CSC 415.03 File System Project

Group Name:
Geek Squad

The GitHub Repository Name That We'd Like You To Grade:
EssaHusary

GitHub Repository Link:
CSC415-2023-Spring/csc415-filesystem-EssaHusary: csc415-filesystem-EssaHusary created by GitHub Classroom

Group Members:

Name	Student ID
Mahdi Hassanpour	922278744
Essa Husary	917014896
Jinwei Tang	922350439
Amit Bal	922832232

*****NOTE: Our references are on the final page of this document where we've cited our sources as well as explained what we referenced and from which source(s).**

Description:

****We will first provide a general description of our file system, and then we will go more into depth about each component of our file system such as our volume control block, the free space, the directory system, parsepath, the directory functions, the b_io functions, and last but not least, a description of our driver, fsshell.

General Description:

In this assignment, we are implementing our very own file system. We are the decision makers of this file system and everything, from the volume control block, the free space system, the directory system, etc., are implemented based off of the tools and techniques discussed in class and in the textbook. We utilize a “SampleVolume”, which is effectively a “hard disk”, allowing us to test our system. As we learned from the textbook and in class, file systems are typically layered. We learned of six layers, and two of these layers, the “logical file system” and the “file-organization module” are the focus of our file system for this project. The directory system (found in “directories.h and .c”), as well as our directoryEntry struct and metadata (found in “volumecontrol.h”), are based around the “logical file system” layer. The free space management system (found in “freespace.h and .c”) is based around the other layer, the “file-organization module”. The functions LBAwrite and LBAread, not implemented by us, are based around the “basic file system” layer, where all that’s done is, well, the reading and writing of blocks of data from/to disk.

Before we start describing the different components of our file system, it would be wise and necessary to define certain terms. We should first define and explain what we mean by a “block” of data. Quite simply, a block is a chunk of data of a predefined number of bytes. It’s neither convenient nor easy to work with individual bytes, so blocks are used throughout this project in order to work with data - allocation, deallocation, and reading and writing - as this is more efficient. The typical size of blocks is 512 bytes, which is something we can see in the Makefile’s default RUNOPTIONS. Next is the term “volume”. A volume is a formatted storage device (formatted meaning prepared for use, essentially). So the “SampleVolume”, generated by the “partition” system in our project, is exactly that, a storage device, effectively a disk, that is prepared to store our file system. According to the textbook, a “partition” is a, "Logical segregation of storage space into multiple area…" (Silberschatz, et. al.). So one can think of it as a logical

section in the volume. The “volume control block” is a little self explanatory at this point given this explanation. It is a block of data which stores details about the volume, such as the size of each block, the number of free blocks, the number of total blocks, etc., etc.. Free blocks are managed by the “free space management” system, a system which manages how many (and which) blocks are available for use, available to be allocated, according to an “allocation method”. An “allocation method” is a way, a method, of providing resources, in our case, blocks of data, in our file system. Our “allocation method” of choice is the use of “extents”, entries that contain information about the starting block number of a “directory entry” (will be defined shortly) in the “logical block array” and a number of contiguous blocks, a count, that this entry goes for in the “logical block array” from this point. A “logical block array” is a logical/high-level way of thinking about how “files” are stored on disk (since things can be stored all over the place on disk). Free space management systems maintain “free-space lists”, which, according to the textbook, are, “...data structures tracking all free blocks in the file system” (Silberschatz, et. al.). Our “free-space list” of choice was a bitmap, where Essa implemented it to be an array of unsigned chars where bits in each byte were manipulated. “Files”, “directories”, and “directory entries” are blobs of data, blobs of bytes, blobs of information. A “directory”, therefore, is an array of these “directory entries”, of these blobs of data. Every directory is a ‘descendant’, so-to-speak, of the “root directory”, which is essentially the proverbial ‘grandfather’ directory that our file system begins working through when performing/calling directory functions. So having defined these terms, it is now appropriate to describe and discuss each component. (Silberschatz, et. al. [textnote for this entire “General Description”, as the textbook and class lectures were referenced to write it])

<u>Description of our Volume Control Block:</u>

Before we discuss our volume control block, it would be appropriate and wise to provide the struct for the volume control block found in volumecontrol.h and explain its contents. For your convenience, Professor, here is the struct of our volume control block that you can reference while we describe it to you:

```
// The struct for our volume control block
struct VCB {

	// The size of each block in the partition
	int sizeOfBlocks;

	// The number of total blocks in the partition
	int numOfTotalBlocks;

	// The number of free blocks in the partition
	int numOfFreeBlocks;

	// The type/signature of the volume
	u_int64_t magicNumber;

	// The starting block number of VCB
	int startingBlockOfVCB;

	// The starting block number of the root directory.
	int startingBlockOfRootDirectory;

	// The length of the root directory
	int lengthOfRootDirectory;

	// The starting block # of our free space map
	int startingBlockOfFreeSpace;

	// The length of the free space map
	int lengthOfFreeSpaceMap;

} VCB;
```

Our volume control block structure begins at block 0 in the logical block array (with a size of obviously one block) and as said previously in the “General Description”, the volume control block contains all of the information necessary to access and control various things in our volume. We have, for instance, stored the size of each block of the volume in a variable called “sizeOfBlocks”, which is by default 512 bytes (default by the

Makefile's RUNOPTIONS). This was useful when doing things such as creating the free space map and was useful when performing allocations using malloc(). We also have the variable "numOfTotalBlocks" which stores the total number of blocks in the partition, which is by default 19,531 blocks. It came in handy when creating the free space map and was used in the calculation to determine how many blocks are required for our free space map [(m + (n - 1)) / n]. Also, we have a variable called "numOfFreeBlocks", which tracks how many free blocks exist in the free space bitmap. We can "query" this data for multiple reasons and purposes. Another useful piece of information in our VCB is the "magicNumber" variable. This is an essential piece of info as it allows us to determine whether or not our volume has already been formatted. The integer value of the magic number is 39,106,557,589,782 (without commas of course) and its hex value is 0x0000 2391 3479 3916. What's more, we have the variables "startingBlockOfVCB", "startingBlockOfFreeSpace", "lengthOfFreeSpaceMap", "startingBlockOfRootDirectory", and "lengthOfRootDirectory", which, as their names suggest, store the starting block number of the VCB, the starting block number of the free space map, the length of the free space map, the starting block number of the root directory, and the length of the root directory, respectively. Now we can have all kinds of uses for such information, but the most basic uses for these pieces of information involve using them in calls to LBAread() and LBAwrite() in order to bring these entities from disk into memory and vice versa. Based on the default RUNOPTIONS, the values for these fields in order of how I just listed them are 0, 1, 5, 6, and 13, respectively.

The volume control block's fields are accessed via the global variable "volumeControlBlock", which we declared in volumecontrol.c, making life easier for us when performing calculations using the default "blockSize" of 512 bytes, when accessing the length of the root directory and the free space map to use in calls to LBAread and LBAwrite, as well as providing other uses throughout this project. In this file, volumecontrol.h, we store the struct for the volume control block, the directoryEntry and ExtendNode structs (will be described in our discussion of the directory system), and #defines for various purposes including "MAGIC_NUMBER", which is the type or signature of our volume, "NUM_ENTRIES", the initial number of entries requested for all directories, "NUM_OF_EXTENTS", the number of extents we want for a directory entry, and "MINIMUM_NUMBER_OF_EXTENT_BLOCKS", the minimum number of blocks to ask the free space for our extents (the latter three #defines will be explained later on in the next sections). We also have in this file, function prototypes for the volumeControlInit() function, which, in the volumecontrol.c file, initializes some of the fields of the volume control block, and the extentsWrite() function, which, also in the volumecontrol.c file, iterates across a directoryEntry's extents table to write a directoryEntry to disk.

Here is the hexdump of the volume control block:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 1 --start 1
Dumping file SampleVolume, starting at block 1 for 1 block:

000200: 00 02 00 00 4B 4C 00 00  38 4C 00 00 00 00 00 00 | ....KL..8L......
000210: 16 39 79 34 91 23 00 00  00 00 00 00 06 00 00 00 | .9y4◆#..........
000220: 0D 00 00 00 01 00 00 00  05 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 | ................
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 | ................

student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Analysis of the dump:

1. Addresses 000 200 - 000 203, highlighted yellow, contain the value 00 00 02 00 (reversed from little endian). In decimal, this value is 512 which is the field in our

VCB called “sizeOfBlocks” which is of type int and stores the size of the blocks of the volume.

