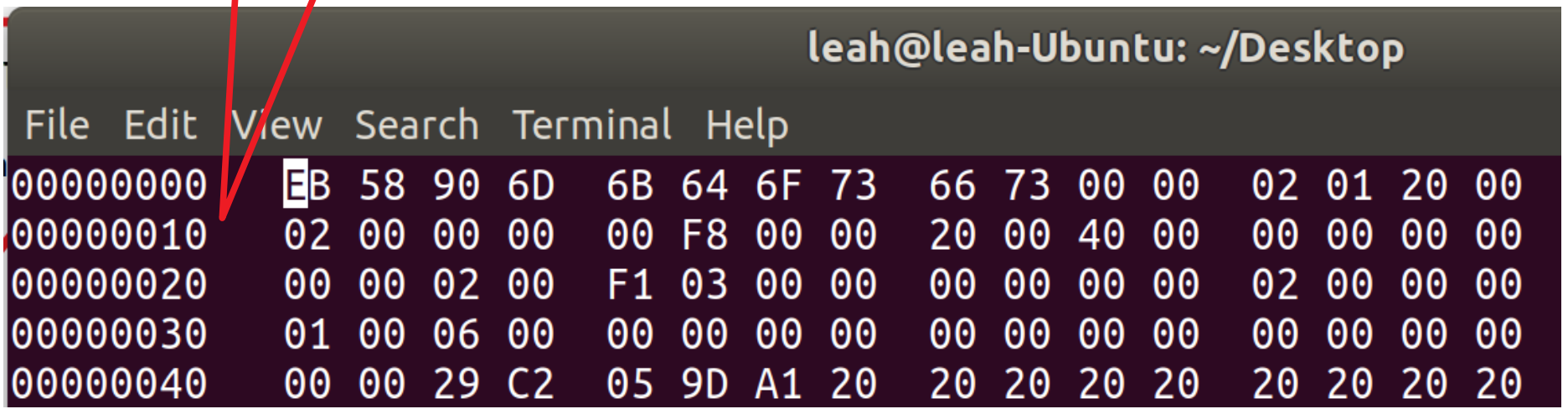
- Use cntrl-g to jump to a new line:



