CSC 415 Operating Systems
Section 02
10/15/2023

Team Name:
Byte Builders

Team Members:

Abdarrahman Ayyaz	922859909
Avitaj Singh Bhangoo	921303783
Mohammed Deeb	922875899
Miguel Antonio Logarta	923062683

Github Username:
Miguel-Antonio-Logarta

GitHub Link:
https://github.com/CSC415-2023-Fall/csc415-filesystem-Miguel-Antonio-Logarta

Structure:
This is an overview of our volume partition.

Boot sector 512 bytes	**Volume Control Block (VCB)** 512 bytes	**File Allocation Table (FAT)**	**Data Region** (Contains file and subdirectories)

Boot Sector:
This is the boot sector of the drive. LBAread cannot access it because it precedes block 0.

```
000000: 43 53 43 2D 34 31 35 20  2D 20 4F 70 65 72 61 74 | CSC-415 - Operat
000010: 69 6E 67 20 53 79 73 74  65 6D 73 20 46 69 6C 65 | ing Systems File
000020: 20 53 79 73 74 65 6D 20  50 61 72 74 69 74 69 6F |  System Partitio
000030: 6E 20 48 65 61 64 65 72  0A 0A 00 00 00 00 00 00 | n Header........
000040: 42 20 74 72 65 62 6F 52  00 96 98 00 00 00 00 00 | B treboR.��.....
000050: 00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | ........KL......
000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000070: 52 6F 62 65 72 74 20 42  55 6E 74 69 74 6C 65 64 | Robert BUntitled
000080: 0A 0A 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
0000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

- One of the first issues regarding the first milestone was knowing what to do with this part of the volume. This was in our sample volume when we first opened it. Because some of the bytes in the beginning of this volume gave us details like the block size and number of blocks, we initially thought that this was the volume control block. After a lot of head scratches, we finally figured out that this was actually supposed to be the boot sector of the volume. The boot sector is usually at the first block of an unpartitioned disk.
- The total size of the boot sector takes up 512 bytes.
- The first 64 bytes contains a string which is the header of the boot sector. It reads "**CSC-415 - Operating Systems File System Partition Header\n\n**"
- At offset 0x40, we get an 8-byte (uint64_t) that contains our magic number. It is a unique special number used to identify the volume. Since the bytes are stored in little endian, the hexadecimal value is read "**42 20 74 72 65 62 6F 52**" as **0x526F626572742042.** Which holds the value 5940074621376406000. Which is the same value as our PART_SIGNATURE macro.
- **00 96 98 00 00 00 00 00** holds the value 9999872. This is the size of the SampleVolume volume in bytes.
- **00 02 00 00 00 00 00 00** holds the value 512, which is our block size
- **4B 4C 00 00 00 00 00 00 holds the value** 19531, which is the total number of blocks in our volume.

Volume Control Block (VCB)

```
000200: 6C 69 75 42 65 74 78 00  00 96 98 00 00 00 00 00 | liuBetx..▯▯.....
000210: 00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | ........KL......
000220: 01 00 00 00 00 00 00 00  99 00 00 00 00 00 00 00 | ........▯.......
000230: 9A 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00 | ▯...............
000240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000270: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000310: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000320: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000330: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000340: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000350: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000360: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000370: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000380: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000390: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

- This is our volume control block. It's stored right after the boot sector of the volume. It is located at lba block 0. This is the lowest block that LBAwrite and LBAread can access.
- For our file system, we decided to go with a FAT file system.

- Our VCB was intended to only be 64 bytes wide. However, since our minimum block size is 512 bytes, our VCB can be expanded to fill that many bytes. Maybe in the future, we can utilize all that extra space.
- **6C 69 75 42 65 74 78 00** is our signature for the volume control block. It is a unique number to identify the partition
- **00 96 98 00 00 00 00 00** is our volume size
- **00 02 00 00 00 00 00 00** is our block size
- **4B 4C 00 00 00 00 00 00** is our number of blocks
- **01 00 00 00 00 00 00 00** is the position of our first FAT block. Notice how it says 01. Our FAT starts at lba block 1, which is after the VCB.
- **99 00 00 00 00 00 00 00** is the length of our FAT. It is 153. The calculation of this number will be covered in the next section.
- **9A 00 00 00 00 00 00 00** is the lba block position of our first directory entry block. The value of this integer is 154. It is the first block after the FAT.
- **01 00 00 00 00 00 00 00** is the length of the directory entry in blocks. The value is 1. Reasons as to why the value is 1 will be covered in the “Directory Entry” section.

File Allocation Table (FAT)

```
000400: 03 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000410: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000420: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000430: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000440: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000450: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000460: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000470: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000480: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000490: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
0004A0: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
0004B0: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
0004C0: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
0004D0: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
0004E0: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
0004F0: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
```

- For our file system, we went with the FAT file system to keep track of our free blocks instead of a simple bitmap.
- Every 4 bytes (32 bit) represents a FAT block. For the sake of clarity, I have set off every block to have the value of 1.
- The structure of our FAT block goes as follows,
    - The first bit is used to check if a block is in use
    - The second bit is used to indicate whether a block is the end of a file
    - The other 30 bit are used for addressing the next lba block of a file
- Notice how the first 4 bytes have the value of 03 00 00 00. This is because the first block has already been allocated for the system. The first block has already been taken up by our root directory entry.
- If we convert 03 00 00 00 to binary, we get the binary value, 11000000000000000000000000000000.
    - The first bit indicates our file is already in use.
    - The second bit indicates that this is the last block of our directory entry. This is because our directory entry's size is less than 512 bytes long, so it's stored in only 1 lba block.
    - The next 30 bits have been set to NULL. Since our second bit tells us that it is the end of the file, we don't require a pointer to the next lba block.