2. Addresses 000 204 - 000 207, highlighted orange, contain the value 00 00 4C 4B (reversed from little endian). In decimal, this value is 19,531 which is the field in our VCB called “numOfTotalBlocks” of type int and stores the number of blocks in the volume.

3. Addresses 000 208 - 000 20B, highlighted light green, contain the value 00 00 4C 38 (reversed from little endian). In decimal, this value is 19,512 which is the field in our VCB called “numOfFreeBlocks” of type int and stores the number of free blocks in the volume. ***The next four bytes after it is just padding so that’s why they’re not highlighted.**

4. Addresses 000 210 - 000 217, highlighted a darker green, contain the value 00 00 23 91 34 79 39 16 (reversed from little endian). In decimal, this value is 39,106,557,589,782 which is the field in our VCB called “magicNumber” of type u_int64_t (since it’s so large) and stores the magic number for the volume.

5. Addresses 000 218 - 000 21B, highlighted red, contain the value 00 00 00 00 (reversed from little endian). In decimal, this value is 0 which is the field in our VCB called “startingBlockOfVCB” of type int and stores the starting block of the VCB.

6. Addresses 000 21C - 000 21F, highlighted blue, contain the value 00 00 00 06 (reversed from little endian). In decimal, this value is 6 which is the field in our VCB called “startingBlockOfRootDirectory” of type int and stores the starting block of the root directory.

7. Addresses 000 220 - 000 223, highlighted a peachy color, contain the value 00 00 00 0D (reversed from little endian). In decimal, this value is 13 which is the field in our VCB called “lengthOfRootDirectory” of type int and stores the length of the root directory.

8. Addresses 000 224 - 000 227, highlighted purple, contain the value 00 00 00 01 (reversed from little endian). In decimal, this value is 1 which is the field in our VCB called “startingBlockOfFreeSpace” of type int and stores the starting block of the free space map.

9. Addresses 000 228 - 000 22B, highlighted off-white, contain the value 00 00 00 05 (reversed from little endian). In decimal, this value is 5 which is the field in our VCB called “lengthOfFreeSpaceMap” of type int and stores the length of the free space map.

The rest of the values are 00 as those bytes are just the result of mallocing space that’s more than enough (but appropriate) to contain the VCB.

Description of the Free Space Management System:

Our free space map begins at block 1 in the logical block array and goes for 5 blocks in that LBA. Our free space management system relies on and revolves around two major things: a free-space list and an allocation method. Our free-space list of choice, as mentioned in the “General Description”, is a bitmap. Essa implemented it such that the map was an array of unsigned chars, as using this data type would make things a lot easier than if we were to use an array of type int or short, where we’d have to perform more math to alter and access a target byte. We have defined 1 to indicate that a block is free and 0 to indicate that the block is in use. The free space map begins at block number 1 in the logical block array, and in the default RUNOPTIONS settings in the Makefile where “volumeSize” is 10,000,000 bytes and “blockSize” is 512 bytes, the length of our free space map is 5. So our free space map in this case spans 5 blocks starting from block number 1. Before initializing the bitmap, we had 19,531 bits each with a value of 1. Since the volume control block, free space map, and root directory are now written to disk, the first 19 blocks are 0s because they are now marked used. So the first three bytes of the bitmap have been altered and the free space map as a result now looks like this (well at least the first three bytes): 0b00000000 00000000 00011111 (or 0x00 00 1F in hex). Our allocation method of choice, as mentioned in the “General Description”, is extents, where we have a starting block number into the logical block array and a count of how many contiguous blocks a directory entry goes for in the logical block array. Each directory entry has a maximum of 8 extents to utilize. This was established by creating a #define called “NUM_OF_EXTENTS” in volumecontrol.h.

Now the question arises, how did Essa manage to create the free space management system in freespace.c (function prototypes in freespace.h) which considers and utilizes extents? Well first, he had to initialize the bitmap. The functions setBit(), clearBit(), and initializeMap() had to be written first and foremost. SetBit and clearBit were inspired by the class lecture where you, Professor, gave us ideas and tips as to how to manipulate bits. SetBit sets bits to values of 1, indicating that a block is free, and clearBit does the opposite, setting bits to values of 0, indicating that a block is in use. There is a function called “clearMultipleBits” which calls clearBit in a ‘for’ loop. As said previously, Essa chose to use as a free-space list an array of unsigned chars, pointed to by a pointer variable aptly named “freeSpaceMap”, declared in “freespace.c”. The bytes of the malloced space pointed to by “freeSpaceMap” were manipulated using setBit and clearBit. The function initializeMap() creates the entire bitmap in the malloced space by iterating through every single one of those 19,531 bits, setting them all to values of 1. Essa also wrote a function called getBlocks() which iterates across the entire bitmap and checks if we are able to find a desired number of contiguous/consecutive bits with values of 1, and if so, getBlocks returns the starting block number in that consecutive/contiguous chain. The function freeSpaceInit(), called in fsInit.c’s initFileSystem() function, calls all of the aforementioned functions to initialize the entire free space, setting all of the bits to values of 1 and clearing the bits/blocks associated with the VCB and the free space itself considering that they now occupy space. So before the root directory is created, the first byte of freeSpaceMap is altered and has a value of 0b00000011 in binary and a value of 0x03 in hex. The bits/blocks associated with the root directory are cleared a bit later in initFileSystem() when the directory system is utilized to create the root directory.

Our allocation method of choice, as stated previously, is extents. There are two functions in freespace.c which allow us to allocate blocks and fill a directory entry’s extent table contained in the directoryEntry struct: blockAllocator() and getNextBit(). BlockAllocator essentially has an extents table passed into it as a parameter and sees if first of all, can the caller’s desired number of contiguous/consecutive 1s be found using getBlocks()? If so, we fill the first extent with the starting block number of where that occurs and set the count to simply be the desired number of blocks, as we found that desired number in a contiguous fashion. But if this is not the case, we must iterate through the extents table, finding a starting block of “MINIMUM_NUMBER_OF_EXTENT_BLOCKS” (which is a #define found in volumecontrol.h, set to value of 3) number of consecutive 1s and a count of consecutive 1s after this initial number of consecutive 1s using the function, getNextBit. GetNextBit returns a count of how many consecutive 1s occur after a certain position. The reason we want this #define number of blocks initially, is so we can give enough blocks to the

caller and so we don’t run the risk of filling in the extents table too much too soon. We want at least this #define number of blocks per extent. So we can have for instance, block 20 with 2 consecutive 1s. This isn’t good enough for the value of the #define value we set (which is 3) as we might run out of extents to fill, so we might need to go to starting block 879 that has this #define number of consecutive 1s, and from there, get a count of how many 1s occur after these number of 1s. The return value of getNextBit plus this #define number is the count for an extent. GetNextBit is different from getBlocks(), as getBlocks tries to see if we can provide to the caller a consecutive/contiguous number of blocks of a desired number, whereas getNextBit just checks how many consecutive 1s there after a certain position. The final function that hasn’t been mentioned yet is freeBlocks(), which actually does the opposite of blockAllocator(), freeing the blocks associated with a directoryEntry and setting those blocks in the freeSpaceMap to values of 1, indicating free.