Hexedit

- To make things easier to read, set each line to 16 bytes

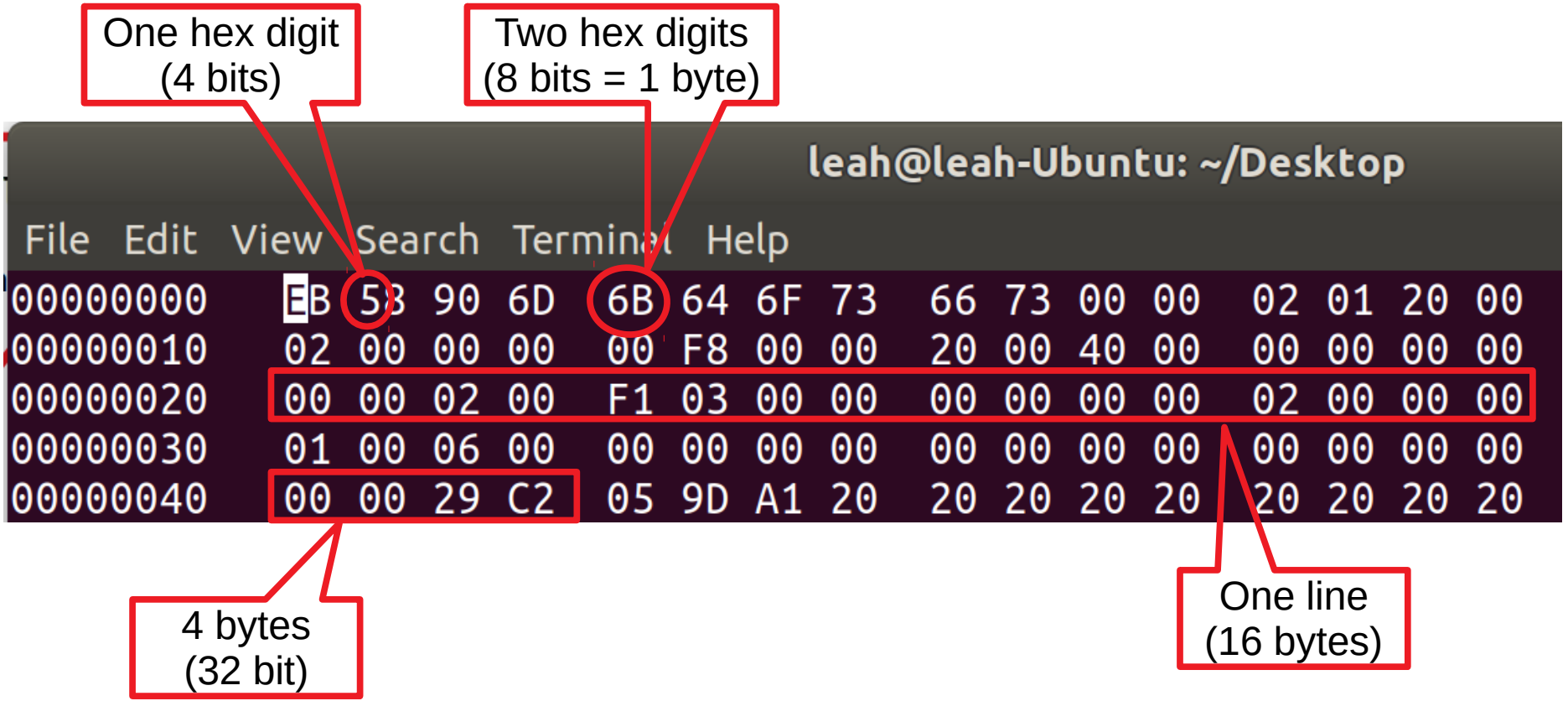
$10_{hex} = 16_{dec}$ bytes



Hexadecimal Refresher

- Represented by 0-9 A-F
- Base 16: each digit ranges from 0 – 15
- Binary 0000 to 1111 = 0 to 15
  - Need 4 bits to represent one hexadecimal digit
  - 2 hexadecimal digits = 8 bits = 1 byte

