Group Term Assignment - File System

Members:
Ulices Gonzalez, Marco Robles, Yash Pachori, Prashrit Magar

Student IDs:
923328897, 921282632, 923043313, 922068027

Github Usernames:
ulicessgg, CaliPrerunner, ypachori0, GameaddictXO

GitHub Repository:

https://github.com/CSC415-2024-Fall/csc415-filesystem-ulicessgg.git

Table of Contents

Description:

In this milestone our file system is now capable of detecting a Volume Control Block in order to determine the need for formatting the current volume. If present the Volume Control Block will be loaded alongside the File Allocation Table to keep track of our free space throughout the file system, and the root directory which contains n amount of entries. If no Volume Control Block is present our file system will then initialize an instance with allocated memory as well as create and allocate an instance of our FAT and our root before writing it to memory. Upon completion the file system will then terminate and free the allocated memory for the VCB, FAT, and Root. This can be tested and observed through the use of the hexdump utility which will dump the sample volume to the terminal showing the information for all three of our components of our file system.

Volume Control Block Structure

```
typedef struct volumeControlBlock
{
   uint64_t signature;// Signature to identify the volume // 8 bytes
   uint64_t totalBlocks; // number of blocks in the volume // 8 bytes
   uint64_t blockSize; // Size of each block in bytes // 8 bytes
   unsigned int freeBlockCount; // total free space count // 4 bytes
   unsigned int freeBlockStart; // Start index for free space // 4 bytes
   unsigned int fatSize; // size of the file allocation table // 4 bytes
   unsigned int rootLoc; // Index for the root directory // 4 bytes
   char sysType[24]; // Holds the type of volume/file system // 32 bytes
} volumeControlBlock;
```

Our implementation of the Volume Control Block works with an identifying signature which is used when creating and or loading the VCB from memory. It contains the amount of blocks it is made up of as well as the size of each block in terms of bytes. Besides this it also contains metadata regarding the location of the free space specially the beginning of it, the quantity of free space, the size of our manager of choice in this case a FAT, and the location of the root directory in terms of bytes. Lastly it has a description of its given system type which helps those unaware of our file system but also to identify it in our hexdump.

FAT / Free Space Tracking

```
extern int* FAT;
FAT[0] = -1;
FAT[1] = -2;
```

Our implementation of free space tracking makes use of a File Allocation table which is created globally for use system-wide through the process of declaring it as an external variable. It is an integer pointer and its first two values are set and reserved for the VCB and the FAT itself. Free space will be marked with the value of 1 for now until files are implemented and anything that is occupied will be set with values based on the number of blocks occupied by a given file or directory. This is managed through the use of helper functions that keep track of the FAT values as well as help allocate memory for directories.

Directory Entry Structure

```
typedef struct dir_Entry
{
   time_t creation_date; // Saves the creation timestamp, 8 bytes
   time_t last_edited; // When edited saves the timestamp, 8 bytes
   time_t last_accessed; // When accessed saves the timestamp, 8 bytes
   unsigned int blockPos; // Holds index of the entry in blocks, 4 bytes
   unsigned int size; // Stores the size in bytes of the file, 4 bytes
   int is_Directory; // Identifies a directory or file, 4 byte
   char name[60]; // Name of the file or directory being created, 60 bytes
} dir_Entry;
```

Our implementation of a Directory Entry works with the use of three timestamp values in order to provide users details on the creation, last edit made, and the last time a file or directory has been accessed. A directory entry also contains a block index for its respective position in our file system along with its size. In order to differentiate between a file and a directory an integer value was created that will be checked based on its set value, if 0 then it is a file, otherwise it is a directory. Lastly, in order to identify directory entries each has a name that can be set upon creation with a maximum length of 60 characters.