Here is the hexdump of our free space map after the initialization of our file system:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 5 --start 2
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 1F FF FF FF FF FF  FF FF FF FF FF FF FF FF | ...◆◆◆◆◆◆◆◆◆◆◆◆◆
000410: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000420: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000430: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000440: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000450: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000460: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000470: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000480: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000490: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000500: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000510: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000520: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000530: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000540: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000550: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000560: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000570: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000580: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000590: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0005A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0005B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0005C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0005D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0005E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0005F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

```
000600: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000610: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000620: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000630: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000640: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000650: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000660: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000670: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000680: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000690: FF FF FF FF FF FF FF FF  FF FF FF FF FF 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 | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000700: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000710: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000720: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000730: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000740: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000750: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000760: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000770: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000780: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000790: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0007A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0007B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0007C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0007D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0007E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0007F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

```
000800: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000810: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000820: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000830: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000840: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000850: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000860: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000870: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000880: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000890: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0008A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0008B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0008C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0008D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0008E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0008F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000900: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000910: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000920: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000930: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000940: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000950: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000960: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000970: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000980: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000990: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0009A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0009B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0009C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0009D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0009E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0009F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

```
000A00: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A80: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000A90: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000AA0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000AB0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000AC0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000AD0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000AE0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000AF0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000B00: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B80: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000B90: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000BA0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000BB0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000BC0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000BD0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000BE0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000BF0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

```
000C00: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C80: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000C90: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000CA0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000CB0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000CC0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000CD0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000CE0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000CF0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000D00: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D10: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D20: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D70: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000D80: FF FF FF FF FF FF FF FF  FF E0 00 00 00 00 00 00 | ◆◆◆◆◆◆◆◆◆◆.......
000D90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000DA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000DB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000DC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000DD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000DE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000DF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Analysis of the dump:

Addresses 000 400 - 000 402, highlighted yellow, contain the hex values 0x00 00 1F, which in binary is 0b00000000 00000000 00011111. This represents the fact that the VCB, which occupies 1 block, the free space, which occupies 5 blocks, and the root directory, which occupies 13 blocks, are all marked used, with a total of 19 used blocks. We've defined 0 as used and 1 as free. All of the other addresses, from 000 403 - 000 D88 are all free, containing values of FF in hex and 11111111 in binary since we haven't used them yet. Address 000 D89, highlighted orange, contains a value of 0xE0 in hex and 0b11100000 in binary, where the 0s simply mean that because we only care about 19,531 blocks, we don't care to assign the blocks/bits after the final 1 any values. Addresses 000 D8A - 000 DFF, highlighted red, contain values of 0x00 or 0b00000000 in binary, as these bytes are just extra space as a result of mallocing space that's more than enough (but appropriate) to contain the free space map.

Description of our Directory System:

Before we discuss our directory system, it would be appropriate and wise to provide our ExtentNode and directoryEntry structs, found in volumecontrol.h, and the directoryEntryCalculations struct, found in directories.h, and explain their contents. For your convenience, Professor, here are those structs that you can reference while we describe them to you:

```
// Global variables to store calculations for creating a new directory
struct directoryEntryCalculations{

        /* The total number of bytes needed for an inital amount of directory entries,
           in our case, 50 entries */
        int totalBytesForEntries;
        // The number of blocks, or length in the LBA, of the new directory
        int lengthOfDirectory;
        // The number of entries we can still fit in inside the final allocated block
        int totalExtraEntries;
        /* The total number of entries in the new directory, after calculating how many
           more entries we can fill into the allocated space */
        int totalNumOfEntries;

} directoryEntryCalculations;
```

```
// The struct for the extent node
struct ExtentNode{

        // The address of the first block in the extent
        int blockNum;
        //The number of blocks counted after the first block
        int count;
```

```
} ExtentNode;



// The struct for our directory entry
struct directoryEntry {

	// The name of the entry
	char fileName[30];
	// The size of the entry
	int  size;
	// The number of entries in the directory entry
	int numOfEntries;
	// The time the entry was created
	time_t  dateCreated;
	// The time the entrywas modified
	time_t  dateModified;
	// The type of file. 0 = directory, 1 = file
	int fileType;
	// The state of a directory entry. Used = true, unused = false.
	bool used;
	// The eight extent tables
	struct ExtentNode extents[NUM_OF_EXTENTS];

} directoryEntry;
```

Our directory system relies on the above three structs to allow us to create a new directory using mathematical calculations and allows us to determine what kind of metadata we want to have in each directory entry. The system starts in the initDir() function in directories.c, where we begin by performing a series of calculations to see how many bytes, blocks, and entries we'll need to make a new directory. In the "directoryEntryCalculations" struct, we have the "totalBytesForEntries" field. This field is populated in the numBytesForEntries() function in directories.c, storing in it the total number of bytes that we need for our initial desired number of entries, which is 50. This initial number is actually a #define in volumecontrol.h called "NUM_ENTRIES". Then after this value is populated, we want to fill the field "lengthOfDirectory", which is the

number of blocks, or length in the LBA, of the new directory. This is filled in the numBlocksToAskFreeSpace() function allowing us to know the next necessary piece of information, which is ascertaining how many more entries we can store in the space we want to allocate. Next, “totalExtraEntries”, the number of additional entries we can squeeze into the space we want to allocate, is filled in the numExtraEntries() function. Once we have determined how much space we need to allocate, we go ahead and malloc space for the new directory. Then, the function initializeDirFields() is called, where we finally begin filling in the fields for a directory entry.

To initialize the fields of the new directory, we must iterate across each array index, as a directory is an array of directory entries. The initialization of fields can begin in any order, by either filling in the “.” and “..” entries first, or by filling in the entries after “.” and “..” first. We chose to do the latter, as that was done by you, Professor, in class, when you went over Milestone 1 to give us some pointers (no pun intended) and tips to improve our directory system. So we began initializing the entries after “.” and “..”. The first field in our directoryEntry struct is “filename”, which as the field name suggests, is the name that we want to give an entry. Every entry after “.” and “..” has its “filename” initialized to the null terminator, ‘\0’. And then, we fill in the rest of the fields. We fill in and set to values of 0, “size”, the size in bytes of the entry; “numOfEntries”, the total number of entries the directory entry contains (useful if the entry is a directory); “dateCreated” and “dateModified”, the times the entry was created and modified, respectively; and “filetype”, the type of the entry, where 0 = directory and 1 = file. And then, we set “used”, the state of the entry, to a value of false, where if “used” = true, the entry is in use, and if “used” = false, the entry is free to use. Finally, the extents of each of those entries after “.” and “..” are initialized to values of 0.

The initialization of “.” and “..” are a little different. The “.” entry for instance, will have its “filename” be “.”, obviously. Its “size” will be set to the number of bytes in use for the new/current directory, as “.” contains info about the current directory. Its “numOfEntries” will be set to “totalNumOfEntries”, the field from the directoryEntryCalculations struct, the number of entries contained in this directory. The “dateCreated” and “dateModified” fields will be set to the number of secs since January 1, 1979, that this new/current directory was created and/or modified. The “filetype” will be 0, since the “.” entry represents the current directory and its “used” state will obviously be set to true, since it’s in use. And, its extents will first be initialized to values of 0, and then its extents table will be passed into the function “blockAllocator”, called from the free space, to fill in its extent table. We don’t really need to initialize the extents to values of 0 first, but we’ve found that we’ve run into quite a few really nasty bugs when we don’t initialize integer values to values of 0, for some odd reason. As for “..”, since “..” entries in all directories contain information about the *parent* directory, not the

current one, its initialization depends on whether or not it actually has a parent directory. The root directory for instance, does not have a parent. This is indicated when the parameter, “parentDirectory” of the initializeDirFields() function, is NULL. In this case, “..” will copy all of the information contained in the “.” entry of the root directory. In all other cases, when “parentDirectory” is not NULL, every “..” in all other directories will simply contain within them the information that the parent’s “.” entry contains.

Here is the hexdump of our root directory after the initialization of our file system:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 13 --start 7
Dumping file SampleVolume, starting at block 7 for 13 blocks:

000E00: 2E 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E20: 00 1A 00 00 34 00 00 00  C2 AD 4D 64 00 00 00 00 | ....4... Md....
000E30: C2 AD 4D 64 00 00 00 00  00 00 00 00 01 00 00 00 |  Md............
000E40: 06 00 00 00 0D 00 00 00  00 00 00 00 00 00 00 00 | ................
000E50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E80: 2E 2E 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EA0: 00 1A 00 00 34 00 00 00  C2 AD 4D 64 00 00 00 00 | ....4... Md....
000EB0: C2 AD 4D 64 00 00 00 00  00 00 00 00 01 00 00 00 |  Md............
000EC0: 06 00 00 00 0D 00 00 00  00 00 00 00 00 00 00 00 | ................
000ED0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000F00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0010F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001100: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001110: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001120: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001130: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001140: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001150: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001160: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001170: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001180: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001190: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0011A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0011B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0011C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0011D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0011E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0011F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001200: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001210: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001220: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001230: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001270: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0012A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0012B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0012C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0012D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0012E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0012F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001310: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001320: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001330: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001340: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001350: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001360: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001370: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001380: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001390: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0013A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0013B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0013C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0013D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0013E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0013F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001400: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001410: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001420: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001430: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001440: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001450: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001460: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001470: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001480: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001490: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0014A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0014B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0014C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0014D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0014E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0014F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001500: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001510: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001520: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001530: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001540: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001550: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001560: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001570: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001580: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001590: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0015A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0015B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0015C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0015D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0015E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0015F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001600: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001610: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001620: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001640: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001660: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001670: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001680: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0016A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0016B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0016C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0016D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0016E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0016F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001700: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001720: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001730: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001740: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001750: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001760: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001770: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001780: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001790: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0017A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0017B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0017C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0017D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0017E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0017F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001800: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001810: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001820: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001830: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001840: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001850: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001860: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001870: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001880: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001890: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0018A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0018B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0018C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0018D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0018E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0018F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001900: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001910: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001920: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001930: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001940: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001950: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001960: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001970: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001980: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001990: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0019A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0019B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0019C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0019D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0019E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0019F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001A00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001A90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001AA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001AB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001AC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001AD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001AE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001AF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001B00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001B90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001BA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001BB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001BC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001BD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001BE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001BF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001C00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001C90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001CA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001CB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001CC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001CD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001CE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001CF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001D00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001D90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001DA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001DB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001DC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001DD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001DE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001DF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
001E00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001E90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001EA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001EB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001EC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001ED0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001EE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001EF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001F00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001F90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001FA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001FB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001FC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001FD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001FE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
001FF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
002000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0020A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0020B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0020C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0020D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0020E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0020F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