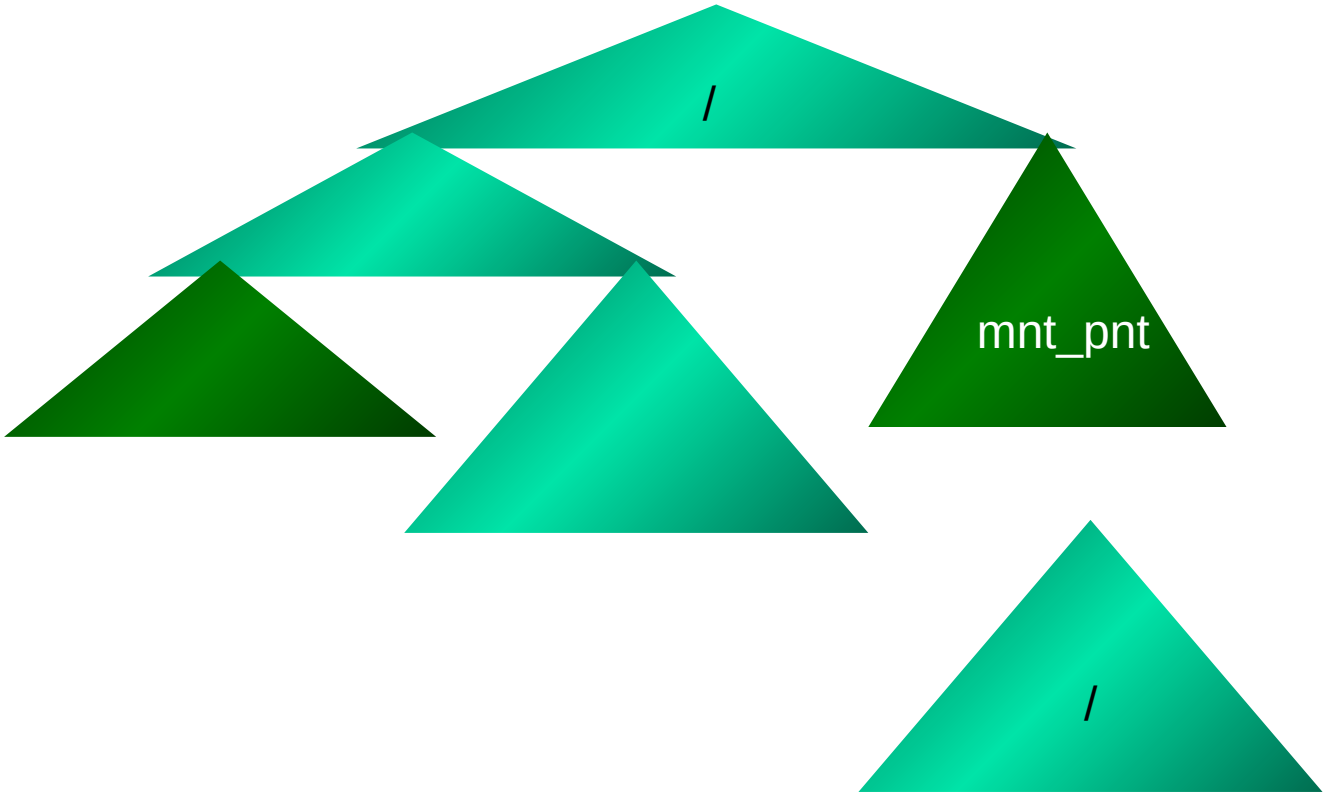
Hexedit


One hex digit
(4 bits)
Two hex digits
(8 bits = 1 byte)
leah@leah-Ubuntu: ~/Desktop
File Edit View Search Terminal Help
00000000 EB 53 90 6D 6B 64 6F 73 66 73 00 00 02 01 20 00
00000010 02 00 00 00 00 F8 00 00 20 00 40 00 00 00 00 00
00000020 00 00 02 00 F1 03 00 00 00 00 00 00 02 00 00 00
00000030 01 00 06 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 29 C2 05 9D A1 20 20 20 20 20 20 20 20 20
4 bytes
(32 bit)
One line
(16 bytes)

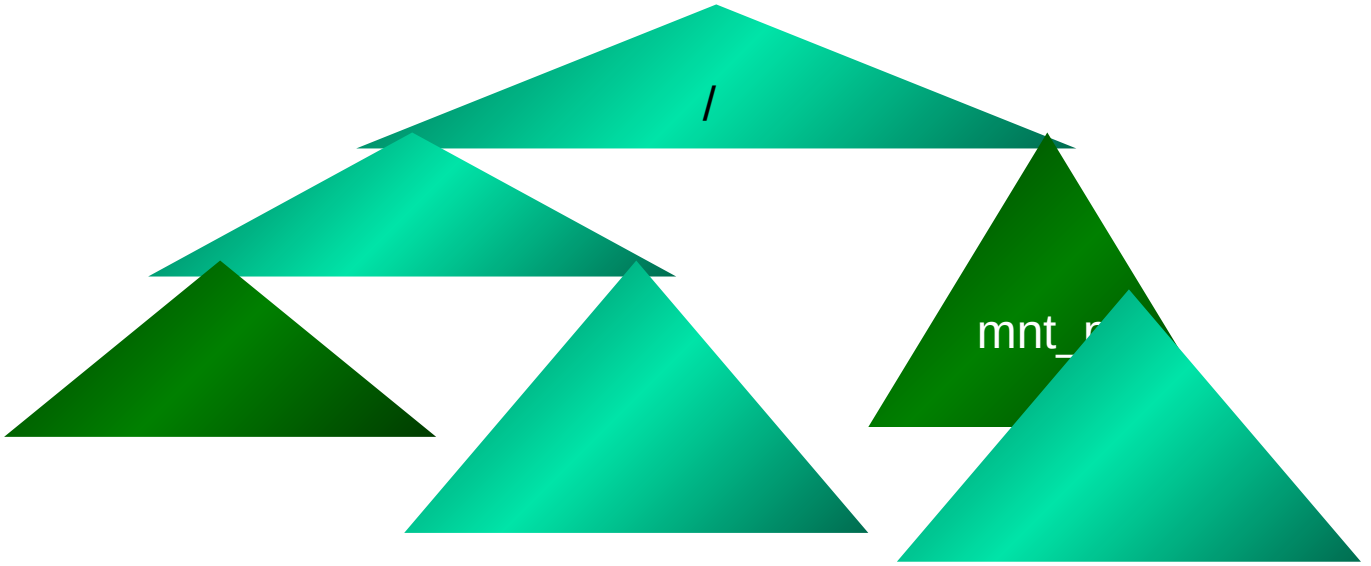
Mount

- In a Linux environment:
  - mkdir mnt_pnt
  - sudo mount -o loop /path/to/image mnt_pnt
  - cd mnt_pnt
- Now you can access the contents of the image file through the mount point
- To remove:
  - sudo umount /path/to/image

