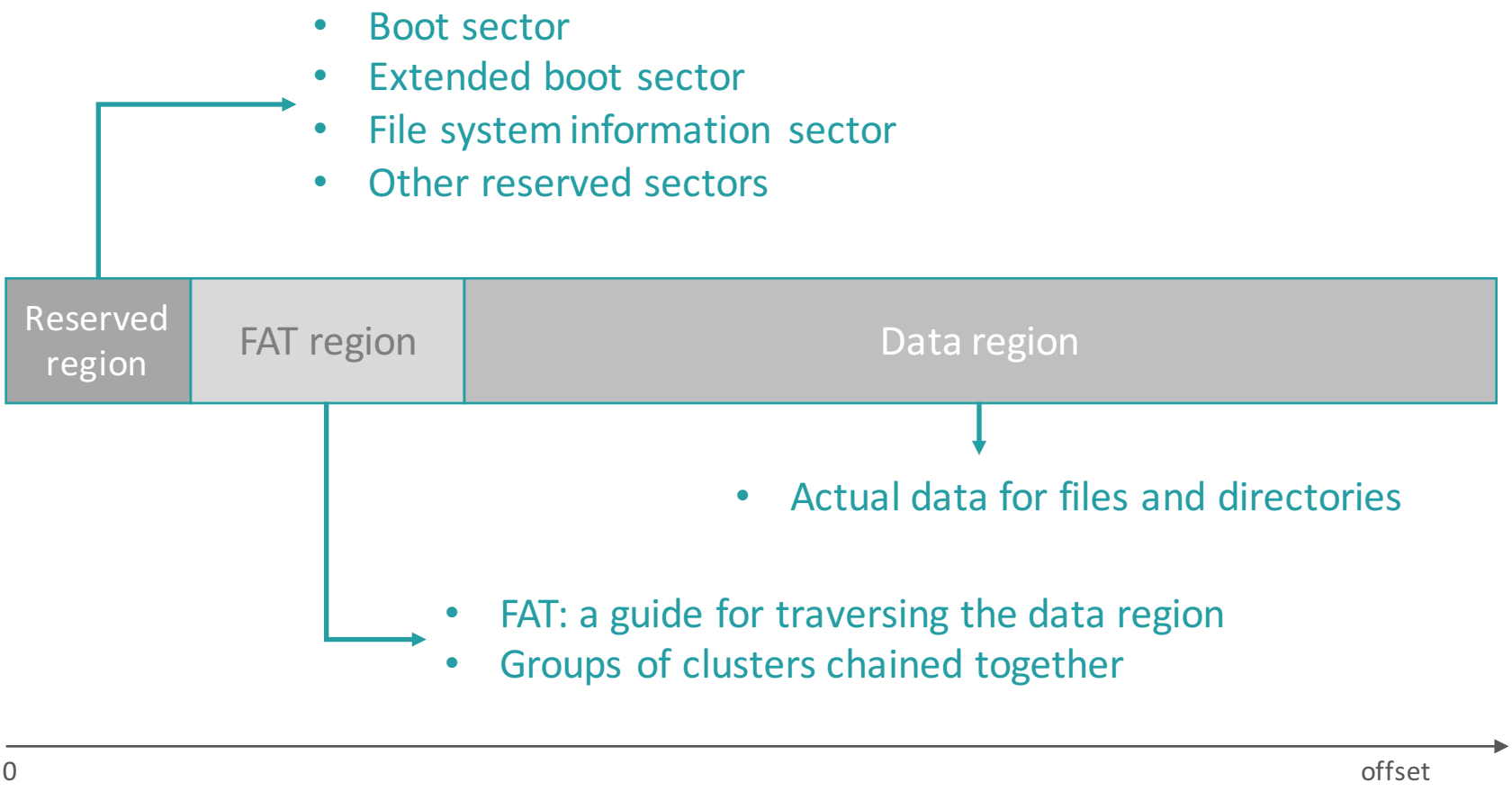
COP4610
Introduction to Operating Systems
Project #3: Implementing a FAT32 File System

