

SAN FRANCISCO STATE UNIVERSITY

MILE STONE 01

MKFS

Members : Tyler Fulinara , Vinh Ngo , Rafael Sant Ana Leitao, Anthony Silva , Rafael Fabiani

IDs: 922002234, 920984945, 922907645, 921919541, 922965105

Course: CSC-415-01 Operating Systems

CSC415 Operating Systems: Milestone 01

July 13, 2023

Members: Tyler Fulinara , Vinh Ngo , Rafael Sant Ana Leitao, Anthony Silva , Rafael Fabiani
IDs: 922002234, 920984945, 922907645, 921919541, 922965105
Course: CSC-415-01 Operating Systems

1 VCB Structure

The volume control block component for our file system has in it meta data needed for the volume as well as code needed to initialize itself and write itself to block 1 of the volume. The first thing that happens as our file system begins to be initialized is the initialization of the vcb. The initialization of the vcb only ever occurs if the magic number is not found in the Volume. If this occurs the fields in the vcb are filled, the fat table is initiated, and the root directory is initiated. The VCB is located on block 0 of our sample volume. It is based on the following structure :

```
typedef struct VCB { //size of fields, desc of field
uint32_t total_blocks_32; // 4 bytes, the total number of blocks in the file system
uint32_t FAT_start; // 4 bytes, Start of the FAT Table
uint32_t FAT_size_32; // 4 bytes, total blocks in the FAT
uint32_t root_cluster; // 4 bytes, location of the root
uint32_t free_space; // 4 bytes, amount of free blocks
uint32_t magic_number; // 4 bytes, Magic Number
uint32_t entries_per_dir; // 4 bytes
uint16_t bytes_per_block; // 2 bytes, blockSize,
uint16_t reserved_blocks_count; // 2 bytes, reserved blocks count
uint16_t root_entry_count; // 2 bytes, entries in the root
} VCB;
```

The VCB component of our file system has a large influence upon how the fat is initialized, how the volume's memory is laid out and how directories are created. The source file contains the implementation of the functions found in the header. Namely functions to initialize the vcb, another to read from the disk and if it doesn't exist or isn't initialized, it initializes it with default values and writes it back to the disk. In case of failure, it frees allocated memory and returns -1 to indicate an error. In the case that the VCB is already initialized, it reads the VCB from the disk, then it tries to loads the root directory, and sets the current working directory to the root. The **vcb_read_from_disk** function reads the VCB from a specific disk block using the LBAREad method, namely it reads it from block 0. The **vcb_is_init** function checks if the VCB has been initialized by verifying its magic number. The magic number of the volume is arbitrarily chose as of yet it is set to be 9090. If it is the case that the VCB pointer happens to be null or the magic number doesn't match the expected value possibly due to corruption, then function returns 0, indicating a failure to init the VCB.

2 Free Space Structure

The free space management component of our file system is handled by the F.A.T which maps the available space on the file system to blocks in memory, marking used blocks and free blocks in the array. This component of our file system formats the volume and uses an array to keep track of locations on the volume. It's responsibilities are keeping tack of free space and finding available locations for our file system to write data to. It has functions to find free blocks, to init the fat, to read the fat and to update the fat. The functions in the source code are intended to handle tasks such as finding the first empty block, updating the FAT on the disk, initializing and reading the FAT, allocating blocks for directory entries, and getting the total number of free blocks. There is also error handle in the event of failures in various operations like handling memory allocation

and reading from disk. The core functions are meant for initializing the FAT, managing blocks. Error handling is implemented in these functions.

A key functionality in the FAT source file is the **fat_init** function, which determines the size of the FAT in bytes and then translates that to blocks based on the provided number of blocks and block size. Memory is then set aside for the FAT array. Each element in this array represents a block on the disk, where the value at each index is the index of the next block in the file or -1 if it's the last block of the file. The FAT is then updated on the disk using the **update_fat_on_disk** function. The intent of the **allocate_blocks** function is to try and find the first empty block in the FAT and then begin allocation from there. In a later portion of the project when we are working with actual file blobs this function is meant to handle the case that additional blocks need to be allocated to an existing file.