Approach:

In order to begin implementing a working File System we plan on starting by creating global instances of the most important portions of our file system, the Volume Control Block, the File Allocation Table (FAT) for free space tracking, and our root directory. This will ensure easier workflow between files as well as cut down on dependencies for local variables as this can lead to unfortunate mistakes that can make or break our file system. We intended on doing this by splitting this phase of the file system into 5 steps. The first being the formatting of the Volume by detecting the presence of the Volume Control Block in our sample volume. The second step will be if there is no volume control block present then we will create an instance of it and initialize before writing it to disk. The third step will be the creation and initialization of our FAT which will occur if no FAT is present on disk when loaded. The fourth step will be the implementation of free space management through the use of the FAT in order to implement the creation of directories, this will be in the form an allocateBlocks function which will be used for directory creation in order to allocate the memory needed for the creation of a directory based on the number of entries present in it. Our fifth and final step will be the implementation of directory creation which we intend to support the use of both root and non root directories depending on the parent passed into the function. Once this is created we will then test our file system through the use of a hexdump function in order to ensure the creation of our VCB, FAT, and Root.

Volume Formatting & Initialization

In order to implement the Volume Control Block into our File System we intend to allocate the memory needed for it and upon confirmation of successful allocation we will then read a block from disk and if a VCB instance is already present we will then load our FAT and Root from disk as well. This will be done through the use of our volume signature which is set to the first 18 digits of Pi. If the signature is not detected we will then begin to initialize the instance of the VCB we have created. In order to properly initialize all the values of the VCB we will create our FAT and Root prior to initialization as this is dependent on the location of our free space, the size of it, and the location of our root directory. Once this is completed we plan to then finally initialize all the values of the VCB and then write the VCB, FAT, and Root to disk.

FAT Creation/ Free Space Initialization

The File Allocation Table (FAT) is represented as an array of integers. Each index corresponds to a block on the disk. All the blocks are set to `-1` to indicate that they are free. When the file system is initialized, the FAT is allocated to a size that is equal to the total number of blocks with the first block being reserved for the Volume Control Block (VCB). While blocks are being allocated, the `allocateBlock` function iterates through the FAT to find a free block and will mark it with a special value and then decrement the count of free blocks in the VCB. For single block allocation, that value is `-2`. For multi-block allocation, the `allocateBlocksForFile` function links consecutive blocks in the FAT and establishes a chain that signifies the blocks being used for a particular file. The last block is marked `-1` to indicate the end of the allocated space.

Directory Entry Creation and Root Directory Initialization

To implement a function for initializing and writing a root directory in a file system, we start by defining the root directory to point to itself and pass NULL as the parent directory. The function needs to track the parent directory and the number of entries. First, we calculate the bytes needed by multiplying the number of directory entries by the size of each entry (size of the directory entry structure), which will tell us the bytes required for all entries. Then, we calculate the number of blocks needed to store this data on disk using the formula (bytes_needed + (block_size - 1)) / block_size. This calculation tells us how many blocks we're going to use. To prevent wasted space we have to recalculate the total number of bytes we are going to use and divide that number by how big the directory entries are to determine if we can fit in more directory entries within the blocks allocated to us. With these values, memory can be allocated for the directory entries using malloc. A helper function, allocate_blocks, finds the block positions on disk which tell us the starting location for the root directory. The entries are then initialized, starting with the root entry labeled . and set it with its own location as the parent if no parent is provided. We label the second entry as .., and, if the parent is null, assign its location to the root entry. Each entry is initialized with metadata such as creation_date, last_edited, last_accessed, and a directory flag. Finally, the directory data is written to disk using LABwrite, passing the allocated buffer to complete the disk write process.

Issues and Resolutions:

Although this milestone of our File System Project is straightforward one of the biggest issues we faced was the division of the work as the steps themselves are very much intertwined. For instance the VCB creation can be partially done but is dependent on the FAT and Root being created. The same occurred with the creation of the Root as it relies on a helper function that manages the FAT in order to allocate a block for the directory being created.

This was solved mostly through consistent communication as a group since in order for us to ensure the code we wrote worked we had to update each other based on our completion of our given task. This primarily was the case for the creation of directories and the FAT as they were created alongside one another. This was mediated by having both comment out dependencies until they were fully implemented and then uncommented them and debugged in order to mount them.

Another issue we encountered came about when we moved code regarding the FAT allocation. We decided to move code from our fsInit.c file to fsFAT.c file to keep coherence in our code. Doing so we encountered an error telling us that the function that we were calling fsInit.c to initialize the FAT was not able to be found.