Outline

- Review
- C functions, data types and program structure
- Reserved region: BPB
- Accessing data
  - File contents
  - Directories
- Program commands
  - `exit`
  - `info`
  - `ls DIRNAME`

Review

Review


Boot sector
Extended boot sector
File system information sector
Other reserved sectors
Reserved region
FAT region
Data region
Actual data for files and directories
FAT: a guide for traversing the data region
Groups of clusters chained together
0
offset

Useful C functions

- `int **open**(const char *path, int oflag, ...);`
- `**lseek**(int fildes, off_t offset, int whence);`
- `ssize_t **read**(int fildes, void *buf, size_t nbyte);`
- `ssize_t **write**(int fildes, const void *buf, size_t nbyte);`
- `int **close**(int fildes);`

Useful C data types

- `unsigned char`
  - 8 bits without sign
- `unsigned short`
  - 16 bits without sign
- `unsigned int`
  - 32 bits without sign

- Be sure to use **unsigned** types when working with the volume!

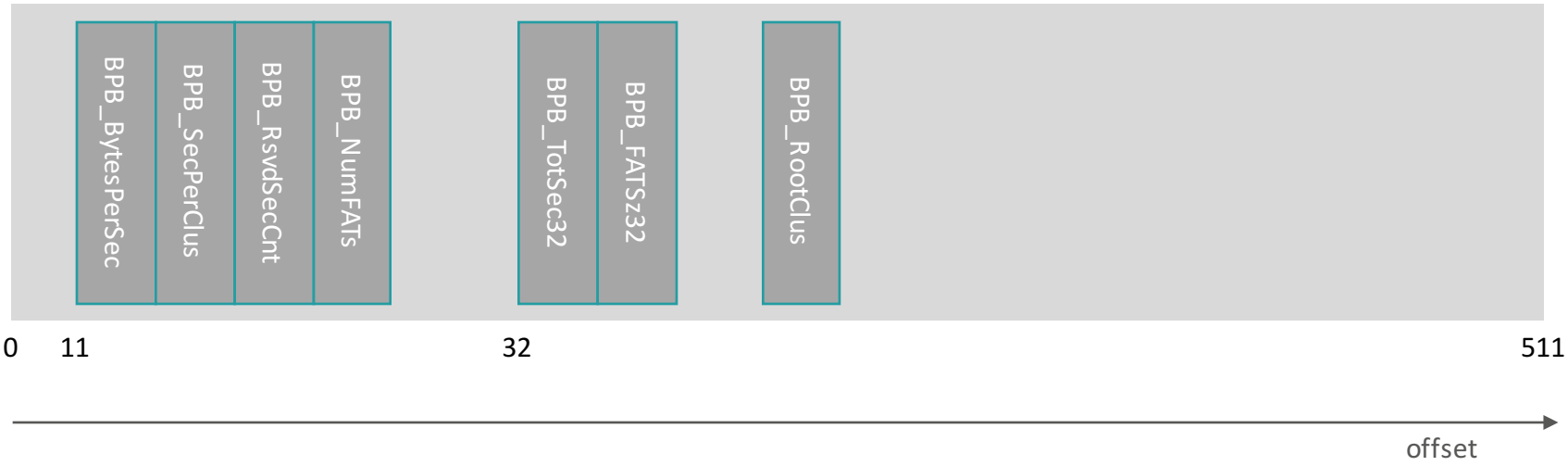
Program structure

```
while (command != “exit”) {
      print(“$ “);
      read_from_stdin(command);
      if (command == “info”) {
            do_something();
      }
      else if (command == “ls”) {
            do_something2();
      }
      ...
}
```

BPB (BIOS Parameter Block)

