File System Milestone 1

Team X

Github: jonathancurimao

Github Repository: https://github.com/CSC415-2024-Fall/csc415-filesystem-jonathancurimao.git

Names:	ID's:
Jonathan Ho	923536806
Yashwardhan Rathore	921759459
Jonathan Curimao	921855139
Juan Segura Rico	921725126

Description of our VCB Structure

The Volume Control Block (VCB) is the main structure that stores key details about our file system. It includes a unique number called magicNumber to check if the system is already set up. The blockSize field defines the size of each block in bytes, and numberOfBlocks gives the total count of blocks in the system. To track free space we have used the freeSpace field. freeSpace field points to where the free space map begins and helps us to manage storage efficiently. Then, rootDirStart points to the starting location of the root directory which allows us to locate and work with files from the root level of the file system. All of these fields allow the VCB to keep track of the storage layout, helping us initialize, access, and manage the file system.

Description of our FreeSpace Structure

The free space structure utilizes a bitmap to quickly search for free blocks. This bitmap is initialized in a function that mallocs the memory needed for the bits, and it would initialize all bits to 0, since all blocks would be free initially. The freeSpace would also be accompanied by the helper functions allocateBlock() and freeBlock(), which would manage the blocks being used. Another helper function that is implemented in this structure is getBlockState, which would essentially check if a block is at a given index or not. The freeSpace would make use of our global volume control block structure for its functions.

Description of Directory System

Our directory entry structure itself is fairly simple. It contains very simple fields such as a character array for the filename, of which it can hold 100 characters, a file location integer which stores the block that the file is held on, the file size, time created and modified, the file number, and booleans for whether the file itself is a directory or is used or not. Our create directory function, when given a number of entries, will loop through all entries and initialize these values to a “free” state, in which these entries are unused. The first two entries in the directory are always reserved for “.” and “..”.

Who worked on what?

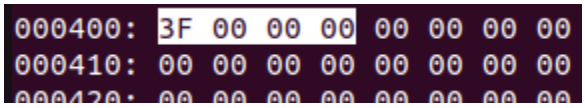
Jonathan Ho	dirEntry structure, fsInit, Hexdump analysis
Yashwardhan Rathore	Initialize VCB, fsInit
Jonathan Curimao	freeSpace structure + description
Juan Segura Rico	freeSpace + description/pdf

How often did we meet?

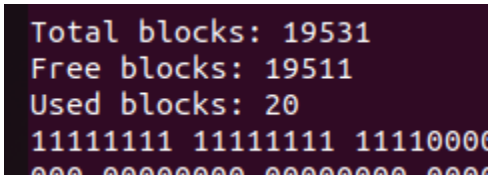
We would meet about 2-3 times a week in discord, but would talk and occasionally plan things out everyday whenever needed in-person. For the tasks, we essentially divided up the tasks based on preference for which part a team member wanted to do, but we would also assist each other if needed.

What issues were faced? How were they resolved?

Hexdump free space map did not match the expected value of how many blocks were allocated. Our initial hexdump showed a value of 3F. Translating this to binary yielded a value of “00111111”, meaning only the first 6 blocks marked as being occupied.

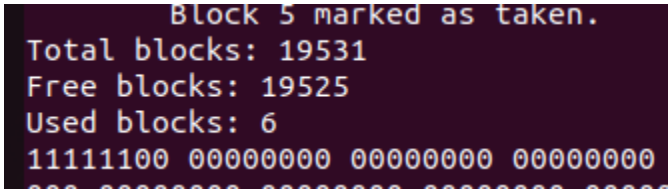


Our custom print bitmap function shows the expected output:

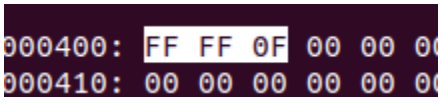


It should display 20 blocks marked, not only 6. We have 6 blocks after allocating our free space map

Upon closer inspection, I was calling LBAwrite too early. Calling our custom print function at the point of writing to disk shows that I was writing our free space map before creating and allocating the root directory. The solution is to either re-call LBAwrite at the end, or call LBAwrite only once AFTER the root directory is created.



Correcting this simple mistake leads to the expected output: “0FFFFF”, which translates to 20 bits marked in binary.