To solve this we looked at how the make file was compiling our program. We then realized it was not compiling the new fsFAT.c to the fsFAT.o file. We then edited the makefile to include this new file along with fsDirEnt., which solved our problem.

Lastly one of the biggest issues we ran into was the creation of the FAT. Due to our misunderstanding of how it would be created we created a node type struct to use for a linked list however this not only failed but led to memory allocation errors resulting in segmentation faults.

After reconfiguring our work and removing the FAT it was reimplemented through the use of an int* which was then supported by helper functions which worked and was able to support the creation of directories from then on as well.

Hexdump Analysis:

VCB Block - 0x200 - 0x3F0

```
000200: FB A0 9E F6 2F 1E 5C 04  4B 4C 00 00 00 00 00 00 | ....../.\.KL......
000210: 00 02 00 00 00 00 00 00  4A 4C 00 00 00 00 00 00 | ........JL......
000220: 4A 4C 00 00 02 00 00 00  46 69 6C 65 20 41 6C 6C | JL......File All
000230: 6F 63 61 74 69 6F 6E 20  54 61 62 6C 65 00 00 00 | ocation Table...
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 | ................
```

Bytes address 0x200-0x207 represent the VCB block signature, highlighted in Yellow. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x045C1E2FF69EA0FB which represents: 314159265358979323 in decimal, this is our VCB signature.

Bytes address 0x208-0x20F represent the total number of blocks, highlighted in Orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000000004C4B which represents: 19531 in decimal. Meaning we have 19,531 blocks in our volume.

Bytes address 0x210-0x217 represent the size of each block, highlighted in Blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000000000200 which represents: 512 in decimal, the number of bytes in one block.

Bytes address 0x218-0x21B represent the count of free block, highlighted in Purple. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00004C4A which as an unsigned int represents: 19530 in decimal, the total number of free blocks in our volume.

Bytes address 0x21C-0x21F represent the total number of blocks, highlighted in Pink. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000 which as an unsigned int represents: 0 in decimal, index of the first free block in the volume.

Bytes address 0x220-0x223 represent the size of the file allocation table (FAT), highlighted in Red. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00004C4A which as an unsigned int represents: 19530 in decimal, index of the first free block in the volume.

Bytes address 0x224-0x227 represent the index of the root directory, highlighted in Green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002 which as an unsigned int represents: 2 in decimal, index of the root directory.

Bytes address 0x228-0x23F represent the name of the volume, highlighted in Dark Blue. Since this data is stored linearly we can read it from left to right. We get: 0x46696C6520416C6C6F636174696F6E205461626C65000000 which when converted to ASCII we get: “File Allocation Table”.

Bytes address 0x240-0x3FF represent the empty unused space in the volume, highlighted in Light Blue. Because each of our block size in our file system is 512 bytes the VCB is stored in the first 64 bytes of this block and the rest is unused space.

FAT Table

```
000400: FF FF FF FF FE FF FF FF  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: 00 00 00 00 00 00 00 00  11 02 00 00 00 00 00 00 | ................
```

Byte addresses 0x400 - 0x407 in hex is 0xFFFFFFFFFEFFFFFF which equal to -1, telling us that the first block is used.

Byte addresses 0x408 - 0x40B in hex is 0xF00000001 which equal to 1, telling us that the first block is free.

There are 126 blocks free because the rest of the byes highlighted in green are all 1 which means that block is free.

Byte addresses 0x4C0 - 0x4C3 and 0x4C4 - 0x4C7 in red are hex values 0x000000 which are 0 and tell us the last blocks in the FAT are being used.

Byte addresses 0x4C8 - 0x4CB in dark green are 0x00000211 which tell us that is the end of the FAT.

Byte addresses 0x4CC - 0x4CF in Pink are padding.

