File System Milestone 1 - Team 42

GitHub name with group work: **nabware**

Name	**Student ID**
Nabeel Rana	924432311
Leigh Ann Apotheker	923514173
Ethan Zheng	922474550
Bryan Mendez	922744274

Nabeel Rana
Github: nabware

A dump (use the provided HexDump utility) of the volume file that shows the VCB, FreeSpace, and complete root directory.

The following hexadecimals are stored in little endian and must be converted to big endian by reversing the order for analysis.

VCB, Block 0
FAT table, Block 1 - 153
Root directory, Block 154 - 167

VCB, Block 0 (first 256 bytes)

```
000200: 2A 00 00 00 00 96 98 00  00 02 00 00 4B 4C 00 00 | *....��.....KL..
000210: 00 00 00 00 9A 00 00 00  01 00 00 00 99 00 00 00 | ....�.......�...
000220: A8 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | �...............
000230: 00 00 00 00 00 00 00 00  00 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 | ................
```

Bytes at address 000200 - 000203 highlighted in red represent the signature. When we convert the hexadecimal “00 00 00 2A” to decimal we get the signature, 42.

$(2 \times 16^1) + (10 \times 16^0) = 42$

Bytes at address 000204 - 000207 highlighted in orange represent the volume size. When we convert the hexadecimal “00 98 96 00” to decimal we get the volume size, 9,999,872.

$(9 \times 16^5) + (8 \times 16^4) + (9 \times 16^3) + (6 \times 16^2) = 9{,}999{,}872$

Bytes at address 000208 - 00020B highlighted in yellow represent the block size. When we convert the hexadecimal “00 00 02 00” to decimal we get the block size, 512.

$(2 \times 16^2) = 512$

Bytes at address 00020C - 00020F highlighted in green represent the number of blocks. When we convert the hexadecimal “00 00 4C 4B” to decimal we get the number of blocks, 19,531.

$(4 \times 16^3) + (12 \times 16^2) + (4 \times 16^1) + (11 \times 16^0) = 19{,}531$

Bytes at address 000210 - 000213 highlighted in cyan represent the block number where the file system metadata is. When we convert the hexadecimal “00 00 00 00” to decimal we get fsBlockNum, 0, or block 0 (the current block).
$(0 \text{ x } 16^0) = 0$

Bytes at address 000214-000217 highlighted in blue represent the block number where the root directory starts. When we convert the hexadecimal “00 00 00 9A” to decimal we get rootDirBlockNum, 154.
$(9 \text{ x } 16^1) + (10 \text{ x } 16^0) = 154$

Bytes at address 000218-00021B highlighted in purple represent the FAT start block. When we convert the hexadecimal “00 00 00 01” to decimal we get 1.
$(1 \text{ x } 16^0) = 1$

Bytes at address 00021C - 00021F highlighted in red represent the number of blocks FAT occupies. When we convert the hexadecimal “00 00 00 99” to decimal we get fatNumOfBlocks, 168.
$(9 \text{ x } 16^1) + (9 \text{ x } 16^0) = 153$

Bytes at address 000220 - 000223 highlighted in orange represent the block number where free space starts. When we convert the hexadecimal “00 00 00 A8” to decimale we get freeBlockNum, 168.
$(10 \text{ x } 16^1) + (8 \text{ x } 16^0) = 168$

FreeSpace, Block 1 - 153

Block 1 (first 256 bytes), shows start of FAT table, space tracking for VCB and start of space tracking for FAT table

```
000400: FE FF FF FF 02 00 00 00  03 00 00 00 04 00 00 00 | ����............
000410: 05 00 00 00 06 00 00 00  07 00 00 00 08 00 00 00 | ................
000420: 09 00 00 00 0A 00 00 00  0B 00 00 00 0C 00 00 00 | ................
000430: 0D 00 00 00 0E 00 00 00  0F 00 00 00 10 00 00 00 | ................
000440: 11 00 00 00 12 00 00 00  13 00 00 00 14 00 00 00 | ................
000450: 15 00 00 00 16 00 00 00  17 00 00 00 18 00 00 00 | ................
000460: 19 00 00 00 1A 00 00 00  1B 00 00 00 1C 00 00 00 | ................
000470: 1D 00 00 00 1E 00 00 00  1F 00 00 00 20 00 00 00 | ............ ...
```

```
000480: 21 00 00 00 22 00 00 00  23 00 00 00 24 00 00 00 | !..."...#...$...
000490: 25 00 00 00 26 00 00 00  27 00 00 00 28 00 00 00 | %...&...'...(...
0004A0: 29 00 00 00 2A 00 00 00  2B 00 00 00 2C 00 00 00 | )...*...+...,...
0004B0: 2D 00 00 00 2E 00 00 00  2F 00 00 00 30 00 00 00 | -......./...0...
0004C0: 31 00 00 00 32 00 00 00  33 00 00 00 34 00 00 00 | 1...2...3...4...
0004D0: 35 00 00 00 36 00 00 00  37 00 00 00 38 00 00 00 | 5...6...7...8...
0004E0: 39 00 00 00 3A 00 00 00  3B 00 00 00 3C 00 00 00 | 9...:...;...<...
0004F0: 3D 00 00 00 3E 00 00 00  3F 00 00 00 40 00 00 00 | =...>...?...@...
```