Mount



Mount



Terms

- **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 **F**ile **A**llocation **T**able and is a map of files to data
- A set of bytes make up a sector and a set of sectors make up a cluster, FAT entries are in terms of clusters

FAT32 Disk Layout

- **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

Reserved Region

- Contains information about the file system itself
- When a FAT file system is mounted, the machine reads the reserved region to determine the type of file system and the characteristics associated with the file system
- For example,
  - Size of clusters
  - Number of FATs
  - etc

FAT

- Data is organized into clusters
- Each cluster in the data region is allocated through the File Allocation Table (FAT)
- The FAT coordinates which clusters each file/directory uses
- To access a file, must know its first, starting cluster
- Then, can follow find subsequent clusters using the FAT

FAT

xxxxxxxx	xxxxxxxx	00000009	00000004
00000005	00000007	00000000	00000008
FFFFFFFF	0000000A	0000000B	00000011
0000000D	0000000E	FFFFFFFF	00000010
00000012	FFFFFFFF	00000013	00000014
00000015	00000016	FFFFFFFF	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Root Directory:
2, 9, A, B, 11

https://www.pjrc.com/tech/8051/ide/fat32.html

FAT

xxxxxxxx	xxxxxxxx	00000009	00000004
00000005	00000007	00000000	00000008
FFFFFFFF	0000000A	0000000B	00000011
0000000D	0000000E	FFFFFFFF	00000010
00000012	FFFFFFFF	00000013	00000014
00000015	00000016	FFFFFFFF	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Root Directory:
2, 9, A, B, 11

File #1:
3, 4, 5, 7, 8

File #2:
C, D, E

File #3:
F, 10, 12, 13, 14, 15, 16

https://www.pjrc.com/tech/8051/ide/fat32.html

Data Region

- Data in the data region can be of two types: directory (folder) data or file data

Data Region

Directory data:

- Each directory contains directory (DIR) entries
- A directory's data in the data region is a list of DIR entries where each DIR entry represents an item in the directory (ie files and/or other directories)

Data Region

Directory data:

- For example,
  - Directory my_dir contains files a.txt and b.txt and directory my_child_dir
  - The portion of the data region corresponding to my_dir would contain DIR entries for a.txt, b.txt, and my_child_dir
- Therefore, ***every*** file and directory will have a DIR entry (except for the root directory)

Data Region

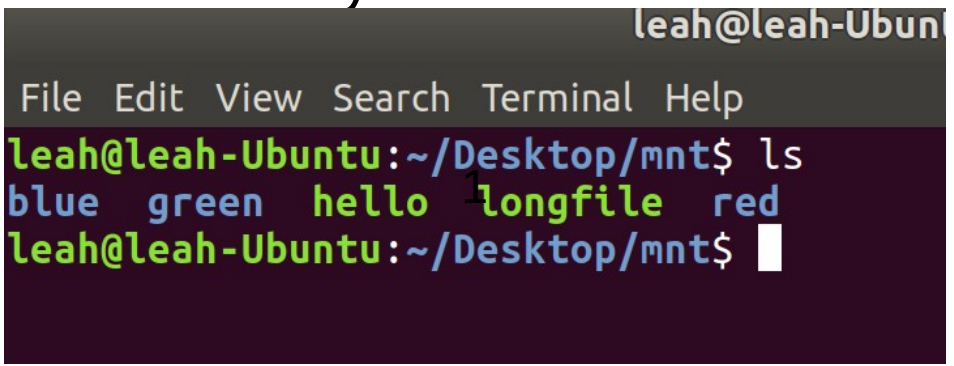
Directory data:

- DIR entries
  - Fixed sized data structures which contain fixed sized fields containing information about the entry
    - Name
    - Size
    - ***First cluster number***
  - Check FATspec for all fields
  - Read in using a struct with corresponding fields