3 Directory System

The Directory management component of our file system is handled by **root_init**. This portion of our file system is responsible for the initialization of the root directory and in the future will be refactored so as to initialize any directory. The header of this file contains two structures :

```
typedef struct Directory_Entry {  
char dir_name[NAME_MAX_LENGTH];  
uint32_t dir_attr;  
uint32_t dir_first_cluster;  
uint32_t dir_file_size;  
uint32_t entries_array_location;  
} Directory_Entry;
```

The Directory Entry structure is meant to represent a directory entry in our file system. Entries in our file system have names, an attribute which signifies whether that entry is a directory entry or a file. This attribute currently only holds whether the entry is a directory or not but in the future may be expanded upon so as to encode multiple attributes in a single field. Directory entries also have a size associated with them and a block location for an array of entries provided the entry is a directory. If the entry is not a directory this field is not used.

The Second structure part of our directory management is :

```
typedef struct Current_Working_Directory {  
char path[DIRECTORY_MAX_LENGTH];  
uint32_t cluster;  
} Current_Working_Directory;
```

This structure is meant to represent the current working directory. It contains a path and a starting cluster. This may be revised in the future and was made with the intent of being used and or adapted for mile stone 2.

The core task of the **root_init.c** file is the initialization and management of directory entry , although currently it was made with the intent to handle directories. Core aspects of this source file **.init_directory** creates a new directory within a given parent directory or at the root if no parent directory is provided. In earlier revisions of this function it was geared toward initializing the root directory. However from lecture it became more apparent to our team that the creation of a root directory was in a sense a special case of a more general function which created directories. Taking this idea the entire source file and header of **root_init** was changed. The name still remains but may change to better reflect what the purpose of these two files really are i.e directory entry management. The next function is **clear_current_working_directory**, which resets the current working directory, and was made with the foresight that in later milestones we may need to handle and work with the current working directory as such it was our reasoning that a structure be created. There is also a function to read directories from the volume namely **load_directory**, which as the name suggest attempts to load a directory from the volume into memory.

4 Table of Component Assignments

Each member of the team had a component they were responsible for implementing that would work together with the other components. Each component of our file system had its own purpose, however the functionality of the file system as a whole depends on a cohesion and correct alignment of each component to operate correctly. Although each team member or team members was responsible for their aspect of the file system a lot of the

functionality of each component were related. As such communication and group effort was needed for each portion. A lot of collaboration came into play when it became more apparent how interrelated each portion of the project was, it seemed that the file system did not just count on one single component successfully functioning but rather it depended on both the component functioning and the components working together in a coherent manner.

Component	Team Member(s)
VCB	Rafael Fabiani
FAT	Vinh Ngo
root_init	Anthony Silva
mfs	Rafael Sant Ana Leitao and Tyler Fulinara

Table 1: Component Assignments

5 Team Collaboration

Team collaboration was a key aspect of this project without it, it would not have been possible to weave together each aspect of the file system. It was essential that we were talking to one another, giving each other updates on our progress as well as discussing how the individual components would fit together. In many ways forming this basis for our file system was like putting together a machinery of gears each moving together towards the forming of one larger contraption. To work together it was necessary for us to be in constant contact and we were able to establish and maintain this thanks to the enthusiasm, effort and willingness each team member had towards conveying ideas, suggesting revisions, and solving problems. There were many obstacles to overcome first off each of us lived in different cities, with our own affairs to balance along with school life. Although our performance on this project was important to each of us it was also important for us to remember that we each have a life outside of csc-415, with our own daily lives to manage. I personally am grateful to have had the opportunity to meet this passionate, ambitious and humble collection of people, as we worked together seeing how many sacrifices each of my team mates made to set aside time from , family, work and the responsibilities each of us have to maintain ourselves and keep a roof over our heads and take care of the people we hold dear to us to put time into this project. We each motivated and uplifted one another and urged each other to be patient and resilient in the face of adversity. I saw in my group a gathering of motivated people with challenge and desire driving them towards the betterment of themselves and the pursuit of higher truth.