Root Directory

```
000600: B3 4F 24 67 00 00 00 00  B3 4F 24 67 00 00 00 00 | ♦O$g....♦O$g....
000610: B3 4F 24 67 00 00 00 00  02 00 00 00 C0 03 00 00 | ♦O$g........♦...
000620: 01 00 00 00 2E 2E 00 00  00 00 00 00 00 00 00 00 | ................
000630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000640: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x600-0x607 represent the name creation date of the this particular directory entry, highlighted in light blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000067244FB3, since this data is not able to translate to a int to get the raw data we are unable to translate the hex.

Bytes address 0x608-0x60F represent the name last edited date of the this particular directory entry, highlighted in dark blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000067244FB3, since this data is not able to translate to a int to get the raw data we are unable to translate the hex.

Bytes address 0x610-0x617 represent the name last accessed date of the this particular directory entry, highlighted in green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000067244FB3, since this data is not able to translate to a int to get the raw data we are unable to translate the hex.

Bytes address 0x618-0x61B represent the name last edited date of the this particular directory entry, highlighted in light orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002, when translated to decimal: 2, which means it is in the second block.

Bytes address 0x61B-0x61F represent the block position of the this particular directory entry, highlighted in purple Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x000003C0, when translated to decimal: 920, meaning the directory entry is 920 bytes big.

Bytes address 0x620-0x623 represent the size of the this particular directory entry, highlighted in red Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000001, when translated to decimal: 1, meaning the directory entry is a directory.

Bytes address 0x624-0x65F represent the name of the this particular directory entry, highlighted in yellow. The hex value of this data is 0x2E2E00000000...., when translated to ASCII: "..', meaning the .. of the root directory.

000660: B3 4F 24 67 00 00 00 00 B3 4F 24 67 00 00 00 00 | ◆O$g....◆O$g....

000670: B3 4F 24 67 00 00 00 00 02 00 00 00 C0 03 00 00 | ◆O$g........◆...

```
000680: 01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x660-0x667 represent the name creation date of this particular directory entry, highlighted in light blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x668-0x66F represent the name last edited date of this particular directory entry, highlighted in dark blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x670-0x677 represent the name last accessed date of this particular directory entry, highlighted in green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x678-0x67B represent the block position date of this particular directory entry, highlighted in light orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002, when translated to decimal: 2, which means it is in the second block.

Bytes address 0x67C-0x67F represent the size of this particular directory entry, highlighted in purple. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x000003C0, when translated to decimal: 920, meaning the directory entry is 920 bytes big.

Bytes address 0x680-0x683 represent the type of this particular directory entry, highlighted in red. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000001, when translated to decimal: 1, meaning the directory entry is a directory.

Bytes address 0x684-0x6BF represent the name of this particular directory entry, highlighted in yellow. The hex value of this data is 0x000000000000...., when translated to ASCII is nothing, meaning the second entry of the root directory.

```
0006C0: B3 4F 24 67 00 00 00 00  B3 4F 24 67 00 00 00 00 | ◆O$g....◆O$g....
0006D0: B3 4F 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ◆O$g....◆◆◆◆....
0006E0: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
0006F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000700: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x6C0-0x6C7 represent the name creation date of this particular directory entry, highlighted in light blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x6C8-0x6CF represent the name last edited date of this particular directory entry, highlighted in dark blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x6D0-0x6D7 represent the name last accessed date of this particular directory entry, highlighted in green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x6D8-0x6DB represent the block position date of this particular directory entry, highlighted in light orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0xFFFFFFFF, when translated to decimal: -1.

Bytes address 0x6DC-0x6F represent the size of this particular directory entry, highlighted in purple. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000, when translated to decimal: 0.

Bytes address 0x6E0-0x6E3 represent the type of this particular directory entry, highlighted in red. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000, when translated to decimal: 1, meaning the directory entry is a file.

Bytes address 0x6E4-0x71F represent the name of this particular directory entry, highlighted in yellow. The hex value of this data is 0x6E756C6C46696C65000000...., when translated to ASCII: "nullFile", giving us the name of this directory entry.

Eight of these directory entries are created because even though we specified 6 entries, we had to allocated blocks to write them on disk of size 512. To use up all the space allocated in the blocks for the disk we were able to create two more, totaling it to eight directory entries.

Communication and Coordination:

In order to try and coordinate with each other the best we can we have been making use and relying on our discord channel where we communicate frequently throughout a given week in order to update each other on progress as well as check in if our work is dependent on someone else. However, due to contrasting work and academic schedules, it has been difficult to meet both in person as well as virtually. We created a when2meet in order to see our overlapping schedules and find time where we can all join a discord call in order to work on a given task or follow up on a topic that was mentioned in our chat. For our first milestone, we have divided the work accordingly as shown below.

Division of work

Ulices Gonzalez	Check if volume is present or not(if so initialize it)
Marco Robles	Create and initialize root directory
Yash Pachori	Initialize the FAT (free space tracking)
Prashrit Magar	Analyzed hexdump
Everyone	Debugging and Write Up

System Hexdump:

```
000200: FB A0 9E F6 2F 1E 5C 04  30 00 00 00 00 00 00 00 | ****/.\.0.......
000210: 00 02 00 00 00 00 00 00  2F 00 00 00 00 00 00 00 | ......../.......
000220: 2F 00 00 00 02 00 00 00  46 69 6C 65 20 41 6C 6C | /.......File All
000230: 6F 63 61 74 69 6F 6E 20  54 61 62 6C 65 00 00 00 | ocation Table...
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 | ................

000400: FF FF FF FF FE FF FF FF  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 | ................
```

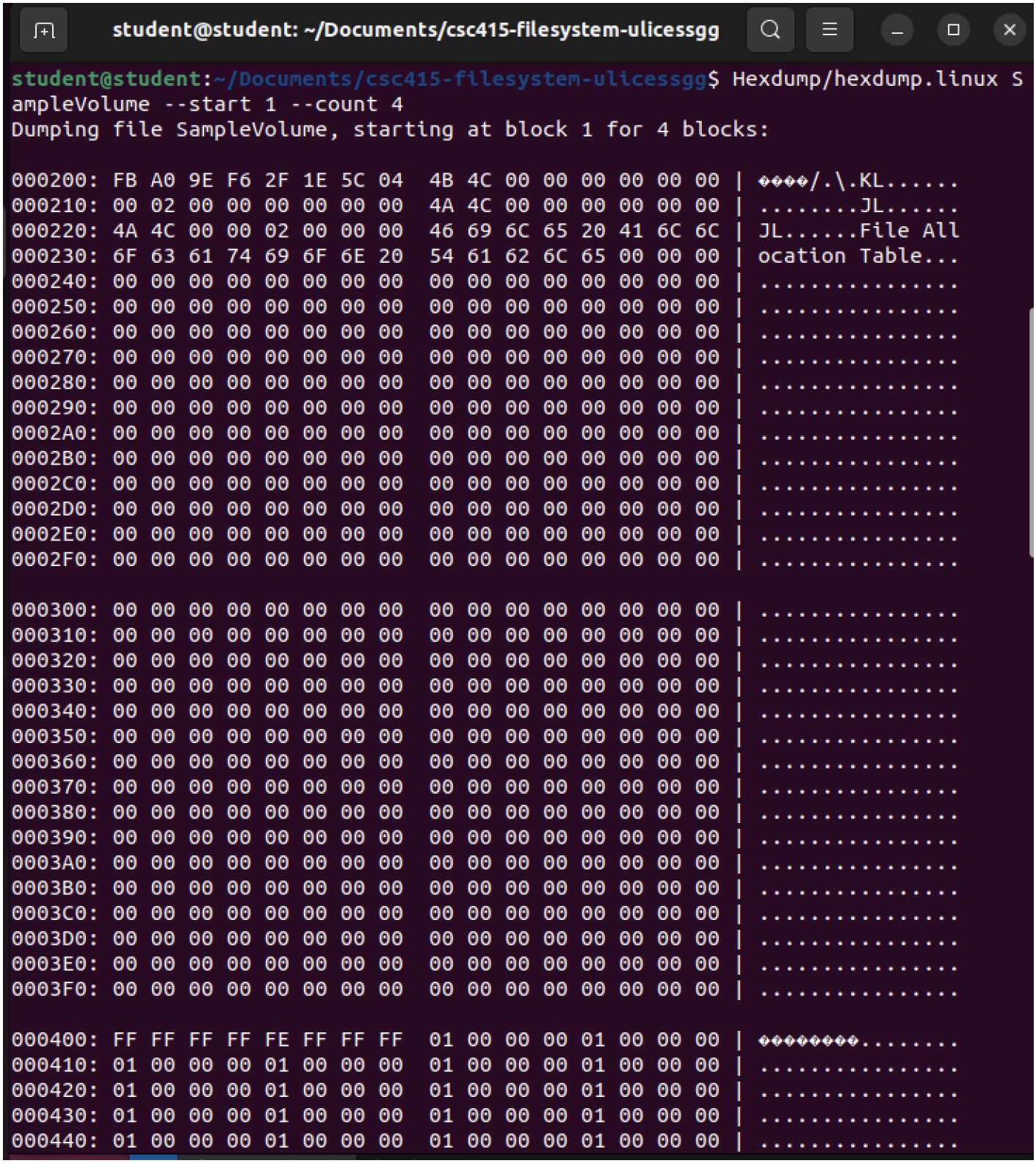
```
000430: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000440: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000450: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
000460: 01 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00 | ................
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: 00 00 00 00 00 00 00 00  11 02 00 00 00 00 00 00 | ................
0004D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000500: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000510: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000520: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000530: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000540: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000550: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000560: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000570: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000580: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000590: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0005A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0005B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0005C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0005D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0005E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0005F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000600: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ◆W$g....◆W$g....