Why did we go with a FAT file system instead of a bitmap?

- Even though a free space bitmap was more memory efficient, we felt that it lacked a lot of features that we desired.

- FAT blocks take up 4 bytes instead of 1 bit, but now we get access to 31 extra bits that we can use to store information about a block.
- FAT blocks can now store an eof bit, an in use bit, and 30 extra bits to store pointers to the next FAT block of a file.

	Free space Bitmap	FAT32
Pros	● Memory efficient, every bit corresponds to a free block on the disk ● Simple to implement	● Bigger addressing ● Efficient use of storage space. Even the smallest gaps in memory can be filled up. ● You can keep track of free clusters / sectors, allowing for faster searches for free space
Cons	● Data becomes fragmented the more data we read and write to it, leading to a lot of wasted space	● Increased overhead ● Requires more memory storage than a free space bitmap ● Files and directories are treated the same way ● Fragmentation from the get-go due to its inherent design.

- The FAT file directory structure also stores files and directories in the same region. We differentiate between the two through the use of the is_directory bit in the struct.
- This design also allows us to represent our file system with a tree-like structure. A directory can contain more files and more directories. We can also have an unlimited amount of directories in theory.

```
013600: 01 2F 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ./..............
013610: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013620: 9A 00 00 00 00 00 00 00  C0 00 00 00 00 00 00 00 | .......@.......
013630: 51 5A 40 65 00 00 00 00  51 5A 40 65 00 00 00 00 | QZ@e....QZ@e....
013640: 01 2E 2E 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013660: 9A 00 00 00 00 00 00 00  C0 00 00 00 00 00 00 00 | .......@.......
013670: 51 5A 40 65 00 00 00 00  51 5A 40 65 00 00 00 00 | QZ@e....QZ@e....
013680: 01 2E 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0136A0: 9A 00 00 00 00 00 00 00  C0 00 00 00 00 00 00 00 | .......@.......
0136B0: 51 5A 40 65 00 00 00 00  51 5A 40 65 00 00 00 00 | QZ@e....QZ@e....
0136C0: 00 00 00 00 00 00 00 00  D1 05 02 00 00 00 00 00 | ........Ñ.......
0136D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0136E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0136F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

- The 01 indicates that our current block is directory/folder.
- The next 31 bytes is the name of our folder / file. In this case, since it is the root directory, it has a name of “/” (2F)
- The first directory takes a total of 64 bytes
- Why 64 bytes instead of a 32 bytes for a file directory?
    - We didn’t want to have to deal with a complicated way of storing time stamps in our directory entries.
    - Our time_t variables took up 8 bytes.
    - We also didn’t want to get stuck with really short file names, we allocated 31 bytes for filenames.
    - We used one byte for the is_directory bit. However, in the future, we can convert this unsigned char variable into a bitfield. We have 7 other bits to show file attributes
- Since this is a directory, we can nest directories inside of it. Inside the root directory, we have the “.” (2E) directory and the “..” (2E 2E) directory. The “.” is just a pointer to itself. The “..” directory is the parent directory. However since this is the root directory, it also just points to itself.
- `01` is the is_directory which hold the value of 1 (1 byte)
- `2F 00 00 00 00 00 00 …` is the 31 bytes name which holds the value ‘/’ (root directory)(3
- `9A 00 00 00 00 00 00 00` is the block_location (where the directory data will be stored)

- `C0 00 00 00 00 00 00 00` is the file_size which is set to sizeof(directory_entry) * 3
- `51 5A 40 65 00 00 00 00` is the date_created
- `51 5A 40 65 00 00 00 00` is the last_modified
- **The part in Pink** is the '..' of the root directory, it contains the directory entry of root but with the name being set to '..'
- **The part in purple** is the '.' of the root directory, it contains the directory entry of root but with the name being set to '.'

A table of who worked on which components:

Abdarrahman Ayyaz	Planning , debugging, VCB: initializing VCB and checking if its already initializing
Avitaj Singh Bhangoo	Planning, debugging, RootDE: initializing the root directory
Mohammed Deeb	Writing to the disk, volume management, debugging, making sure everything stored in the right place in the volume
Miguel Antonio Logarta	Free space management, Root directory debugging, and making sure free space works correctly and the root directory stores '..' and '.' correctly
Together	Putting the code together and doing the analysis

How did your team work together, how often you met, how did you meet, how did you divide up the tasks.

Due to the difference in our availability we had to meet separately (not the whole group) most of the time to help each other. But we were able to all meet twice to plan and to put the code together and do the analysis.

We divided the tasks where everyone is working on a concept they understand well. But when we put the code together every person presented their code to the others and made sure they understand it

A discussion of what issues you faced and how your team resolved them

Issue 1: offset mistakes. Mess calculation of where each of rootdir, freespace were store

Solution1: this issue was easy to resolve we had to read over the calculations again to use the Hex dump to make sure they were stored at the right spot