We met twice each week , shared each others contact info to make ourselves available even outside of discord in the event that an update or bug was found that needed to be addressed and we were away from keyboard. Our forms of communication ranged from phone calls to screen sharing to in person meetings where each group member shared their take on our progress and ideas for moving the project forward.

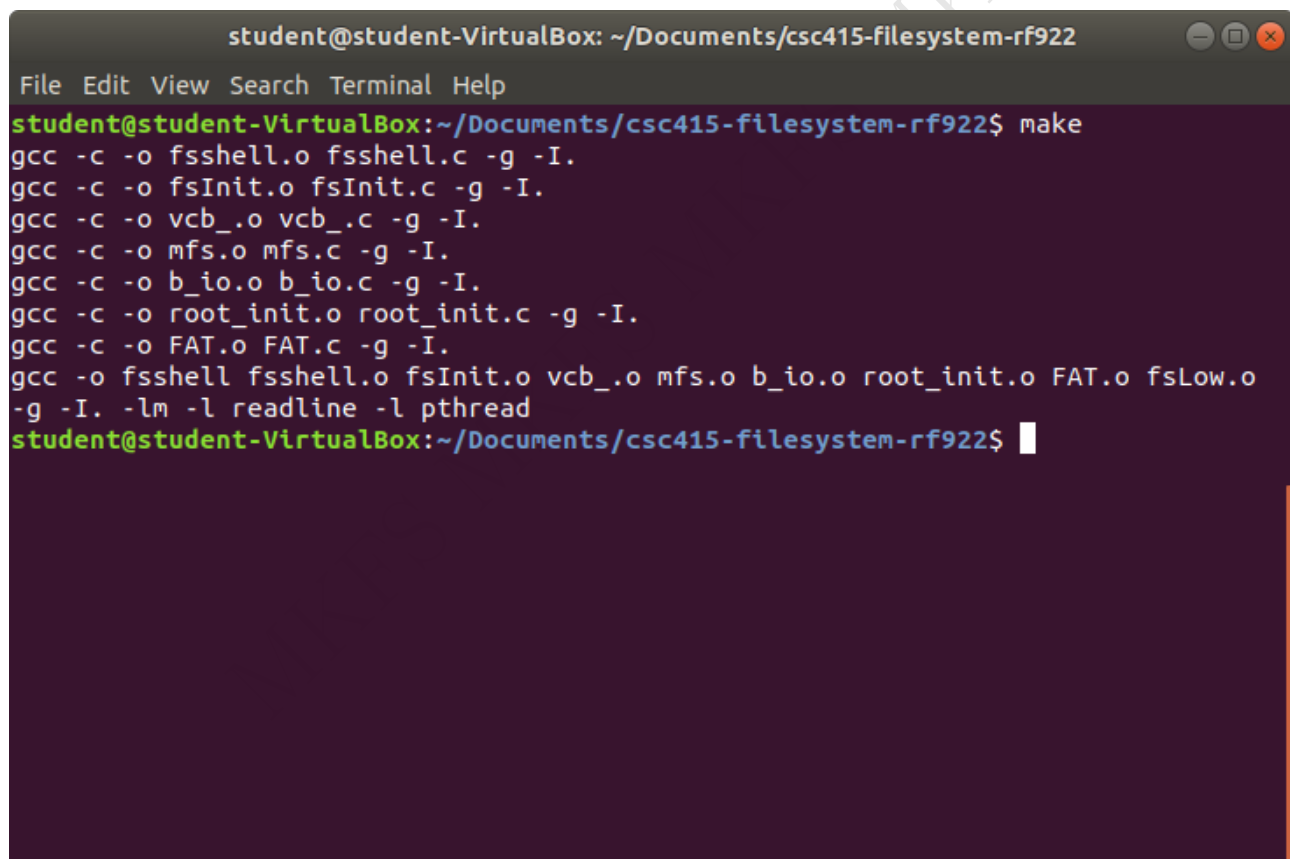
6 Issue Resolution

We had many issues to resolve and it seemed that whatever progress we made unveiled a new string of issues to carefully navigate. One of the earliest and most concerning of these was the handling of writing our data to the volume. At first we had written an outline of our approach but noticed that nothing was being written to the disk and upon review we realized that it was because we hadnt actually written anything to disk. This was resolved in part by rereading the readme, meeting and speaking about what each of us thought would need to be changed and using the **LBawrite** function in the appropriate circumstances where it would be needed. One of the next issues which persisted for a time was making sure that the vcb was properly initialized, at first it was the case that the volume would rewrite the disk regardless of whether or not the vcb had already been initialized, to resolve this a conditional was used to check if the magic number was present. Initialization of this structure took determining what values were most appropriate for our file system. After which it seemed that the more we discussed the more a general form for how initialization of this structure would take place. Another problem we had was making sure the FAT was working properly, one issue in particular was making sure that the correct values were given to both **LBawrite** and **LBaread**. One of the issues in particular was ensuring that the fat was initialized and that meant that the output of the hex dump would align with how we structured the initialization. To resolve this took several hours of planning and a lot of refactoring. Another difficult aspect of the project so far was ensuring that we didnt let any parts of the code get too cumbersome or so long that it was hard to read and therefore would likely be difficult to maintain. This was particularly true for the **mfs** file. To resolve this issue two people worked on this aspect of the assignment while we all had to contribute by sharing ideas and helping form strategies for simplifying the code. Our main resolution for this aspect of the project was the introduction of several helper functions to handle smaller tasks within the

functionality of a larger function. This had the effect of making functions easier to maintain as well as easier for us to read. Another issue came about in the initialization of the root, namely there were times where the root would not be written or would appear in an incorrect location. To troubleshoot this we relied heavily upon the use of print statements and reviewing the outline for approaching this that was mentioned in some of the lectures. I think a large part of this project was made easier given that each of us made sure to be present whenever there was class, this way we would each get to hear and see a strategy for a component of the file system. Then afterward we would each come together to review and what we found during these discussions was that each of us keyed in on different aspects of the implementation but when we came together we were able to bring each of those perspectives into focus and were able to plan a way for implementing the ideas we saw in class for our file system. Another issue was how geared **root_init** was toward only handling the initialization of the root where instead it would be better and beneficial in the long run for us to refactor it such that it could work for a directory entry and treat the root case as a special case in and of itself.