002100: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002110: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002120: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002130: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002140: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002150: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002160: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002170: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002180: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002190: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0021A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0021B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0021C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0021D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0021E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0021F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
002200: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002210: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002220: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002230: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002270: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0022A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0022B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0022C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0022D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0022E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0022F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

002300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002310: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002320: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002330: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002340: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002350: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002360: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002370: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002380: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002390: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0023A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0023B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0023C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0023D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0023E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0023F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
002400: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002410: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002420: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002430: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002440: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002450: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002460: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002470: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002480: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002490: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0024A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0024B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0024C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0024D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0024E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0024F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

002500: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002510: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002520: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002530: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002540: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002550: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002560: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002570: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002580: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002590: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0025A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0025B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0025C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0025D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0025E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0025F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

```
002600: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002610: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002620: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002640: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002660: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002670: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002680: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0026A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0026B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0026C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0026D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0026E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0026F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

002700: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002720: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002730: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002740: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002750: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002760: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002770: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002780: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
002790: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0027A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0027B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0027C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0027D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0027E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0027F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Analysis of the dump:

1. Addresses 000 E00 - 000 E1D, highlighted yellow, contain the value 0x2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00. This value represents the field “filename” in our directoryEntry struct and is the filename of the “.” entry, which is just obviously “.”. The two bytes after it, in addresses 000 E1E and 000 E1F are just padding since our filename is an array of 30 bytes.

2. Addresses 000 E20 - 000 E23, highlighted orange, contain the value 0x00 00 1A 00 (reversed from little endian). In decimal, this value is 6,656 which is the field

in our directoryEntry struct called “size” which is of type int and stores the size, in bytes, of a directory entry. This is the total number of bytes in use by the current directory which is why “.” has it in its respective directoryEntry.

3. Addresses 000 E24 - 000 E27, highlighted red, contain the value 0x00 00 00 34 (reversed from little endian). In decimal, this value is 52 which is the field in our directoryEntry struct called “numOfEntries” which is of type int and stores the number of entries in the directory entry. This is the total number of entries in the current directory which is why “.” has it in its respective directoryEntry.

4. Addresses 000 E28 - 000 E2F, highlighted green, contain the value 0x00 00 00 00 64 4D AD C2 (reversed from little endian). In decimal, this value is 1,682,812,354 which is the field in our directoryEntry struct called “dateCreated” which is of type time_t and stores the time, in seconds, since January 1, 1979 since the entry was created. This is the time the current directory was created which is why “.” has it in its respective directoryEntry.

5. Addresses 000 E30 - 000 E37, highlighted blue, contain the value 0x00 00 00 00 64 4D AD C2 (reversed from little endian). In decimal, this value is 1,682,812,354 which is the field in our directoryEntry struct called “dateModified” which is of type time_t and stores the time, in seconds, since January 1, 1979 since the entry was modified. This is the time the current directory was modified which is why “.” has it in its respective directoryEntry.

6. Addresses 000 E38 - 000 E3B, highlighted peach, contain the value 0x00 00 00 00 (reversed from little endian). In decimal, this value is 0 which is the field in our directoryEntry struct called “filetype” which is of type int and stores filetype, where 0 = directory and 1 = file. The “filetype” = 0, a directory which is why “.” has it in its respective directoryEntry, since “.” represents an entire directory.

7. Addresses 000 E3C - 000 E3F, highlighted off-white, contain the value 0x00 00 00 01 (reversed from little endian). In decimal, this value is 1 which is the field in our directoryEntry struct called “used” which is of type bool and stores the state of the entry, whether it’s used or unused, where true indicates used, and false indicates unused. Booleans are either 1s or 0s. Since this is the “.” entry and is used, the value is 1, true.

8. Addresses 000 E40 - 000 E43, highlighted olive green, contain the value 0x00 00 00 06 (reversed from little endian). In decimal, this value is 6 which is the field in our “ExtentNode” struct called “blockNum”, contained in our directoryEntry struct

called “extents” which is of type int and stores the starting block number of the root directory in the logical block array and free space map.

9. Addresses 000 E44 - 000 E47, highlighted pink, contain the value 0x00 00 00 0D (reversed from little endian). In decimal, this value is 13 which is the field in our “ExtentNode” struct called “count”, contained in our directoryEntry struct called “extents” which is of type int and stores the count, or, the number of blocks the root directory goes for in the logical block array and free space map

10. Addresses 000 E48 - 000 E7F, highlighted purple, contain values of 0x00 (reversed from little endian). In decimal, these values are 0 and represent the fields in our “ExtentNode” struct called “blockNum” and “count”, contained in our directoryEntry struct called “extents”. They are of type int and are all 0s since only the first extent of the “.” entry is filled.

11. Addresses 000 E80 - 000 E9D, highlighted yellow, contain the value 0x2E 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00. This value represents the field “filename” in our directoryEntry struct and is the filename of the “..” entry, which is just obviously “..”. The two bytes after it, in addresses 000 E9E and 000 E9F are just padding since our filename is an array of 30 bytes.

12. Addresses 000 EA0 - 000 EA3, highlighted orange, contain the value 0x00 00 1A 00 (reversed from little endian). In decimal, this value is 6,656 which is the field in our directoryEntry struct called “size” which is of type int and stores the size, in bytes, of a directory entry. This is the total number of bytes copied from the “.” entry, since this is the root directory and the root does not have a parent.

13. Addresses 000 EA4 - 000 EA7, highlighted red, contain the value 0x00 00 00 34 (reversed from little endian). In decimal, this value is 52 which is the field in our directoryEntry struct called “numOfEntries” which is of type int and stores the number of entries in the directory entry. This is the total number of entries copied from the “.” entry, since this is the root directory and the root does not have a parent.

14. Addresses 000 EA8 - 000 EAF, highlighted green, contain the value 0x00 00 00 00 64 4D AD C2 (reversed from little endian). In decimal, this value is 1,682,812,354 which is the field in our directoryEntry struct called “dateCreated” which is of type time_t and stores the time, in seconds, since January 1, 1979 since the entry was created. This is the time the current directory was created,

which is copied from the “.” entry, since this is the root directory and the root does not have a parent.

15. Addresses 000 EB0 - 000 EB7, highlighted blue, contain the value 0x00 00 00 00 64 4D AD C2 (reversed from little endian). In decimal, this value is 1,682,812,354 which is the field in our directoryEntry struct called “dateModified” which is of type time_t and stores the time, in seconds, since January 1, 1979 since the entry was modified. This is the time the current directory was modified, which is copied from the “.” entry, since this is the root directory and the root does not have a parent.

16. Addresses 000 EB8 - 000 EBB, highlighted peach, contain the value 0x00 00 00 00 (reversed from little endian). In decimal, this value is 0 which is the field in our directoryEntry struct called “filetype” which is of type int and stores filetype, where 0 = directory and 1 = file. The “filetype” = 0, a directory, which is copied from the “.” entry, since this is the root *directory* and the root does not have a parent.

17. Addresses 000 EBC - 000 EBF, highlighted off-white, contain the value 0x00 00 00 01 (reversed from little endian). In decimal, this value is 1 which is the field in our directoryEntry struct called “used” which is of type bool and stores the state of the entry, whether it’s used or unused, where true indicates used, and false indicates unused. Booleans are either 1s or 0s. Since this is the “..” entry and is used, the value is 1, true.

18. Addresses 000 EC0 - 000 EC3, highlighted olive green, contain the value 0x00 00 00 06 (reversed from little endian). In decimal, this value is 6 which is the field in our “ExtentNode” struct called “blockNum”, contained in our directoryEntry struct called “extents” which is of type int and stores the starting block number of the root directory in the logical block array and free space map (copied from the “.” entry, since this is the root directory and the root does not have a parent).

19. Addresses 000 EC4 - 000 EC7, highlighted pink, contain the value 0x00 00 00 0D (reversed from little endian). In decimal, this value is 13 which is the field in our “ExtentNode” struct called “count”, contained in our directoryEntry struct called “extents” which is of type int and stores the count, or, the number of blocks the root directory goes for in the logical block array and free space map (copied from the “.” entry, since this is the root directory and the root does not have a parent).

20. Addresses 000 EC8 - 000 EFF, highlighted purple, contain values of 0x00 (reversed from little endian). In decimal, these values are 0 and represent the fields in our “ExtentNode” struct called “blockNum” and “count”, contained in our directoryEntry struct called “extents”. They are of type int and are all 0s since only the first extent of the “.” entry is filled. (copied from the “.” entry, since this is the root directory and the root does not have a parent).

21. The rest of the values of the dump, since the size of directoryEntry is 128, making it a multiple of 512, are all values of 0x00, representing all of the other entries that come after “.” and “..” whose fields have all been initialized to values of 0.

<u>Description of Parsepath:</u>

For your convenience Professor, here is the struct of return values returned by parsePath() after its execution. We will reference it towards the end of this description of parse path:

```
// Return values to store the details from the search
struct directoryDetails
{
   // To store the index of the directory entry in the directory
   int indexOfDirectoryEntry;

   // To store the pointer to the current directory to keep it loaded
   struct directoryEntry *ptrToDirectory;

   /* To check if the final element of the path is either a file or a directory.
      A value of 0 = true, it is a directory, and a value of 1 = false,
      indicating that it's a file */
```

```
    int isDirectory;

    // The name of the final element of the pathname
    char *finalElementName;


} directoryDetails;
```

Parse path is a function that allows us to parse/tokenize each of the directory names in a path, traversing each one, and loading each one into memory, again and again, to allow us to perform certain directory operations on the final element of the path. Essa devised a plan to execute parse path. He reviewed the lecture video where you, Professor, discussed and taught us about parse path and wrote a header file called “parsepath.h” where he came up with a struct of return values that we we can pass into and return from parsePath(), in addition to four helper functions that would allow us to parse the path. These functions are: tokenizationOfPath (implemented by Jinwei in parsepathtokenization.c), locateEntry (implemented by Mahdi in parsepathtraversal.c), loadDirectory (implemented by Amit in parsepathload.c), and, the big one, parsePath (implemented by Essa in parsepathintegration.c).

The function tokenizationOfPath accepts a path name and an array of strings as parameters and using strtok_r, tokenizes the entire path name to allow the next phase to occur. The next phase is the use of locateEntry, which accepts as parameters the pointer to the loaded directory we want to traverse (the parent/current working directory) and the name of the current token whose name we want to check against the entry names of the current/parent directory. If no such name exists or if it does exist but the entry is not a directory, return an error. Otherwise, return the index of the entry of the current directory we’re traversing. Phase 3 is loading via the loadDirectory function. This function accepts as parameters the pointer to the loaded current/parent directory and the index into this current/parent directory and using these two very important pieces of information, loads that index using LBAread(), passing into LBAread that entry’s extents table. All three phases, all three helper functions, are called in the parsePath function, where Essa integrated all of these parts together. In this function, a pointer called “parentDir” is declared, representing the parent/current working directory that we want to parse from, and an array of strings, “arrayOfTokens” as Essa called it, is declared. This array of strings is passed into tokenizationOfPath where the array of strings is filled with C string values. Then, in a ‘for’ loop, we continually grab the next name/string of the path and pass that name/token into locateEntry, where we get the index of that

located directory in the parent/current working directory. And then, we load that located directory, resetting parentDir to be the newest loaded parent/current working directory. This ‘for’ loop runs for n - 1 number of times, where n is the number of strings contained in the array of strings “arrayOfTokens”. The final element of the tokenized path and the new parent/current working directory, “parentDir”, are assigned to the fields of the struct “directoryDetails” (the struct of return values we want to return from parsePath), “finalElementName” and “ptrToDirectory”. Parse path complete.

Description of Each Directory Function:

We will provide a description of each directory function in the order of how they appear in mfs.h:

fs_mkdir - fs_mkdir takes in as parameters a “pathname” and a mode in order to make a new directory named after the final element of that pathname. Mode is not a concern of Essa who implemented this function, just the pathname.

fs_rmdir - fs_rmdir takes in as a parameter a “pathname” in order to delete a directory whose name is the final element of that pathname.

fs_opendir - fs_opendir takes in as a parameter a “pathname” in order to open a directory for reading, that is, to allow the reading of its contents.

fs_readdir - fs_readdir takes in as a parameter “dirp”, an instance of the struct fdDir, in order to allow one to process a directory entry.

fs_closedir - fs_closedir takes in as a parameter “dirp”, an instance of the struct fdDir, to free that pointer returned by fs_opendir in order to close the directory since the task of reading it has finished.

fs_getcwd - fs_getcwd takes in as parameters a “pathname” as well as a “size” in order to allow one to obtain/print the current working directory.

fs_setcwd - fs_setcwd takes in as a parameter a “pathname” in order to allow one change their current working directory to a new one.

fs_isFile - fs_isFile takes in as a parameter a “filename” in order to complete a check to see whether or not the directory entry in question is a file.

fs_isDir - fs_isDir takes in as a parameter a “pathname” in order to complete a check to see whether or not the directory entry in question is a directory.

fs_delete - fs_delete takes in as a parameter a “filename” in order to delete a file whose name is the final element of that filename.

fs_stat - fs_stat takes in as parameters a “path” and “buf”, an instance of the struct fs_stat, in order to retrieve data about a file.

fs_move - fs_move is a function created by Essa to implement “mv” that takes in as parameters a “pathname1” and a second pathname, “pathname2”, in order to move a file/directory from one directory to another (and renames a file/directory as well).

Description of Each b_io Function:

We will provide a description of each b_io function in the order of how they appear in b_io.h:

b_open - b_open takes in as parameters a “filename” as well as “flags” in order to allow one to do things such as open a file for reading, or even creating a new file.

b_read - b_read takes in as parameters a file descriptor “fd”, a “buffer”, and a “count” in order to allow one to read data from disk into the caller’s buffer.

b_write - b_write takes in as parameters a file descriptor “fd”, a “buffer”, and a “count” in order to allow one to write data from a caller’s buffer to disk.

b_seek - b_seek takes in as parameters a file descriptor “fd”, an “offset”, and a “whence” directive in order to allow one to change the offset in an open file.

b_close - b_close takes in as a parameter a file descriptor “fd” in in order to close the open file.

Description of the fsshell Driver:

The driver of our file system project is fsshell.c. This driver is essentially a shell program whose purpose is to allow us to test our directory and b_io functions via command line arguments. There are multiple commands to utilize in order to test those functions: ls, cp, mv, md, rm, touch, cat, cp2l, cp2fs, cd, and pwd.

The command “ls” allows us to list the files in a directory. The directory functions fs_stat, fs_opendir, fs_readdir, fs_closedir, fs_isDir, fs_isFile, and fs_getcwd are all utilized for this purpose of listing a file(s). The command “cp” utilizes b_open, b_read, b_write, and b_close in order to copy a file from a source (via reading) into a destination (via writing). Command “mv” utilizes a function Essa wrote and implemented called “fs_move” in fs_move.c, which deletes a directory entry from the source directory and stores directory entry metadata about the source into the destination directory. The “md” command utilizes directory function fs_mkdir in order to create a new directory. It’s opposite, command “rm”, utilizes fs_isDir, fs_rmdir, fs_isFile, and fs_delete in order to delete/remove either a directory (if isDir is true) or a file (if isFile is true). Command “touch” utilizes b_open and b_close, and if the flag for b_open is either O_CREAT or O_WRONLY, a new file will be created. The command “cat” utilizes b_open, b_read, and b_close in order to display/print files to the console. Command “cp2l” utilizes b_open, b_read, and b_close in order to allow us to copy a file from our file system into the Linux file system (via b_read). Its opposite command, a “vice versa” command is “cp2fs”, utilizes b_open, b_write, and b_close in order to copy a file from the Linux file system into our own file system (via b_write). The final two commands are “cd” and “pwd”, where “cd” utilizes fs_setcwd in order to change the current working directory, and “pwd” utilizes fs_setcwd in order get or print out the current working directory.

So as can be seen, fsshell is a TOOL. It is a program that allows us to make sure that our file system is running just fine. If we didn't have this valuable tool to utilize, it would be quite difficult to know whether or not our functions are working properly.

Contributions:

Name	Contributions
Mahdi Hassanpour	- Created the writeup file, and shared it with the group. Wrote the meeting summaries on the write-up. - For Milestone 1 I worked on steps 4e,f, Loop through the directory entries and initialize each to a known free state - for Milestone 2 I worked on 3 root directory functions: - fs_isFile - fs_isDir, and fs_delete. function fs_isFile takes a filename as a parameter and returns an integer value indicating whether the file exists and is not a directory (i.e., a regular file). The function calls the fs_isDir function to determine if the given pathname refers to a directory, and returns 0 if it does, 1 if it is a file, and -1 if it's neither one (indicating an invalid path). fs_isDir takes a pathname string as input and returns an integer value of 1 if the given path refers to a directory, 0 if it is not a directory, and -1 if it is an invalid path. fs_delete takes in a filename as an argument and deletes the corresponding file from the file system by freeing up the associated blocks and updating the directory entry. - For Milestone 3 i worked on the b_seek b_io function. b_seek function allows the user to seek to a specific position in a file, based on a given file descriptor, offset, and whence parameter. The function initializes the system if necessary, checks that the file descriptor is within range, and modifies the file position based on the whence parameter. Finally, the function returns the final offset value.