- Located at the first sector of the volume
- Part of the Reserved Region (first sector(s))
- Contains important information about the file system
  - Bytes per sector (`BPB_BytsPerSec`)
  - Sectors per cluster (`BPB_SecPerClus`)
  - Reserved region size (`BPB_RsvdSecCnt`)
  - Number of FATs (`BPB_NumFATs`)
  - FAT size (`BPB_FATSz32`)
  - Root cluster (`BPB_RootClus`)
  - Total sectors (`BPB_TotSec32`)

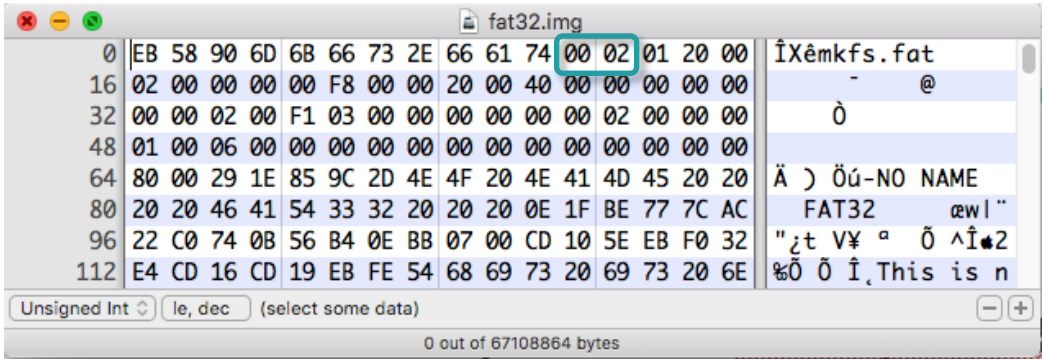
BPB (BIOS Parameter Block)



- Refer to FAT32 specification for details

BPB (BIOS Parameter Block)

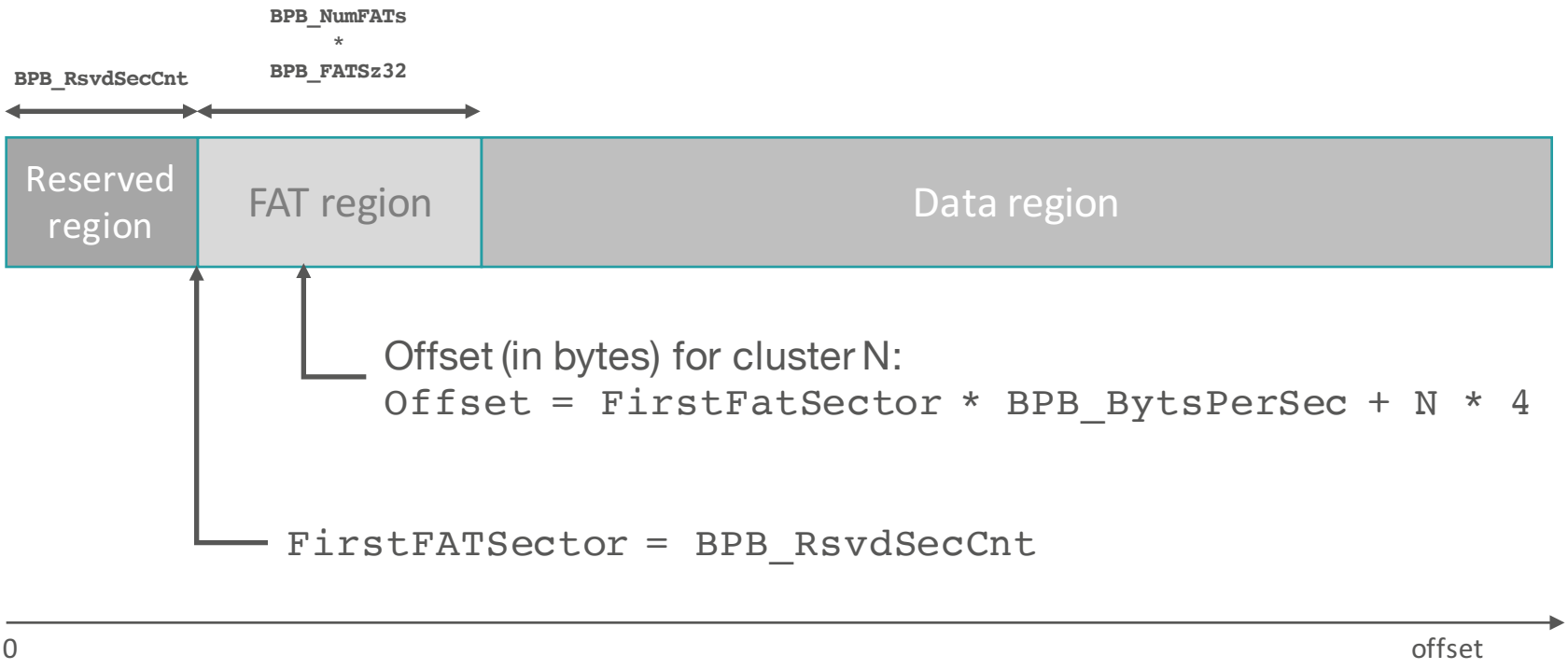
- Example: Bytes per sector
  - Field in FAT32 spec: BPB_BytsPerSec
  - Offset: 11 bytes
  - Size: 2 bytes