7 HeX Dump of the Volume File

7.1 Compilation

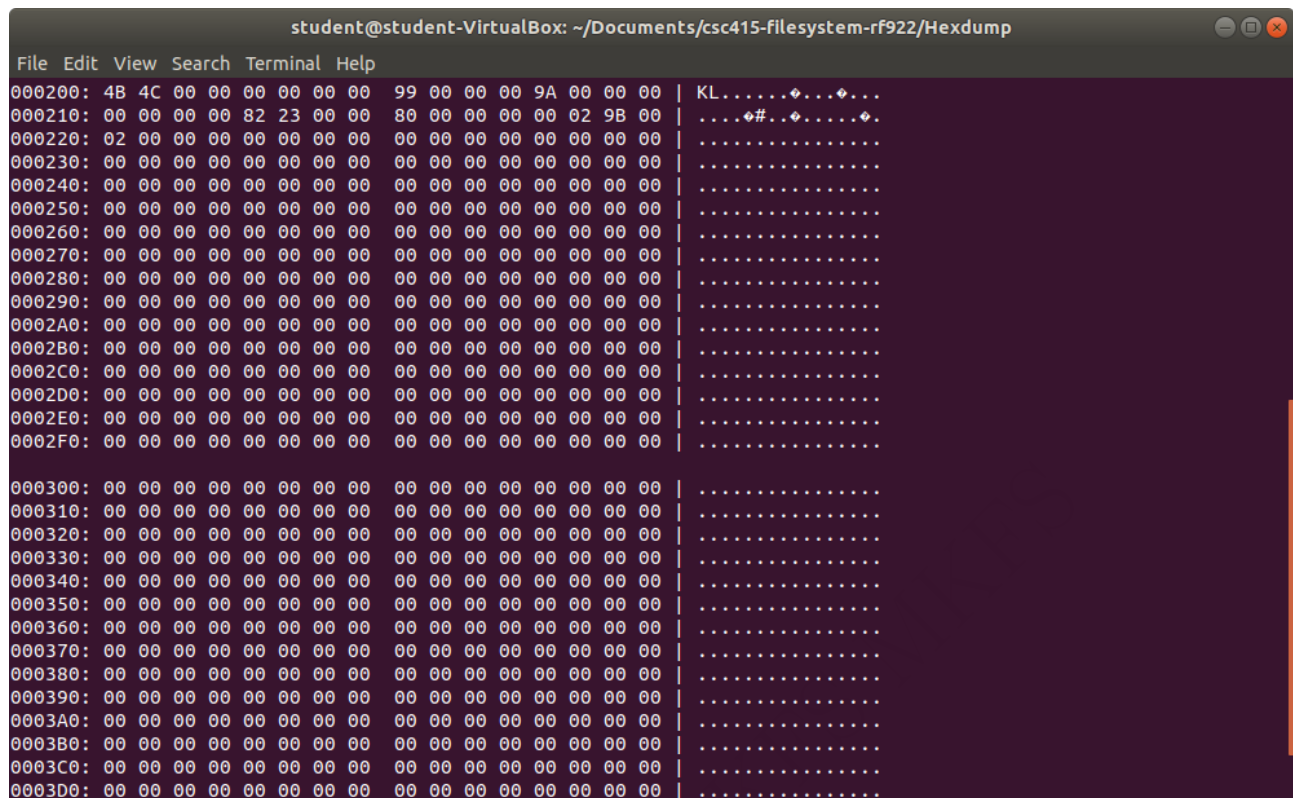
A screenshot of a terminal window titled "student@student-VirtualBox: ~/Documents/csc415-filesystem-rf922". The terminal shows the execution of the 'make' command, which compiles several C source files into object files and then links them into a final executable. The compilation commands are: gcc -c -o fsshell.o fsshell.c -g -I., gcc -c -o fsInit.o fsInit.c -g -I., gcc -c -o vcb_.o vcb_.c -g -I., gcc -c -o mfs.o mfs.c -g -I., gcc -c -o b_io.o b_io.c -g -I., gcc -c -o root_init.o root_init.c -g -I., gcc -c -o FAT.o FAT.c -g -I., and gcc -o fsshell fsshell.o fsInit.o vcb_.o mfs.o b_io.o root_init.o FAT.o fsLow.o -g -I. -lm -l readline -l pthread. The terminal output shows the successful compilation of each file and the final linking step.

```
student@student-VirtualBox: ~/Documents/csc415-filesystem-rf922
File Edit View Search Terminal Help
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o vcb_.o vcb_.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -c -o root_init.o root_init.c -g -I.
gcc -c -o FAT.o FAT.c -g -I.
gcc -o fsshell fsshell.o fsInit.o vcb_.o mfs.o b_io.o root_init.o FAT.o fsLow.o
-g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$
```

Figure 1: picture of compilation

7.2 VCB HEX DUMP

The following is a picture of compilation and the hex dumps for each section along with a brief interpretation of the hex seen.



```
student@student-VirtualBox: ~/Documents/csc415-filesystem-rf922/Hexdump
File Edit View Search Terminal Help
000200: 4B 4C 00 00 00 00 00 00 99 00 00 00 9A 00 00 00 | KL.....
000210: 00 00 00 00 82 23 00 00 80 00 00 00 00 02 9B 00 | ....#...
000220: 00 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 | .....

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 | .....
```