000610: D3 57 24 67 00 00 00 00  02 00 00 00 C0 03 00 00 | ◆W$g........◆...
000620: 01 00 00 00 2E 2E 00 00  00 00 00 00 00 00 00 00 | ................
000630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000640: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000660: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ◆W$g....◆W$g....
000670: D3 57 24 67 00 00 00 00  02 00 00 00 C0 03 00 00 | ◆W$g........◆...
```

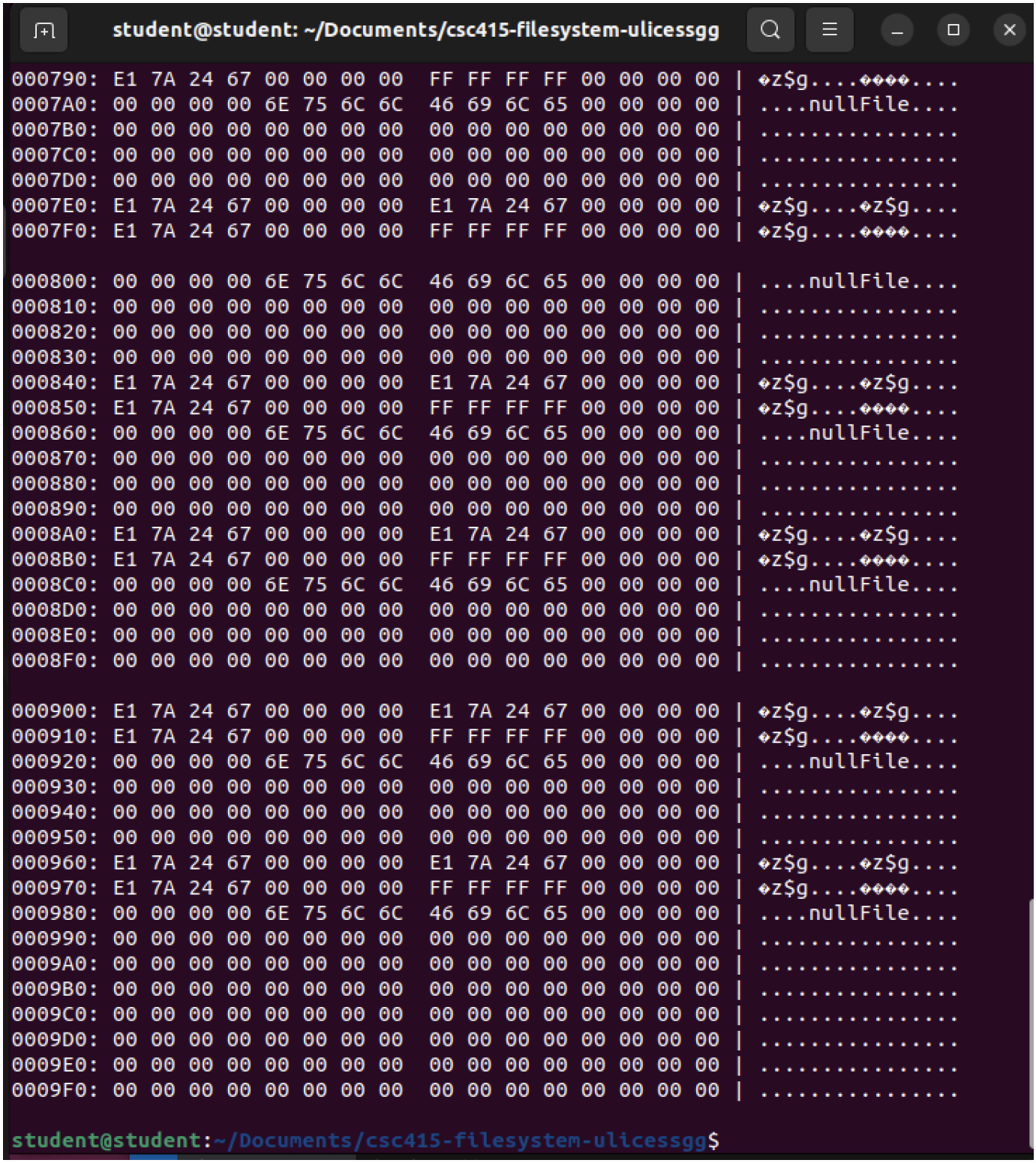
```
000680: 01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006C0: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ♦W$g....♦W$g....
0006D0: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ♦W$g....♦♦♦♦....
0006E0: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
0006F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000700: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000720: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ♦W$g....♦W$g....
000730: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ♦W$g....♦♦♦♦....
000740: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
000750: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000760: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000770: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000780: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ♦W$g....♦W$g....
000790: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ♦W$g....♦♦♦♦....
0007A0: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
0007B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007E0: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ♦W$g....♦W$g....
0007F0: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ♦W$g....♦♦♦♦....