  - Remember: Little endian
    - 00 02 → 0x0200 = 512 bytes per sector

Traversing the FAT

- Find FAT's offset, then find offset within FAT for a given cluster

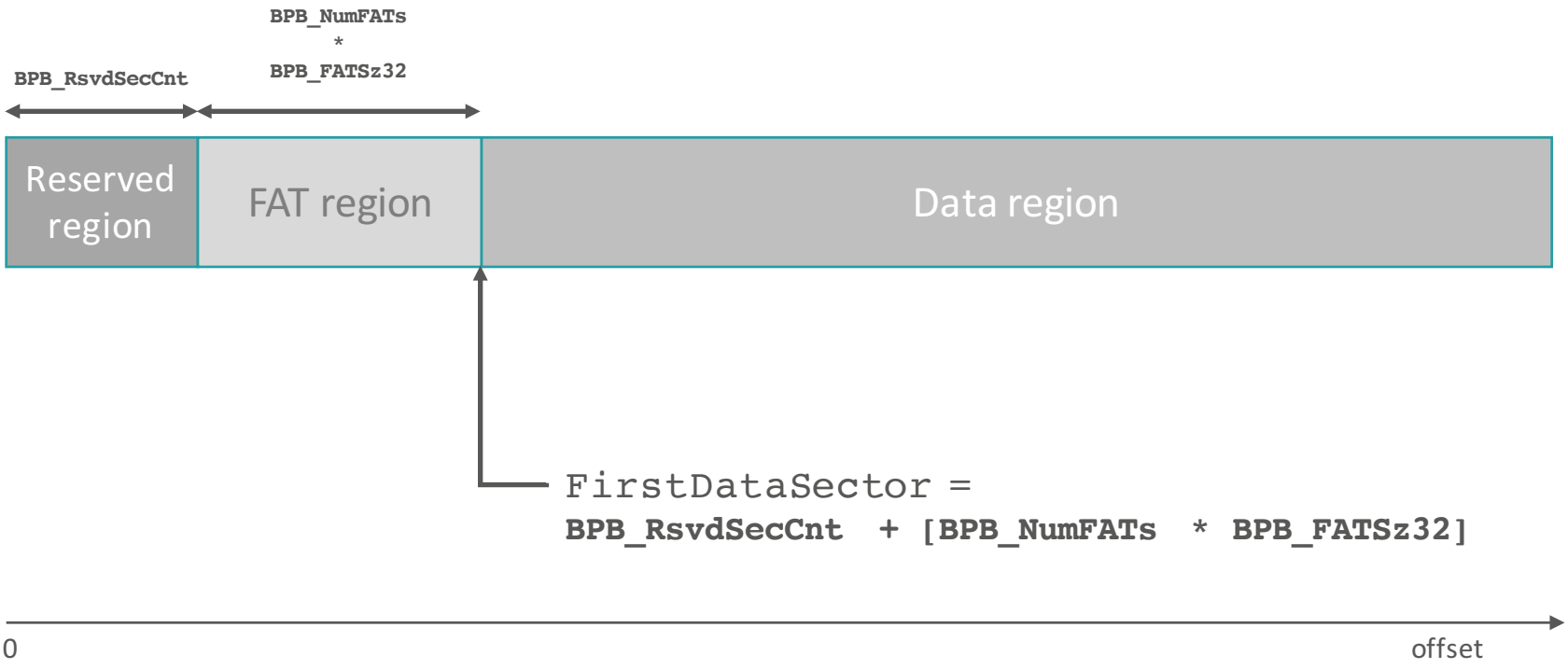


Traversing the FAT

- Get the offset of the FAT (in bytes)
- For a given cluster, find its entry
  - Find the cluster's offset
  - Read the entry (4 bytes) at that offset
  - If the entry is 0x0FFFFFF8 to 0x0FFFFFFE (or 0xFFFFFFFF), it means it is the last cluster. Stop
  - If not, the read entry is the next cluster. Repeat.