Figure 2: hex dump.png

total_blocks_32 (4 bytes): 4B 4C 00 00 -> 00004C4B in Little-Endian -> 19515 in Decimal
FAT_start (4 bytes): 00 00 00 00 -> 00000000 in Little-Endian -> 0 in Decimal
FAT_size_32 (4 bytes): 99 00 00 00 -> 00000099 in Little-Endian -> 153 in Decimal
root_cluster (4 bytes): 9A 00 00 00 -> 0000009A in Little-Endian -> 154 in Decimal
free_space (4 bytes): 00 00 00 00 -> 00000000 in Little-Endian -> 0 in Decimal
magic_number (4 bytes): 82 23 00 00 -> 00002382 in Little-Endian -> 9090 in Decimal
entries_per_dir (4 bytes): 80 00 00 00 -> 00000080 in Little-Endian -> 128 in Decimal
bytes_per_block (2 bytes): 00 02 -> 0200 in Little-Endian -> 512 in Decimal
reserved_blocks_count (2 bytes): 9B 00 -> 009B in Little-Endian -> 155 in Decimal
root_entry_count (2 bytes): 02 00 -> 0002 in Little-Endian -> 2 in Decimal

7.3 Start of FAT

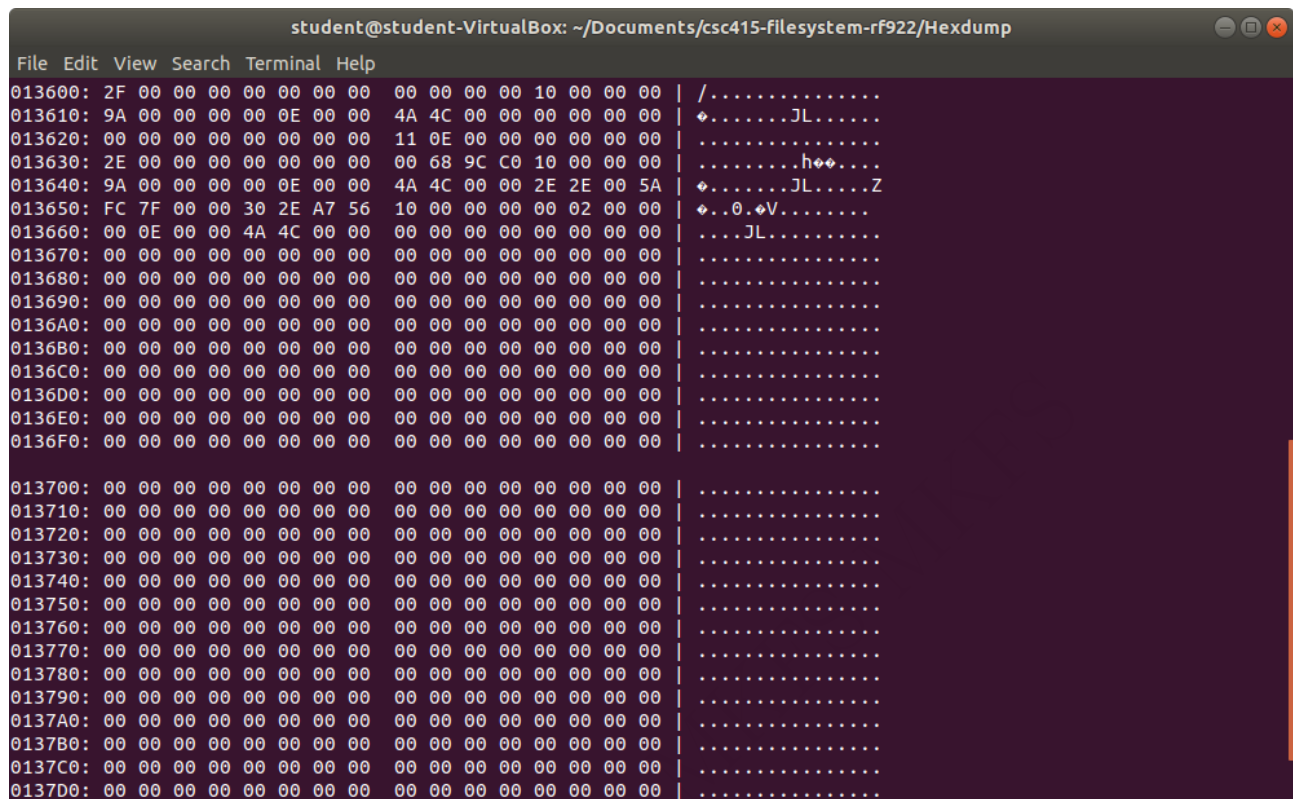
```
student@student-VirtualBox: ~/Documents/csc415-filesystem-rf922/Hexdump
File Edit View Search Terminal Help
000400: 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 | .....
000410: 05 00 00 00 06 00 00 00 07 00 00 00 08 00 00 00 | .....