Data Region

Directory data:

- Example: Root Directory



- In the data region for the root directory, there will be DIR entries for blue, green, and red folders, and hello, longfile files

Data Region

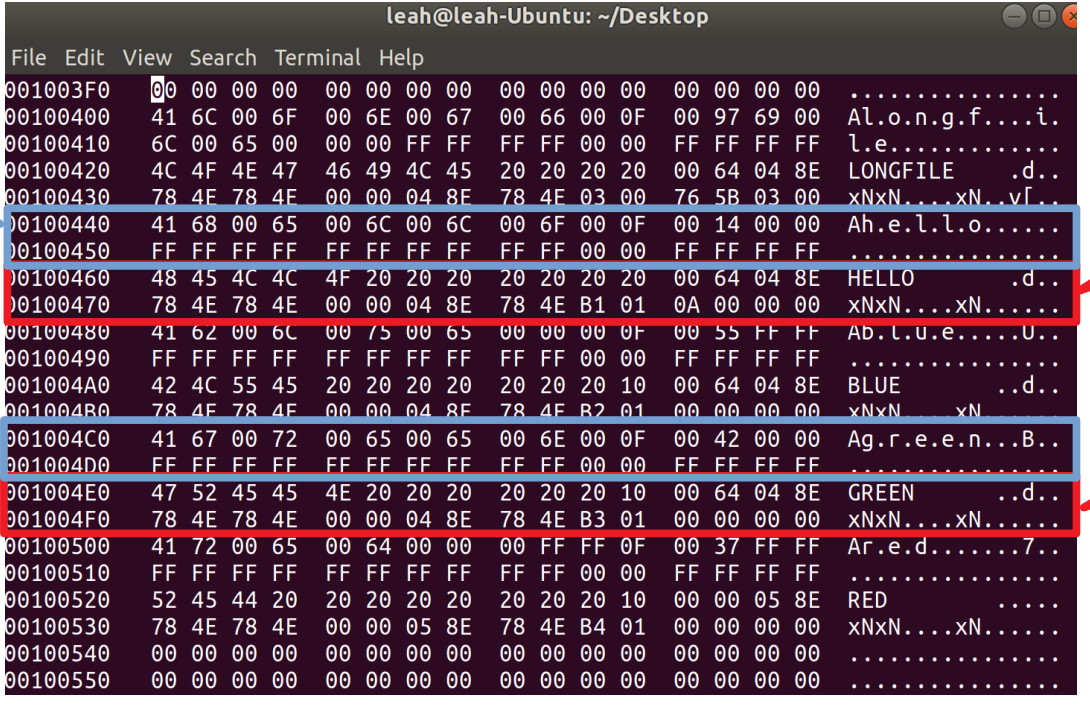
Directory data:

- All DIR entries (except for . and ..) will be preceeded by a single LDIR entry
- All directories will have a “.” and “..” DIR entry except for the root directory

Data Region

Directory data:

- Example: Root Directory



Corresponding long dir name entries (safe to ignore)

DIR entry for hello

DIR entry for green directory

Data Region

File data:

- In the data region, file data is simply the content of the file
- If the file is a text file, clusters in the data region allocated to the text file is simply the text contents