Accessing File Contents

- Find the first data sector



Accessing File Contents

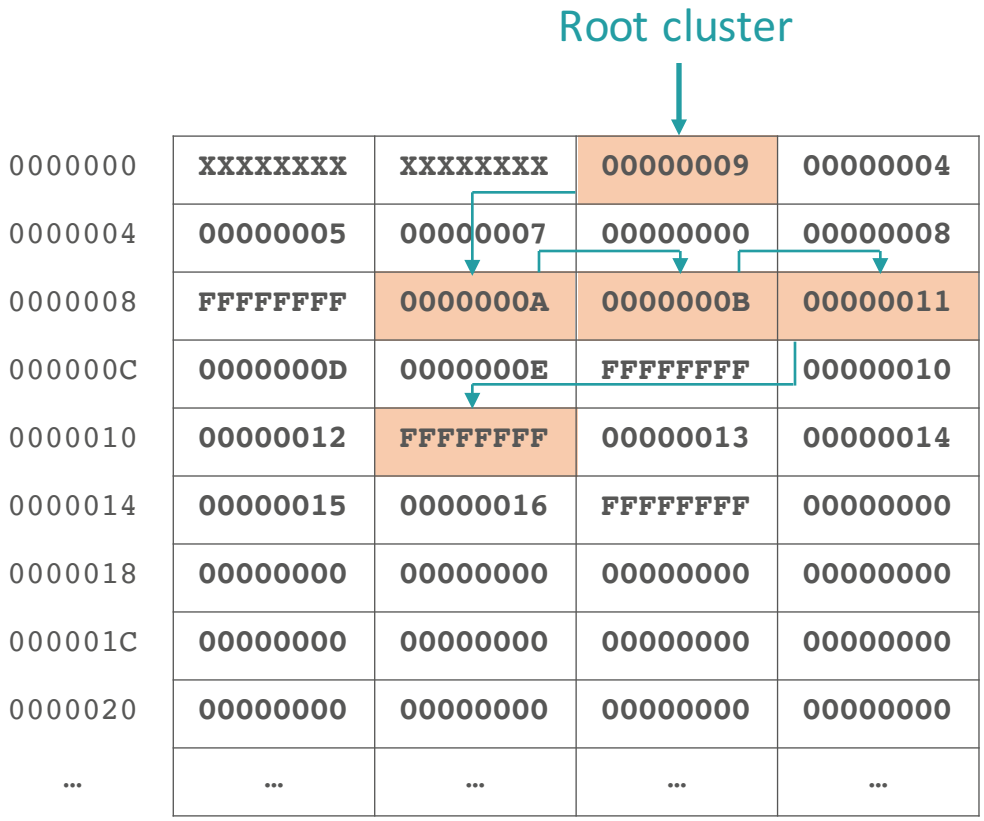
- Then find the offset for a given cluster N (within the data region)
  - Intuition: ~~`N * BPB_SecPerClus`~~
    - But the first cluster was 2!
    - And the data begins at the beginning of the data region
  - → **`(N – 2) * BPB_SecPerClus`**
- Now combine with the first data region
  - We get the offset (sector) of cluster N within the volume, given its number:

`Offset = FirstDataSector + (N – 2) * BPB_SecPerClus`

Accessing File Contents

- Steps
  - Get the offset for the first cluster (previous slide)
  - Calculate the byte offset
    - `ByteOffset = Offset * BPB_BytsPerSec`
  - Read/write data at that offset within the image file
  - Traverse FAT when needed (i.e.: you reached the end of the cluster and need to read more data)
  - Don't go beyond the file size
  - Don't go beyond cluster size without traversing
    - Clusters will (most likely) not be one after each other

Accessing File Contents




```
while not_last_cluster():
    read/write_current();
    get_next_cluster();
end while;
```

FAT entry indicates where the next cluster is located

- If 0x0: cluster not allocated
- If 0xFFFFFFFF (or 0x0FFFFFF8 to 0x0FFFFFFE): last cluster
- Otherwise: indicates next cluster of the file/directory

Accessing Directories

- Directories are also considered files
- Content is directory entries (DIRENTRY)
  - Fixed size structure (32B) that contains information about objects in the directory (files, subdirectories)
  - See section 6 of FAT32 spec
- How to get info about the directory entries
  - Find directory's first cluster (and its offset)
  - Use a structure to read the data at the offset
    - Does `read()` sound familiar?
  - Traverse the FAT when needed

Accessing Directories

DIRENTRY

```
struct DIRENTRY {
    unsigned char DIR_name[11];
    unsigned char DIR_Attributes;
    ...
} __attribute__((packed));
```