000420: 09 00 00 00 0A 00 00 00 0B 00 00 00 0C 00 00 00 | .....
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 | =...>...?...@...

000500: 41 00 00 00 42 00 00 00 43 00 00 00 44 00 00 00 | A...B...C...D...
000510: 45 00 00 00 46 00 00 00 47 00 00 00 48 00 00 00 | E...F...G...H...
000520: 49 00 00 00 4A 00 00 00 4B 00 00 00 4C 00 00 00 | I...J...K...L...
000530: 4D 00 00 00 4E 00 00 00 4F 00 00 00 50 00 00 00 | M...N...O...P...
000540: 51 00 00 00 52 00 00 00 53 00 00 00 54 00 00 00 | Q...R...S...T...
000550: 55 00 00 00 56 00 00 00 57 00 00 00 58 00 00 00 | U...V...W...X...
000560: 59 00 00 00 5A 00 00 00 5B 00 00 00 5C 00 00 00 | Y...Z...[...\\...
000570: 5D 00 00 00 5E 00 00 00 5F 00 00 00 60 00 00 00 | ]...^..._...'...
000580: 61 00 00 00 62 00 00 00 63 00 00 00 64 00 00 00 | a...b...c...d...
000590: 65 00 00 00 66 00 00 00 67 00 00 00 68 00 00 00 | e...f...g...h...
0005A0: 69 00 00 00 6A 00 00 00 6B 00 00 00 6C 00 00 00 | i...j...k...l...
0005B0: 6D 00 00 00 6E 00 00 00 6F 00 00 00 70 00 00 00 | m...n...o...p...
0005C0: 71 00 00 00 72 00 00 00 73 00 00 00 74 00 00 00 | q...r...s...t...
0005D0: 75 00 00 00 76 00 00 00 77 00 00 00 78 00 00 00 | u...v...w...x...
```