Data Region

File data example

- longfile contents:
  - this is a loooong file
  - this is a loooong file
  - this is a loooong file
  - this is a loooong file
  - this is a loooong file
  - ...

```
00104DE0   00 00 00 00   00 00 00 00   00 00 00 00   00 00 00 00  ................
00104DF0   00 00 00 00   00 00 00 00   00 00 00 00   00 00 00 00  ................
00104E00   74 68 69 73   20 69 73 20   61 20 6C 6F   6F 6F 6F 6E  this is a loooon
00104E10   67 20 66 69   6C 65 0A 74   68 69 73 20   69 73 20 61  g file.this is a
00104E20   20 6C 6F 6F   6F 6F 6E 67   20 66 69 6C   65 0A 74 68   loooong file.th
00104E30   69 73 20 69   73 20 61 20   6C 6F 6F 6F   6F 6E 67 20  is is a loooong
00104E40   66 69 6C 65   0A 74 68 69   73 20 69 73   20 61 20 6C  file.this is a l
00104E50   6F 6F 6F 6F   6E 67 20 66   69 6C 65 0A   74 68 69 73  oooong file.this
00104E60   20 69 73 20   61 20 6C 6F   6F 6F 6F 6E   67 20 66 69   is a loooong fi
00104E70   6C 65 0A 74   68 69 73 20   69 73 20 61   20 6C 6F 6F  le.this is a loo
00104E80   6F 6F 6E 67   20 66 69 6C   65 0A 74 68   69 73 20 69  oong file.this i
00104E90   73 20 61 20   6C 6F 6F 6F   6F 6E 67 20   66 69 6C 65  s a loooong file
00104EA0   0A 74 68 69   73 20 69 73   20 61 20 6C   6F 6F 6F 6F  .this is a loooo
00104EB0   6E 67 20 66   69 6C 65 0A   74 68 69 73   20 69 73 20  ng file.this is
00104EC0   61 20 6C 6F   6F 6F 6F 6E   67 20 66 69   6C 65 0A 74  a loooong file.t
00104ED0   68 69 73 20   69 73 20 61   20 6C 6F 6F   6F 6F 6E 67  his is a loooong
00104EE0   20 66 69 6C   65 0A 74 68   69 73 20 69   73 20 61 20   file.this is a
00104EF0   6C 6F 6F 6F   6F 6E 67 20   66 69 6C 65   0A 74 68 69  loooong file.thi
00104F00   73 20 69 73   20 61 20 6C   6F 6F 6F 6F   6E 67 20 66  s is a loooong f
00104F10   69 6C 65 0A   74 68 69 73   20 69 73 20   61 20 6C 6F  ile.this is a lo
```

Function: exit

- Close program and free up any allocated resources

Function: info

- 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

Function: info

- Read all FAT32 fields into a struct – you will need these values throughout your program
- Within your struct, make a corresponding variable for each field
  - Make sure the byte size of the variable matches the size of the field as specified in the FATspec!
  - Make sure the struct variables are listed in the same order as shown in the FATspec
- Info function prints these fields
- Check your values make sense
  - watch out for endian issues!

Endianness

- Endianess describes the order in which *bytes* are stored in memory
- One hex value represents 16 bits
  - Two hex values represent a byte
- Given an integer 0x0A0B0C
  - Big endian systems would represent integer as 0A | 0B | 0C
    - Most significant byte first
  - Little endian systems would represent integer as 0C | 0B | 0A
    - Least significant byte first
- FAT32 uses little endian
- (characters are single bytes → no reordering necessary)

Endianness

- Reading in all structs using __attribute__((packed))
  - Look at documentation for syntax
  - This often resolves endianness issues
  - Whenever you read in a new type of value from FAT, make sure the endianness is correct