Bytes at address 000400 - 00403 highlighted in red represent the END_OF_FILE_FLAG. When we convert the hexadecimal "FF FF FF FE" to decimal we get the END_OF_FILE_FLAG, -2.

FE FF FF FF = 1111 1110 1111 1111 1111 1111 1111 1111
Inverted: 0000 0001 0000 0000 0000 0000 0000 0000
Add 1: 0000 0001 0000 0000 0000 0000 0000 0001
1 0000 0001 = 2
Leftmost bit is 1 -> -2

Bytes at address 000404 - 00407 highlighted in orange represent pointing to the next FAT block. When we convert the hexadecimal "00 00 00 02" to decimal we get the next FAT block (the second FAT block, block number 1), 2. This shows the FAT implementation working as detailed, in which a FAT block points to the next FAT block.
$(2 \times 16^0) = 2$

Block 2 (first 256 bytes), shows end of space tracking for FAT table, space tracking for root directory, and start of free space

```
000600: 81 00 00 00 82 00 00 00  83 00 00 00 84 00 00 00 | �...�...�...�...
000610: 85 00 00 00 86 00 00 00  87 00 00 00 88 00 00 00 | �...�...�...�...
000620: 89 00 00 00 8A 00 00 00  8B 00 00 00 8C 00 00 00 | �...�...�...�...
000630: 8D 00 00 00 8E 00 00 00  8F 00 00 00 90 00 00 00 | �...�...�...�...
000640: 91 00 00 00 92 00 00 00  93 00 00 00 94 00 00 00 | �...�...�...�...
000650: 95 00 00 00 96 00 00 00  97 00 00 00 98 00 00 00 | �...�...�...�...
000660: 99 00 00 00 FE FF FF FF  9B 00 00 00 9C 00 00 00 | �...������...�...
000670: 9D 00 00 00 9E 00 00 00  9F 00 00 00 A0 00 00 00 | �...�...�...�...
000680: A1 00 00 00 A2 00 00 00  A3 00 00 00 A4 00 00 00 | �...�...�...�...
000690: A5 00 00 00 A6 00 00 00  A7 00 00 00 FE FF FF FF | �...�...�...����
0006A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
0006B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
0006C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
0006D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
0006E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
```

```
0006F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
```

Bytes at address 000664 - 00667 highlighted in red represent the END_OF_FILE_FLAG. When we convert the hexadecimal “FF FF FF FE” to decimal, we get the END_OF_FILE_FLAG, -2. This correctly marks the end of the FAT.

FE FF FF FF = 1111 1110 1111 1111 1111 1111 1111 1111
Inverted: 0000 0001 0000 0000 0000 0000 0000 0000
Add 1: 0000 0001 0000 0000 0000 0000 0000 0001
1 0000 0001 = 2
Leftmost bit is 1 -> -2

Bytes at address 000668 - 0069F highlighted in orange represent the root directory blocks. When we convert the hexadecimal “00 00 00 9B” to decimal, we get 155. rootDirBlockNum stores the block number where the root directory starts at 154. The following blocks increase by one from 155 until the block “FF FF FF FE”, corresponding to the END_OF FILE FLAG, -2, which marks the end of the root directory.
$(9 \times 16^1) + (11 \times 16^0) = 155$

Bytes at address 0006A0 - 0006A3 highlighted in blue represent the FREE_BLOCK_FLAG. When we convert the hexadecimal “FF FF FF FF” to decimal, we get the FREE_BLOCK_FLAG, -1. This matches the freeBlockNum in the VCB at A8 00 00 00 (decimal is 168) that stores the block number where free space starts. The remainder of this portion of the dump indicates all free blocks (all FF FF FF FF).