Figure 3: Start of the fat dump

From the output of hexdump we see that the initialization of the blocks aligns with the code and that every block is set to the index of the next block.

7.4 Root dump



```
student@student-VirtualBox: ~/Documents/csc415-filesystem-rf922/Hexdump
File Edit View Search Terminal Help
013600: 2F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | /.....
013610: 9A 00 00 00 00 00 0E 00 00 4A 4C 00 00 00 00 00 | .....JL....
013620: 00 00 00 00 00 00 00 00 00 11 0E 00 00 00 00 00 | .....h.....
013630: 2E 00 00 00 00 00 00 00 00 00 68 9C C0 10 00 00 00 | .....h.....
013640: 9A 00 00 00 00 00 0E 00 00 4A 4C 00 00 2E 2E 00 5A | .....JL....Z
013650: FC 7F 00 00 30 2E A7 56 10 00 00 00 00 02 00 00 00 | .....V.....
013660: 00 0E 00 00 4A 4C 00 00 00 00 00 00 00 00 00 00 | .....JL.....
013670: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
013680: 00 00 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: 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: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
013700: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
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 | .....
```

Figure 4: Root Dump

dir_name (11 bytes): 2F 00 00 00 00 00 00 00 00 00 00 00 -> ASCII "/" followed by NULL and padding.
dir_attr' (1 byte): 10 -> 0x10 in hex -> 16 in Decimal
dir_first_cluster (4 bytes): 9A 00 00 00 -> 0000009A in Little-Endian -> 154 in Decimal
dir_file_size (4 bytes): 00 0E 00 00 -> 00000E00 in Little-Endian -> 3584

And we also see the two dot and dot dot entries in the following

2E 00 01 00 00 00 00 00 00 00 00 00 10 00 00 00

and

2E 00 01 00 00 00 00 00 00 00 00 00 10 00 00 00

with their corresponding fields set