Refactoring code: double VCB declaration

```c
int initFileSystem (uint64_t numberOfBlocks, uint64_t blockSize)
    {
    printf ("Initializing File System with %ld blocks with a block siz
    /* TODO: Add any code you need to initialize your file system. */

    // Step 1: Allocate memory for the VCB and read block 0
    VCB *vcb = (VCB *)malloc(blockSize);
```

```c
#include "volumeControlBlock.h"

#define BLOCK_SIZE 4096 // Example block size in bytes
#define NUM_OF_BLOCKS 512 // The total number of blocks

VCB superBlockObj;
unsigned char *bitmap = NULL; // Bitmap array for free blocks
```

We had two separate VCB objects that we were updating independently of each other. This didn't cause any problems yet, but had potential to make things harder in the project later on. There were two solutions to this problem: either pass in the original VCB or make a singular VCB object global. Because our functions already assumed the VCB was global, I did not want to have to refactor all of our existing functions, so I opted for the latter.

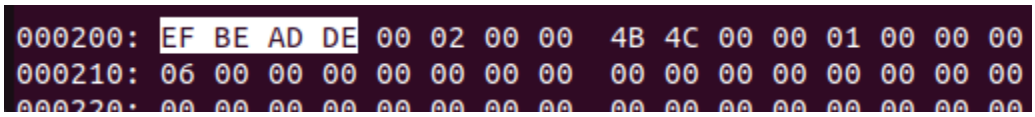
```
student@student:~/Documents/csc415-filesystem-jonathancurimao$ make
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsLow.o -g -I. -lm -l readline -l pthread
/usr/bin/ld: fsInit.o: in function `initFileSystem':
/home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:39: undefined reference to `initVCB'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:72: undefined reference to `initFreeSpace'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:73: undefined reference to `getBlockState'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:76: undefined reference to `allocateBlock'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:92: undefined reference to `allocateBlock'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:105: undefined reference to `createDir'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:133: undefined reference to `freeVCB'
collect2: error: ld returned 1 exit status
make: *** [Makefile:61: fsshell] Error 1
student@student:~/Documents/csc415-filesystem-jonathancurimao$
```

I was getting compilation errors from making custom .h and .c files. The compiler would complain about references to undefined functions. After doing some research, it turns out that I had to edit the Makefile to tell the compiler to compile my external files, like so:

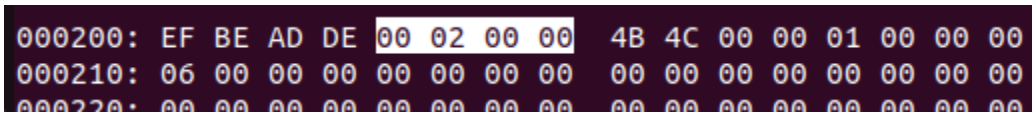
```
# Add any additional objects to this list
ADDOBJ= fsInit.o dirEntry.o freeSpace.o volumeControlBlock.o
```

This solved my compiler issues.

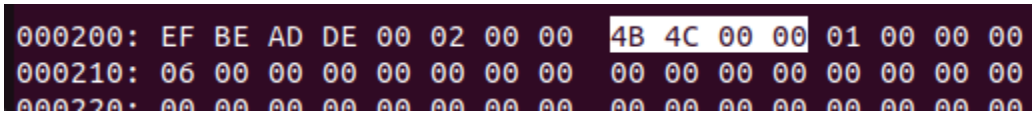
Analysis of our HexDump:



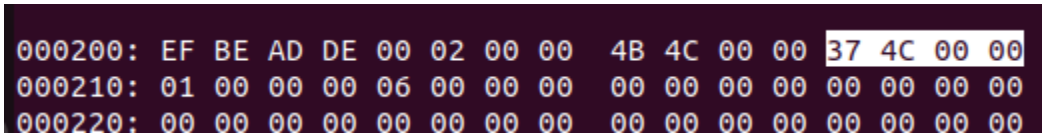
This is our hexdump. Block 1 contains the VCB structure. Our VCB structure consists of 6 ints; each int element is 4 bytes long, meaning that it will occupy 24 bytes total. “0xDEADBEEF” is our magic number, which is shown in the above picture.