FF FF FF FF = 1111 1111 1111 1111 1111 1111 1111 1111
Inverted: 0000 0000 0000 0000 0000 0000 0000 0000
Add 1: 0000 0000 0000 0000 0000 0000 0000 0001
$(1 \times 16^0) = 1$
Leftmost bit is 1 -> -1

Root Directory, Block 154 - 167

Block 154, shows start of root directory

```
013600: 2E 00 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: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013640: 00 00 00 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: 00 00 00 00 00 00 00 00  9A 00 00 00 01 00 00 00 | ........�.......
```

```
013670: 01 00 00 00 00 00 00 00  BF 53 F0 67 00 00 00 00 | ........�S�g....
013680: BF 53 F0 67 00 00 00 00  2E 2E 00 00 00 00 00 00 | �S�g............
013690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0136A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0136B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0136C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
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: 9A 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00 | �...............

013700: BF 53 F0 67 00 00 00 00  BF 53 F0 67 00 00 00 00 | �S�g....�S�g....
013710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013720: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013730: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013740: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013750: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013760: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013770: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013780: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013790: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0137A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0137B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0137C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0137D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0137E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0137F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes at address 013600 - 013663 in red represent the name for the current directory. When we convert "00 00 00 2E" to ASCII we get "."
$(2 \times 16^1) + (14 \times 16^0) = 46$
46 in ASCII: .

Bytes at address 013664 - 013667 in orange represent the size for the current directory. When we convert "00 00 00 00" to decimal we get 0.
$(0 \times 16^0) = 0$

Bytes at address 013668 - 01366B in blue represent the location for the current directory, it points at itself. When we convert "00 00 00 9A" to decimal we get 154 (the current block number, where the root directory starts).
$(9 \times 16^1) + (10 \times 16^0) = 154$

Bytes at address 01366C - 01366F in green represent if it is a directory. When we convert "00 00 00 01" to decimal we get 1 for the flag isDirectory (so it is a directory).
$(1 \times 16^0) = 1$

Bytes at address 013670 - 013673 in purple represent if it is used. When we convert “00 00 00 01” to decimal we get 1 for the flag used (so it is used).
$(1 \times 16^0) = 1$

Bytes at address 013678 - 01367F in orange represent the last time the file was created at. When we convert “00 00 00 00 67 F0 53 BF” to decimal we get the timestamp for when the file was created at 1743803327.
$(6 \times 16^1) + (7 \times 16^0) = 103$
$(15 \times 16^1) + (0 \times 16^0) = 240$
$(5 \times 16^1) + (3 \times 16^0) = 83$
$(11 \times 16^1) + (15 \times 16^0) = 191$
$(103 \times 16^6) + (240 \times 16^4) + (83 \times 16^2) + (191 \times 16^0) = 1743803327$

Bytes at address 013680 - 013687 in yellow represent the last time the file was modified at. When we convert “ “00 00 00 00 67 F0 53 BF” to decimal we get the timestamp for when the file was last modified at 1743803327, the same as when it was last created.

Bytes at address 013688 - 0136EB in cyan represent the name of the second entry, the parent directory (itself for root). When we convert “00 00 2E 2E” to ASCII we get “..”
$(2 \times 16^1) + (14 \times 16^0) = 46$
46 46 in ASCII: ..

Bytes at address 0136EC - 0136EF in green represent the size for the parent directory. When we convert “00 00 00 00” to decimal we get 0.
$(0 \times 16^0) = 0$

Bytes at address 0136F0 - 0136F3 in purple represent the location for the parent directory, it points at itself since root is its own parent. When we convert “00 00 00 9A” to decimal we get 154 (the current block number, where the root directory starts).
$(9 \times 16^1) + (10 \times 16^0) = 154$

Bytes at address 0136F4 - 0136F7 in red represent if it is a directory. When we convert “00 00 00 01” to decimal we get 1 for the flag isDirectory (so it is a directory).
$(1 \times 16^0) = 1$

Bytes at address 0136F8 - 0136FB in orange represent if it is used. When we convert “00 00 00 01” to decimal we get 1 for the flag used (so it is used).
$(1 \times 16^0) = 1$

Bytes at address 013700 - 013707 in blue represent the last time the file was created at. When we convert “00 00 00 00 67 F0 53 BF” to decimal we get the timestamp for when the file was created at 1743803327, the same as the current directory.

Bytes at address 013708 - 01370F in green represent the last time the file was modified at. When we convert " "00 00 00 00 67 F0 53 BF" to decimal we get the timestamp for when the file was last modified at 1743803327, the same as when it was created.

The remaining bytes of this portion of the hexdump is all 0, indicating the remaining blocks are free with their used set to 0.

A description of the VCB structure

The *signature* is a unique identifier to verify the file system type. The *volumeSize* field is the size of the volume in bytes. The *blockSize* field is the size of each block in bytes. The *numOfBlocks* field is the total number of blocks in the volume. The *fsBlockNum* is the block number where the filesystem metadata is. The *rootDirBlockNum* is the block number where the root directory is. The *fatBlockNum* is the block number where the free space management starts and *fatNumOfBlocks* is the number of blocks that it occupies.