000800: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
000810: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000820: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000830: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000840: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ♦W$g....♦W$g....
000850: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ♦W$g....♦♦♦♦....
000860: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
000870: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000880: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000890: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0008A0: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ♦W$g....♦W$g....
0008B0: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ♦W$g....♦♦♦♦....
0008C0: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
```

```
0008D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0008E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0008F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000900: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ◆W$g....◆W$g....
000910: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ◆W$g....◆◆◆◆....
000920: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
000930: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000940: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000950: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000960: D3 57 24 67 00 00 00 00  D3 57 24 67 00 00 00 00 | ◆W$g....◆W$g....
000970: D3 57 24 67 00 00 00 00  FF FF FF FF 00 00 00 00 | ◆W$g....◆◆◆◆....
000980: 00 00 00 00 6E 75 6C 6C  46 69 6C 65 00 00 00 00 | ....nullFile....
000990: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0009A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0009B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0009C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0009D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0009E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0009F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Screenshots:

```
student@student: ~/Documents/csc415-filesystem-ulicessgg
student@student:~/Documents/csc415-filesystem-ulicessgg$ make clean
rm fsshell.o fsInit.o fsFAT.o fsDirEnt.o fsshell
student@student:~/Documents/csc415-filesystem-ulicessgg$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
fsInit.c: In function ‘initFileSystem’:
fsInit.c:96:9: warning: implicit declaration of function ‘initFAT’ [-Wimplicit-f
unction-declaration]
   96 |         initFAT(numberOfBlocks);
      |         ^~~~~~~
gcc -c -o fsFAT.o fsFAT.c -g -I.
gcc -c -o fsDirEnt.o fsDirEnt.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsFAT.o fsDirEnt.o fsLow.o -g -I. -lm -l readl
ine -l pthread
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume Control Block not Present!
FAT array initialized with 19531 blocks.
the block of the root is at 2Finished initializing VCB and FAT!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    OFF   |
| cd                   |    OFF   |
| md                   |    OFF   |
| pwd                  |    OFF   |
| touch                |    OFF   |
| cat                  |    OFF   |
| rm                   |    OFF   |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    OFF   |
| cp2l                 |    OFF   |
|---------------------------------|
make: *** [Makefile:67: run] Interrupt

student@student:~/Documents/csc415-filesystem-ulicessgg$ Hexdump/hexdump.linux S
ampleVolume --start 1 --count 4
Dumping file SampleVolume, starting at block 1 for 4 blocks:
```