Essa Husary	- Worked on File System Design 1 with the group.

	- For Milestone 1, I worked on the entire free space system myself. I created the map, initialized it, and allocated it, and did the bit manipulation functions. I also worked with Amit on the initialization/formatting of the volume. - For Milestone 2, I revamped the free space system by including the extent allocation function (called “blockAllocator”) and the function to free blocks via extents (called “freeBlocks”), as well as various other helper functions to accomplish these two tasks. I also revised and generalized the directory system so that ANY directory can be made regardless of whether it’s the root directory or any other directory as per the suggestions given to us by you Professor, in class. It was a great system, done excellently by my team, but it just needed this tweak along with some additional clean up. In addition, I planned out parsepath(). I created the header file for parsepath.h, thinking about what info we might need to store after calling parsepath() and wrote a struct called “directoryDetails” which stores the returnValues that we’ll need after calling parsepath(). I wrote the helper function prototypes that we’ll need for parsepath() called locateEntry, loadDirectory, tokenizationOfPath, and parsepath, and directed the group on how to do them and assigned who’s doing what. For parsepath, I implemented parsepath(), integrating everyone’s code, whereas my other groupmates implemented the other three helper functions, all of whom quite frankly, did an amazing job on. Finally, I wrote my directory functions, fs_mkdir, fs_rmdir, and fs_stat. - For Milestone 3, I didn’t work on the b_io functions, as I was concerned more with improving the free space system I wrote, improving the directory system my three other groupmates wrote, and making sure mkdir and rmdir work properly. However, I did write a large, large bulk of the writeup, including describing how we worked together and how we split up the work, and descriptions of every component of our file system for the writeup such as the general description and all of the descriptions following that (including hexdumps and the analyses of the hexdumps), while the team continued to refine and debug their code. Jinwei graciously volunteered to take on the b_write function after finishing b_open and b_close, since I was behind due to improving our free space and directory systems which

	was a necessary step to have a better file system, so I figured I would do the bulk of the writeup so the group can continue testing their code. I also implemented the move command in fsshell. After testing was complete, I, Jinwei, Amit, and Mahdi provided screenshots showcasing the commands utilizing all of our functions to complete the writeup.
Amit Bal	Worked on File System Design 1 with the group -For Milestone 1, worked on steps, 4a,b,c,d. Created functions to make necessary calculations. For example, writing a function to calculate the total number of blocks needed for the root directory. Worked on initializing the root directory with Mahdi. Initialized the fields needed for the root directory. As well as created the root directory array. The initialization method is called intiliazeRootDir(). Worked on writing and reading the root directory from disk using LBARead() and LBAWrite(). -For Milestone 2, we needed to first finish implementing the parsePath function. For parsePath, I worked on loadDirectory, which loads a directory, given the dirEntry array. I also worked on opendir(), readdir(), and closedir(). These directory functions allowed us to implement a fully working ls command. During meetings, I provided explanations on the implementation of my code. I did my best to try to describe my approach to the implementation. -For Milestone 3, I worked on b_read, I wanted to make sure that b_read worked with b_seek and b_write. I met with both Mahidi and Jinwei to make sure b_read, b_write, and b_seek all worked well together. I also helped resolve some issues in b_open(). For b_read specifically, I wanted to make sure that it could read from any part of a given file. I also needed to make sure that the cp2l command was working properly in fs_shell.
Jinwei Tang	Worked on File System Design 1 with the group For milestone 1, I work on step 4 g,h, implementing a method to set up the first and second directory entries as current directory entries and parent directory entries

	in every directory. For the root directory, its parent directory is NULL. For milestone 2, firstly we need to implement our parsePath function. During the development of the parsePath, I work on implementing a routine tokenizing the pathname and storing the tokens in a char[][]. For the actual fs_ functions, I work on fs_setcwd and fs_getcwd. During the milestone 2 meetings, I provide explanations on the fs functions base on my understanding. Also, I actively participate in evolving the structure of our fs system. For milestone 3, I work on b_open, b_write, and b_close. Also, I provide some advice on b_seek for Mahdi. Since the b_open directly relates to the implementation of b_read, I add more fields in b_fcb and inform the group to utilize the field for their implementation. During the implementation, I also actively participating in debugging and spending more time on addressing problems with teammates.

How our team worked together:

Our team tried to meet as often as we possibly could. We met on Discord on the days that all four of us could meet. We had many meetings of around 2-3 people at times in between group meetings in-person and via Discord when we wanted to express concerns, share ideas, or check in on one another as all four of us meeting frequently wasn't always plausible. However, on the days that all four of us could meet, we did multiple things such as talking about the next steps in the milestone at hand, dividing up the work for the milestone at hand, and even spending time debugging each other's code. We divided the work such that we all had a balanced workload given the amount of understanding each person had with the different parts of the file system and also given the amount of homework people had from other classes. The point was to make things fair for everyone. So we split up the work in the following way for each milestone:

Milestone 1:

Essa and Amit worked on step 1, sharing ideas with each other, understanding how to check whether or not the volume has been formatted, and figuring out how to initialize the system. Essa worked individually on and concerned himself with the free space system, doing things such as creating/initializing it (the initialization function is called freeSpaceInit()), and writing the allocation and bit manipulation functions (the allocation function is called getBlocks()). Amit worked on steps 4.a - 4.d, performing the necessary calculations to determine how many blocks and entries we need. Mahdi worked on steps 4.a - 4.d, looping through each directory entry structure to be in a known free state. Jinwei worked on steps 4.g and 4.h. While he began his implementation, he gained information about what the directory entry array will look like and also wrote the code that would serve as a foundation for Mahdi to continue implementing his steps. While implementing, he gained updated information on the implementation of the volume control block and free space map through the meetings. He utilized that information to update the way he codes. After he finished his method, he pushed his code to his own branch so others can view them and use the method to coordinate with their sections.

Milestone 2:

The group met to discuss milestone 2 and who would work on what. Essa provided his plan for parse path and assigned each person a helper function to implement. Essa assigned Jinwei the helper function tokenizationOfPath, Mahdi the

helper function locatEntry, and Amit loadDirectory, and assigned himself the integration function, parsePath. When the group had finished working on their assigned helper functions for parse path, the group met again to now split up the directory functions amongst each other. Essa assigned himself fs_mkdir, fs_rmdir, and fs_stat, assigned Mahdi fs_isFile, fs_isDir, and fs_delete, and assigned Jinwei fs_getcwd and fs_setcwd. Amit volunteered to do fs_opendir, fs_readdir, and fs_closedir. The plan to do our directory functions by milestone 2 unfortunately didn't work out as planned because things from other classes and our personal lives affected our progress on our functions. However, over the next couple of weeks, after multiple office hours sessions and reviewing lecture recordings, the group had eventually finished their directory functions.

Milestone 3:

At around the time when the directory functions of milestone 2 should have been completed, the group reconvened to now check up on each other's progress for milestone 2, to discuss milestone 3, and to split up the b_io functions amongst each other. Essa assigned himself b_write, Mahdi b_seek, Amit b_read, and Jinwei b_open. Jinwei also volunteered to do the corresponding b_close. Just like milestone 2, things didn't go according to plan and unfortunately, we were late in completing those functions. Essa was behind on his milestone 2 functions because of the fact that he had to revamp the free space system a little in order to create the functions that allocate blocks for extents and to free blocks associated with a directory entry's extents. When he created the free space system, he didn't have time to write those two functions. He also revised the directory system a bit to make it comply with what you, Professor, showed us in class in your review of milestone 1. Since he had to do these tasks, he was behind on his milestone 2 directory functions. However, Jinwei, the rockstar that he is, graciously offered to do b_write for him so we can finish in time. In this time, Mahdi finished his directory functions and his b_seek function and Amit finished his directory functions. Amit also was finishing up his b_read function as well. Essa finished his directory functions and then, since b_write had been written for him by Jinwei, he decided that the best thing to do for the group was to do a large bulk of the writeup so Mahdi, Amit, and Jinwei had lots and lots of time for testing and didn't need to worry about the writeup. And so, Essa went on to write all of the descriptions for every single component of the file system (all found before the "Contributions" section), the hexdumps and their associated analyses, and this section. The "Contributions" table, the "Approach/What I Did" section, the "Issues and Resolutions" section, and the screenshots were what the rest of the group including Essa contributed. Essa also