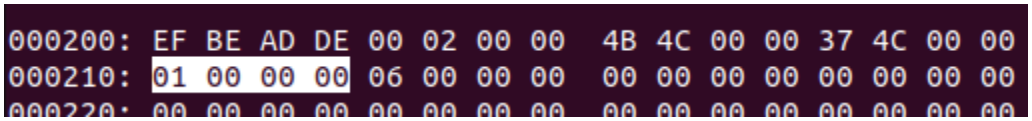
The next element in our VCB struct is blockSize. The default is 512, which translates to 0x0200 in hexadecimal. This is shown in the above. Reading from right to left, it prints “00000200”, which correlates to our blockSize value.



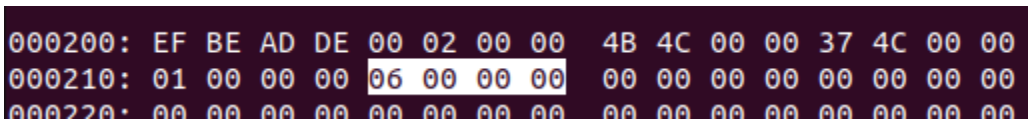
Our third element is numberOfBlocks. The default is 19,531 blocks, which translates to 0x4C4B. These can be seen in the next 4 bytes, reading “00004C4B”.



The fourth element is the number of free space blocks left. After initialization, we allocate 20 blocks, meaning we should have 19311 free blocks. Translating the above selection does indeed result in 19311 blocks.

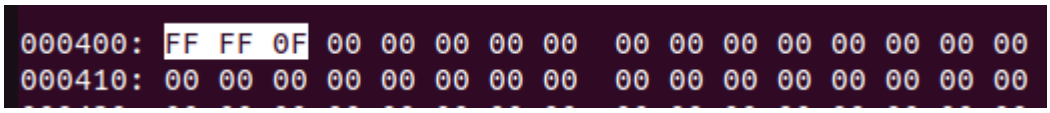


The fifth element is the value of the block position where our free space map begins. Our free space map spans 5 blocks and starts at block 2 according to the hexdump. Our internal block tracking is indexed from 0, so the expected value should be 1 in decimal; this translates to simply 0x01 in hex. This correlates with the above print.



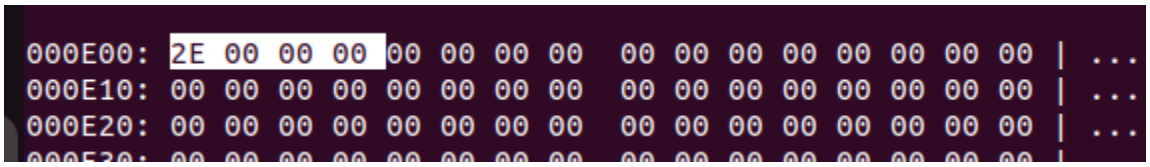
The last printed number is the block location where the root directory starts. Our free space map ends at block 6, so the root directory starts on block 7. The expected value should be 0x06 in hex, which is reflected in the highlighted selection.

To see our freespace map, we need to see blocks 2 - 6. Right now, with 50 directories created, we should see 20 blocks being marked as occupied. Analyzing block 2, where the free space map begins, yields “0F FF FF”:

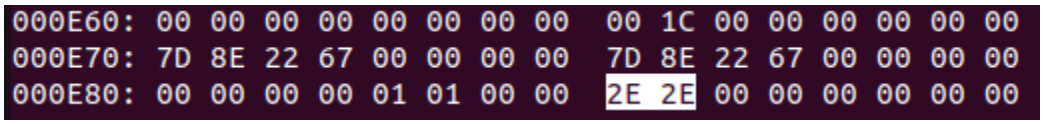


This is the expected value, since translating that number into binary results in 20 “1” bits in a row.

Next, we need to analyze the root directory structure. Starting at block 7, we see this data in the hexdump:



The very first element is our filename. Doing research online indicates that “2E” is actually the hex representation of “.”, which is the filename of the very first element in our root directory. The size of our directory entry structure is 136 bytes, so we should jump 136 bytes over to see elements of our next directory entry. Notice how the first directory entry starts on E00; if we jump 136 bytes over, we should end up on E88. Our hexdump matches with our expected jump location:



We end up on E88! Since the next entry in our directory is “..”, we have “2E 2E”. This confirms that our root directory is being initialized and written to disk.