student@student: ~/Documents/csc415-filesystem-ulicessgg
student@student:~/Documents/csc415-filesystem-ulicessgg$ Hexdump/hexdump.linux SampleVolume --start 1 --count 4
Dumping file SampleVolume, starting at block 1 for 4 blocks:
000200: FB A0 9E F6 2F 1E 5C 04 4B 4C 00 00 00 00 00 00 | ◆◆◆◆/.\.KL......
000210: 00 02 00 00 00 00 00 00 4A 4C 00 00 00 00 00 00 | ....JL......
000220: 4A 4C 00 00 02 00 00 00 46 69 6C 65 20 41 6C 6C | JL......File All
000230: 6F 63 61 74 69 6F 6E 20 54 61 62 6C 65 00 00 00 | ocation Table...
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 | ....
000400: FF FF FF FF FE FF FF FF 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 | ....


student@student: ~/Documents/csc415-filesystem-ulicessgg
000790: E1 7A 24 67 00 00 00 00 FF FF FF FF 00 00 00 00 | ◆z$g....◆◆◆◆....
0007A0: 00 00 00 00 6E 75 6C 6C 46 69 6C 65 00 00 00 00 | ....nullFile....
0007B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0007C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0007D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0007E0: E1 7A 24 67 00 00 00 00 E1 7A 24 67 00 00 00 00 | ◆z$g....◆z$g....
0007F0: E1 7A 24 67 00 00 00 00 FF FF FF FF 00 00 00 00 | ◆z$g....◆◆◆◆....
000800: 00 00 00 00 6E 75 6C 6C 46 69 6C 65 00 00 00 00 | ....nullFile....
000810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000820: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000830: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000840: E1 7A 24 67 00 00 00 00 E1 7A 24 67 00 00 00 00 | ◆z$g....◆z$g....
000850: E1 7A 24 67 00 00 00 00 FF FF FF FF 00 00 00 00 | ◆z$g....◆◆◆◆....
000860: 00 00 00 00 6E 75 6C 6C 46 69 6C 65 00 00 00 00 | ....nullFile....
000870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000890: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0008A0: E1 7A 24 67 00 00 00 00 E1 7A 24 67 00 00 00 00 | ◆z$g....◆z$g....
0008B0: E1 7A 24 67 00 00 00 00 FF FF FF FF 00 00 00 00 | ◆z$g....◆◆◆◆....
0008C0: 00 00 00 00 6E 75 6C 6C 46 69 6C 65 00 00 00 00 | ....nullFile....
0008D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0008E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0008F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000900: E1 7A 24 67 00 00 00 00 E1 7A 24 67 00 00 00 00 | ◆z$g....◆z$g....
000910: E1 7A 24 67 00 00 00 00 FF FF FF FF 00 00 00 00 | ◆z$g....◆◆◆◆....
000920: 00 00 00 00 6E 75 6C 6C 46 69 6C 65 00 00 00 00 | ....nullFile....
000930: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000940: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000950: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
000960: E1 7A 24 67 00 00 00 00 E1 7A 24 67 00 00 00 00 | ◆z$g....◆z$g....
000970: E1 7A 24 67 00 00 00 00 FF FF FF FF 00 00 00 00 | ◆z$g....◆◆◆◆....
000980: 00 00 00 00 6E 75 6C 6C 46 69 6C 65 00 00 00 00 | ....nullFile....
000990: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0009A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0009B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0009C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0009D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0009E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0009F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
student@student:~/Documents/csc415-filesystem-ulicessgg$