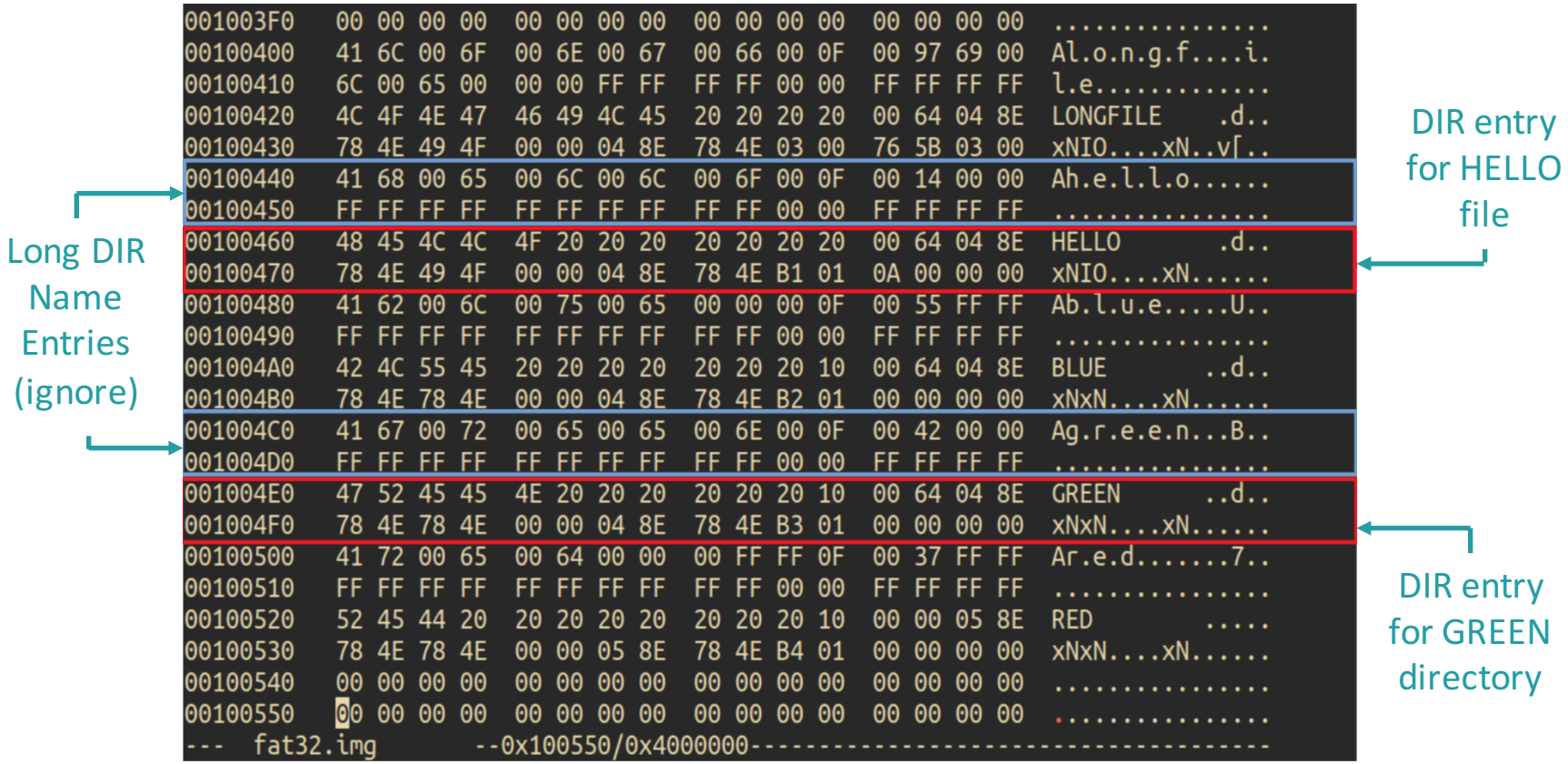
- A `struct DIRENTRY*` can be used as parameter for `read()` and `write()`
- Be sure to use `__attribute__((packed))` in order to avoid attribute padding

Accessing Directories

- Not all read directory entries are actual directory entries
  - Some are empty entries
    - Check first byte (`DIR_name[0]`)
      - 0x0: empty entry and no other entries after it
      - 0xE5: empty entry but more entries after it
      - Otherwise: entry is taken
  - Some of them are long DIR name entries
    - Can be ignored
    - Use `ATTR_LONG_NAME` mask on `DIR_Attributes` to determine whether it is a DIRENTRY or long DIR name
      - Check FAT32 specs

Accessing Directories

- DIR entries



Program Commands

`exit`

- Releases all allocated resources
- Safely exits the program

Program Commands

`info`

- Prints important data about the file system
- Read from BPB
- Two options
  - Move to the offset of the volume and read values individually
  - Create a structure that represents the BPB and read by passing the whole structure

Program Commands

`info`

- Be careful!
  - If you get the wrong data from the BPB you might not be able to continue until you fix it
    - All the other commands depend on this data
- Do the values make sense?
  - verify by using Hexedit and the FAT32 spec

Program Commands

`ls DIRNAME`

- Show the names of the files/directories under the directory given by DIRNAME (local)
- Read the current directory's entries until you find DIRNAME's (and its first cluster!)
  - Then read DIRNAME's entries and print their names
- If no argument given, print the names of the entries at the current directory
  - Read entries for current directory's cluster

Program Commands

`ls DIRNAME`

- Hint: Once current directory's entries were read, find DIRNAME's cluster within current dir
  - And repeat the process
- If DIRNAME does not exist, print an error

Questions?