decided to implement the “mv” command in fsshell.c after he finished the bulk of the writeup on behalf of the group since he’d much rather have them work on testing and debugging their functions and not have to worry about that command. Amit finished up his b_read function in fantastic fashion, and the group went on to finish up the writeup and project excellently, and finally submitted the assignment.

Overall Summary:

Needless to say, this group, though we were behind on our tasks, did not hesitate to help each other one bit. We all worked with each other very well and tried our best to meet as much as possible to discuss this project. We sacrificed a lot of time from homework for other classes as well as sacrificed a lot of free time in order to be there for one another when we each struggled. No one wasted any time. No one procrastinated. We all were very serious about this project and were very professional in our dealings with one another.

********Mahdi has graciously provided the group with meeting summaries, where during each meeting, he wrote notes as to what is being discussed and what the meeting is about. Here are the logs/notes for the meetings where all four of us attended:

Meeting 1 (discord): 03/28
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Worked on File System Design 1
- Accepted the assignment on GitHub and sent invitations

Meeting 2 (discord): 04/03
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Worked on File System Milestone One
- Divided the tasks for File System Milestone One

Meeting 3 (discord): 04/05

Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Going over what we have done so far for milestone 1
- Helping each other with our problems

Meeting 4 (discord): 04/07
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Started planning on how to do the milestone 2
- Dividend the parse path function helper to fully implement the parsed path

Meeting 5 (discord): 04/09
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Organized the code from milestone 1 and pushed it to the main branch
- Divided the functions for milestone 2

Meeting 6 (discord): 04/16
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Going over what each person did for milestone 2 so far
- Start Planning how to do milestone 3

Meeting 7 (discord): 04/20
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Worked on directory entries function

Meeting 8 (discord): 04/23
Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Catching up on hows everyone doing on their part for milestone 3

Meeting 9 (discord): 04/25

Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Just like the previous meeting, we were catching up on how everyone is doing on their part for milestone 3.
- Helping each other debug code

Meeting 10 (discord): 04/29

Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Just like the previous meeting, we were catching up on how everyone is doing on their part for milestone 3.

Meeting 11 (discord): 04/30

Attended: Mahdi Hassanpour, Essa Husary, Amit Bal, Jinwei Tang

- Finishing the project
- Helping each other debug code
- LOTS and LOTS of testing
- As a group, we provided screenshots of the commands to showcase our file system

How we approached completing our parts in the assignment:

Essa:

Milestone 1:

I, Essa, worked on the free space system and worked with Amit on the initFileSystem() function of the fsInit.c file where we formatted the volume and wrote the checks to see whether or not we have already formatted the volume (for subsequent boots of the system). I approached my parts of the assignment in the following way, starting with the formatting of the volume. So my first thought when looking at step 1 of the "Steps for milestone 1" pdf document was, "Okay, so we have to basically allocate some space to store information about our VCB. I can do that. But as for checking to see if the magic numbers match, how can I do that?" So I began thinking about maybe storing the data in a buffer and reading it in somehow. That turned out to be the wrong approach as I learned in a one-to-one office hours session with you, Professor. I learned that all I had to do was just do an LBAread() starting at block 0 into the space I allocated for the VCB using malloc(). It was that simple. And then I had to check if the field in our VCB for the magic number matches with what was read from disk. Amit thought about it on his own separately, but funny enough, we had the same exact implementation and ideas. Now onto how I approached my individual part, which was the free space map.

As I was reading step 3, I thought about how to initialize the free space map. I first, after reading step 3.a, came up with my own equation to calculate the number of blocks we'll need. But after an office hours session, I realized that the correct equation to use was the equation covered in class, which was [(m + (n - 1)) / n]. I then read step 3.b and did what it said to do which was to malloc the space needed for the free space map. Next, I read 3.c and thought to myself, "Okay, so I remember from one of the in-class lectures that the Professor actually went over how we can manipulate individual bits. Why don't I look at that." And I did. But it was more complex than it seemed and I ran into some issues implementing it (see "Issues and Resolutions" below). But I eventually figured it out. I wrote three functions after I figured out how to do so: setBit(), clearBit(), and initializeMap(). The function setBit() sets the value of a bit to a value of 1, which indicates that a block is free. The function clearBit() sets a bit's value to a value of 0, indicating used. And, initializeMap(), creates the entire map using setBit(), setting every bit in the allocated space to a value of 1 which appears in the hexdump as "FF". So I figured out how to manipulate bits, but I, unfortunately, ran into another issue. I had a hard time writing the allocation procedure, figuring out how to get the caller their desired number of blocks, returning the starting block number, and marking them as used in an efficient way (see "Issues and Resolutions" below). Fortunately, I figured out how to do so and once I did, I proceeded to step 3.d, writing the free space map to disk. And finally, I then proceeded to the final part of that step, step 3.e, and recorded the value of the starting block of the free space map in the volume control block. My part has been completed.

Milestone 2:

I, Essa, for Milestone 2, first of all, revamped the free space system by writing the extent allocation function (called “blockAllocator”) and the function to free blocks via extents (called “freeBlocks”), as well as various other helper functions to accomplish these two tasks. So the allocation function we had for Milestone 1 was good, but it doesn’t consider extents. All it did was check for a desired number of *contiguous* blocks. I simply didn’t have enough time to write an allocation function that fills a directory entry’s extents table with starting block numbers and counts. I didn’t even have enough time to write the freeBlocks function. But I did in this Milestone. So I thought about how to write the blockAllocator function. I thought, “Hmm, we probably as best as possible want to avoid filling in more than one extent if we can just find a desired number of contiguous blocks. This way, only one extent is filled with a starting block number and a count if we can find many contiguous blocks. We’d only need to fill in extents once things get hairy and there starts to be blocks scattered here and there. This would be most efficient for us to do.” And so I did. I included a check at the beginning of this function to check if we can just get a contiguous number of blocks first and foremost. Then I thought, “Okay, now things get tricky. How can I now figure out how many blocks are left to give to the caller and how can I fill in an entry’s extents table?” This took a lot of thinking……..A LOT of thinking (see “Issues and Resolutions” below). But once I figured out how to do so, I began filling in the extents table of the directory entry, one by one, until all of the blocks the user needs are allocated. We now have a function that allocates blocks via extents.

After this step, the next step was to now finally write a function that frees the blocks allocated to a directory entry. This was honestly very easy. I thought, “Okay, so what I need is a starting position to pass into setBit (which sets a bit to a value of 1 by passing in a block position in the bitmap) and this value should be the starting block number of the first extent. Then, I can just have a nested ‘for’ loop which loops for “count” number of times, setting “count” number of bits to values of 1 by calling setBit.” And I did just that for every extent in the directory entry’s table of extents. We now have a function that deallocates blocks via extents.

Next up was the revision of the directory system. Now, when I say revision, I am in no way implying that what we had was BAD. In fact, it was super good. My three other teammates did an excellent job on it. The only glaring issue was that it was specific only to creating the root directory and there was some clean up that had to be done to reflect how you, Professor, wanted it to look like. And so, I got to it. I first thought, “Okay, so I remember Professor discussing and going over Milestone 1 in