```
typedef struct VolumeControlBlock
{
   int signature; // unique identifier to verify the file system type
   int volumeSize; // size of the volume in bytes
   int blockSize; // size of each block in bytes
   int numOfBlocks; // total number of blocks in the volume
   int fsBlockNum; // block number where the filesystem metadata is
   int rootDirBlockNum; // block number where the root directory is
   int fatBlockNum; // block number where the free space management (FAT) starts
   int fatNumOfBlocks; // number of blocks FAT occupies
   int freeBlockNum; // block number where free space starts
} VCB;
```

A description of the Free Space structure

To track the free space in our filesystem, we are using a File Allocation Table (FAT) which consists of an array of integers. Our logic is that we first initialize the free space in memory using malloc() for our FAT table and then set all the entries inside of that array to -1, which acts as free space with the exception of the first block which is set to -2 (EOF Flag) to indicate the VCB. We then write that table to disk using LBAWrite.

Next for allocating blocks, we created an array (blockAllocations) to keep track of which blocks are to be allocated. This is to make sure there's enough free blocks before marking them as allocated in the FAT table. Otherwise, we return an error if there's not enough space.

findNextFreeBlock() uses another for loop to find the first free space block in fatTable and returns that index back to nextFreeBlock. From here, we can now get to the allocation portion and we set that block along with the following blocks as used (with their respective index) with the number of blocks specified in the argument. All other blocks point to the next block in the chain (acting as a linked list). We mark the last block in the chain as the end of file flag (-2) and then write it to disk.

This also handles a edge case where if a file's blocks gets released and all the blocks get written as -1, we aren't confident that the next block is *truly* free, hence what findNextFreeBlock essentially does.

We use our findNextFreeBlock helper function once more to update the current freeBlockNum of our VCB so we don't have to search for the next free block the next time we call allocateBlocks.

Finally, we return startBlockNum which is the first block in the allocated chain that you'd iterate through until you reached a block with the end of file flag to access all the blocks of whatever file this is being used for.

A description of the Directory system

Directories are stored as a collection of directory entry structures; fifty empty entries are initialized in rootDirBlockNum, which stores the block number that the root directory is (within the VCB structure). numOfEntries represents the maximum number of entries and is used to find the number of blocks (requiredNumOfBlocks) needed to allocate for the entries. "." and ".." entries represent the current directory's location or parent directory's location (itself for root), respectively. All entries aside from "." and ".." are initialized with used = 0. Directories are allocated using FAT, the function allocateBlocks() is used to find and link free blocks.

Looking ahead, when we are given a path to look for a file, we will read the root directory from disk, recursively look for subdirectories, and read them from disk until we find the file.

A table of who worked on which components

Component	**Who worked on it**
VCB	Bryan
Free space	Ethan
Root Directory	Nabeel
Hexdump analysis	Leigh

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

We made sure to meet up in person at least once to discuss the initial planning phase. After that, we met on several Discord calls to discuss our progress and issues we had. We divided the tasks into the components suggested by the assignment, but would discuss ideas of all the components since they are very related to each other.

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

We had a couple of bugs with our allocate blocks function. First, we noticed that the root directory was not showing up in the FAT table hexdump. We realized that after allocating blocks for the root directory to the FAT table in memory, we forgot to write it to disk. Second, we noticed the root directory's start block was skipping a block. This was because we were not including the current free block that we were tracking as a block to allocate to.