Function: info

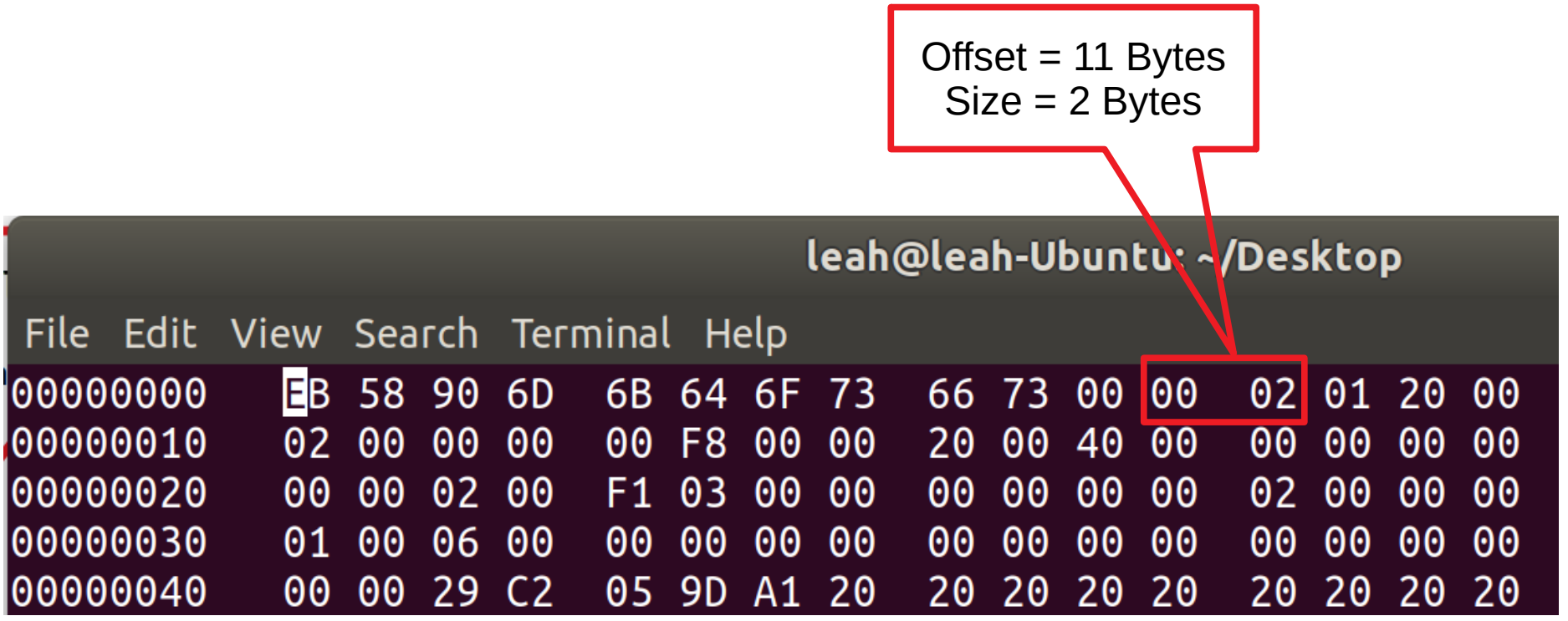
ex) Finding the BPB_BytesPerSec

FATspec

- Field Name: BPB_BytesPerSec
- Offset (byte): 11
- Size (byte): 2

Boot Sector and BPB Structure

Name	Offset (byte)	Size (bytes)	Description
BS_jmpBoot	0	3	Jump instruction to boot code. This field has two allowed forms: **jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90** and **jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??** **0x??** indicates that any 8-bit value is allowed in that byte. What this forms is a three-byte Intel x86 unconditional branch (jump) instruction that jumps to the start of the operating system bootstrap code. This code typically occupies the rest of sector 0 of the volume following the BPB and possibly other sectors. Either of these forms is acceptable. **JmpBoot[0] = 0xEB** is the more frequently used format.
BS_OEMName	3	8	"MSWIN4.1" There are many misconceptions about this field. It is only a name string. Microsoft operating systems don't pay any attention to this field. Some FAT drivers do. This is the reason that the indicated string, "MSWIN4.1", is the recommended setting, because it is the setting least likely to cause compatibility problems. If you want to put something else in here, that is your option, but the result may be that some FAT drivers might not recognize the volume. Typically this is some indication of what system formatted the volume.
BPB_BytsPerSec	11	2	Count of bytes per sector. This value may take on only the following values: 512, 1024, 2048 or 4096. If maximum compatibility with old implementations is desired, only the value 512 should be used. There is a lot of FAT code in the world that is basically "hard wired" to 512 bytes per sector and doesn't bother to check this field to make sure it is 512. Microsoft operating systems will properly support 1024, 2048, and 4096. **Note:** Do not misinterpret these statements about maximum compatibility. If the media being recorded has a physical sector size N, you must use N and this must still be less than or equal to 4096. Maximum compatibility is achieved by only using media with specific sector sizes.



- 00 02 little endian = 02 00 big endian
- $0200_{hex} = 512_{dec} \rightarrow$ 512 bytes per sector

Function: ls DIRNAME

- First, need to find the desired directory (DIRNAME) via file system traversal
- (directories are just a special type of file with DIR_attr = 0x10)

Function: info

- Note: one of the fields in the boot block is the BPB_RootClus
  - This field tells you the cluster number of the root directory
  - This field is usually = 2

ls DIRNAME

- Let’s first try running ls at the root
- Start at root directory – cluster number given in the boot sector
- Index into the FAT using this cluster number
- Value at this location in FAT table will either contain an end of cluster marker or a new index
- FAT entries form a chain of indices for a given file

ls DIRNAME

- Running ls on other directories is the exact same process except the starting cluster will be different
- If no argument given, use current working directory’s cluster
- If DIRNAME given, start at the first cluster number of DIRNAME (given in DIRNAME’s DIR entry)

FAT Entries

- FAT[current_cluster_number] = next_cluster_number → file continues at next_cluster_number
- FAT[current_cluster_number] = 0x0FFFFFF8 or 0x0FFFFFFF → end of cluster marker, current_cluster_number is the last cluster in the file
- FAT[current_cluster_number] = 0 → you are at an empty cluster

FAT

xxxxxxxx	xxxxxxxx	00000009	00000004
00000005	00000007	00000000	00000008
FFFFFFFF	0000000A	0000000B	00000011
0000000D	0000000E	FFFFFFFF	00000010
00000012	FFFFFFFF	00000013	00000014
00000015	00000016	FFFFFFFF	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Root Directory:
2, 9, A, B, 11

File #1:
3, 4, 5, 7, 8

File #2:
C, D, E

File #3:
F, 10, 12, 13, 14, 15, 16

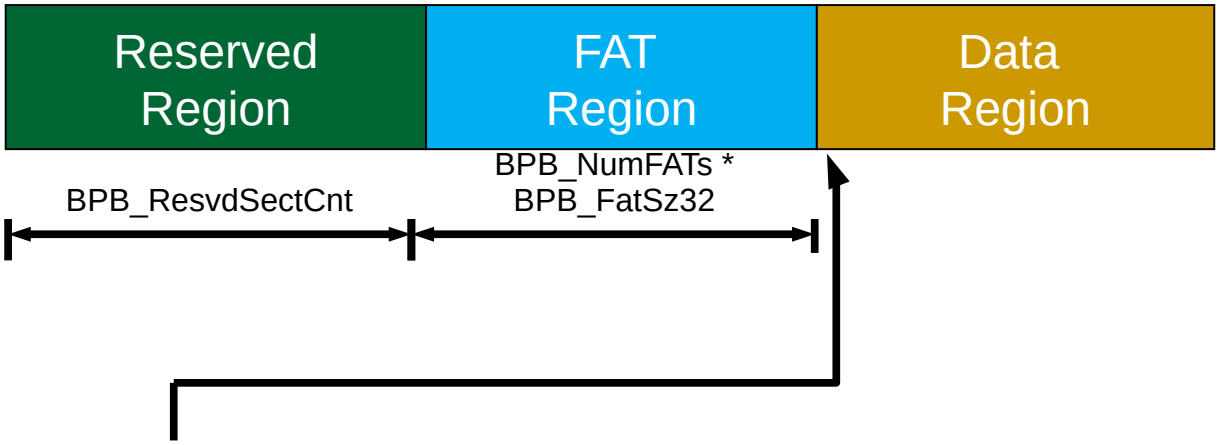
- So, the root directory contents are located in *cluster* numbers 2, 9, A, B, and 11

https://www.pjrc.com/tech/8051/ide/fat32.html

File System Traversal

- FAT table shows *cluster number* of file contents!
- Must go to that cluster number in the data region to find the file’s actual contents

Finding a File's Contents



- FirstDataSector = BPB_ResvdSectCnt + (BPB_NumFATs*BPB_FatSz32)
- FirstDataSector is the Sector where data region starts (should be something around 100400 in hex)

Finding a File's Contents

- FirstSectorofCluster =

  FirstDataSector + ((N -2)*BPB_SecPerClus)

  Start of Data Region

  This term is equal to the offset of the data from the beginning of the data region

  Sectors per cluster * number of clusters = offset in terms of sector

  (use cluster number – 2 because we start indexing clusters with N=2, remember how root started at 2?)

FAT32 Directories

- Similar to Boot Sector, make a struct with the fields corresponding to the entries in the directory structure
  - Check FATspec for details
- You do not have to support long entries → directory consists of second long directory entry and short entry
  - But they may exist in the image file, you can safely skip over them

Function: cd DIRNAME

- Similar to ls, but instead of printing filenames, compare filenames to DIRNAME
- If DIRNAME is found and DIR_attr = 0x10, construct DIRNAME's location using DIR_FstClusHI and DIR_FstClusLO and set it as the present working directory (simply store it in your program, do not change environment variables)

Functions: size

- Same as cd except you are looking for DIR_Size upon match of filename

When in doubt, check the FATspec!