class. He talked about the directory system and what we can do to make sure it can create any kind of directory." And so I reviewed the lecture video and knew what I had to do. It was simple. Just include a new parameter called "parentDirectory" and if this parameter is NULL, I initialize values such that the ".." entry of the root directory contains the same information as the "." entry of the root directory. If it's not NULL, I initialize ".." to store the information about the parent directory. Our directory system looks very good now and we can use mkdir to its full potential.

After the revision of the directory system, my group and I now had to think about parse path. We decided to meet to discuss it and how to go about doing it, and before this meeting, I decided to go ahead and come up with a plan to present to the group. Amit had suggested that we can work on the different parts of it separately and we decided that that was a good idea. And so, I devised a plan for that to work. I began my thought process as such, "According to the lecture video that I just reviewed, we can have three helper functions and one integration function. One of these helper functions can tokenize the pathname (which I called "tokenizationOfPath" in parsepathtokenization.c), another one can traverse the entire parent directory to locate one of its children (which I called "locateEntry" in parsepathtraversal.c), and another one to load the located child directory (which I called "loadDirectory" in parsepathload.c). And then," I continued, "We can have an integration function which will piece everything together so that we may fully parse the path (which I called "parsepath" in parsepathintegration.c)." I then wrote these prototypes in parsepath.h and then starting thinking about the return values you, Professor, mentioned in the lecture video. I thought, "Okay, what I can do is create a struct that'll store information such as a pointer to the loaded parent directory in which we may want to create a directory, remove one, create a file, remove a file, etc., and we can also store the final element of the pathname so that we can perform one of the aforementioned operations in that loaded parent directory." And that's what I did. I compiled all of my ideas into a header file where I wrote comments, really instructions, as to how my groupmates can do them and assigned each function to each person (I assigned Jinwei "tokenizationOfPath", Mahdi "locateEntry", Amit "loadDirectory", and myself "parsepath"). My group loved it. Of course, the function prototypes weren't perfect as two functions required an extra parameter, but 95% of the plan was fully executed according to how I imagined it. Each person amazingly implemented/wrote the function definitions and then I pieced everything together in parsepath(). It was a simple process really. We keep the currentDirectory loaded into memory (currentDirectory is always equal to rootDirectory when we first initialize the system) and I use a 'for' which iterates through the array of tokens, locates an entry with a matching name, loads it, moves on to the next token, etc., etc. Parsepath complete.

Next on the agenda was mkdir. Now honestly, mkdir wasn't too bad. It was just a little tricky. The first thing I had to think about was the case in which a directory entry with the same name already exists. If it does, my logic went, return -1 since we've failed to make a directory. The next thing I had to think about was the normal scenario in which there doesn't exist an entry with the same name and there are entries that are marked unused. In this case, we simply iterate across the parent directory, find the first unused entry, initialize a new directory via initializeDir(), allocating some blocks for it (initializeDir allocates blocks automatically), and mark it as used. Now comes the tricky part. I had to consider two very likely scenarios that may occur at anytime when making more directories. Scenario one is the scenario in which we can find blocks adjacent to the range of the parent directory's LBA extent positions. What I mean is for example, if an extent had a starting block number of 20 and a count of 3, that range would obviously be 20-22. This scenario considers the case in which, due to a series of allocating blocks and freeing blocks, we can somehow find 3 blocks starting at block number 23, reallocate the size of the parent, and modify the children directories' ".." entries to reflect the changes in the parent directory. And I say 3 blocks not because it's a hard-coded number, but because I made a #define called "EXTRA_BLOCKS_FOR_NEW_DIR" which has a value of 3. So that's the first scenario. The second scenario considers the case in which the scenario I just described doesn't apply, where we can't just modify a count by adding "EXTRA_BLOCKS_FOR_NEW_DIR" number of blocks to the count. This is where we'll unfortunately need to resort to creating a new extent entry where we will need a new starting block number. Both of these were tricky to deal with in terms of how to modify the parent directory (see "Issues and Resolutions" below) but I eventually figured it out. Once I did, mkdir was now complete.

After mkdir was completed, I moved on to rmdir. Rmdir was rather straight forward and simple to implement. All I needed to do was call parsepath and have a parent directory loaded. Once it's loaded, I can find the target child directory we want deleted, load it, iterate across it to ensure that every entry in the child is empty, free the child directory, and then in the parent directory, deallocate the blocks associated with the child directory. Of course, I had to mark it unused and set its filename to the null terminator, "\0". Rmdir complete.

Finally, I moved on to fs_stat, the easiest function of all. All I did was first, call parse path. Once parse path was called and the parent directory was loaded, I found the specified file by its name. Then once I did that, all that was left to do was to fill the fs_stat struct with values/information contained within that directory entry. Fs_stat complete.

Milestone 3:

I, Essa, for Milestone 3, wrote the bulk of the writeup for the group while they continued testing and debugging their code. As I said previously, I was behind on my directory functions because I had to revamp the free space and revise the directory system a bit, so Jinwei volunteered to do my b_write function so that we can get those systems improved. As a result, I had no b_io function to do. I did however have two things on the agenda to do, instead of doing nothing while Jinwei finishes up b_write. I can one, do the writeup for the group, and two, implement the “mv” command, both of which I did. I of course won’t be discussing how I approached the writeup, but I will now talk about how I approached implementing the “mv” command.

So I first used the man pages to figure out what “mv” does. I figured out that it copies a file or directory from one directory to another. It acts similarly to “cp”, I realized, so I figured that the code in fsshell for “mv” would be similar to the code for “cp”. I then thought, “Okay, well, once I do a move, I’m going to have to modify the directory that the entry was moved from in addition to modifying the directory that the entry was moved to. The metadata will be transferred. But how do I do that?” I figured that I’d need to do something similar to mkdir where I copy a directory entry’s fields into another directory entry via strcpy and a ‘for’ loop. But there was an issue doing so as data was being flipped around since parse path was being called twice. This issue took me some time to figure out (see “Issues and Resolutions” below). But once, I did, once I managed to figure out how to successfully move files and directories from one directory to another, I had finished the “mv” command.

Mahdi:

Milestone 1:

I, Mahdi. Worked on steps 4e,f , Loop through the directory entries and initialize each to a known free state, then request 6 blocks from the free space system and get the starting block number. And set the 1st entry to “.’ and the 2nd entry to “..”. The code begins by looping through the array of directory entries and setting each entry's file name, size, date created and modified to 0 or an empty string. This is done to ensure that all entries in the array are properly initialized before being used. Next, the code sets the first two directory entries to "." and "..", respectively. These are typically used to represent the current directory and its parent directory, respectively. The code uses dynamic memory allocation to allocate space for the file names of these two entries since their lengths are different from the other entries. It then copies the file names into the allocated memory and sets the size and date created and modified to 0.

Milestone 2:

For Milestone 2 i worked on fs_isFile, fs_isDir, and fs_delete directory functions.
Fs_delete: deleting a file is marking it unused and asking for the free space to free the blocks. So the first thing was to allocate memory for directoryDetails structure to store the return values of parsePath() function. And after that call parsePath() function to parse the given filename and get the details of the parent directory and the file to be deleted. if the file is not found, we will free the memory allocated for directoryDetails and return an error. And then If the file is found, free the blocks associated with the file by calling the freeBlocks() function with the extents array of the directory entry, reset the file name to the null terminator, and mark the directory entry as unused.

fs_isFile:
The fs_isFile function takes a char pointer filename as input and returns an int value. * The function checks whether the specified file is a regular file and not a directory or symbolic link. My approach to writing this function was to first check if the given pathname refers to a directory by calling the fs_isDir function. If it does, I return 0 because it is not a file. If fs_isDir returns 0, then it's a file, so I return 1. If fs_isDir returns anything other than 0 or 1, it's neither one, in fact, it may be an invalid path, so I return -1.

fs_isDir:
The fs_isDir function takes a char pointer pathname as input and returns an int value. * The function checks whether the specified path is a directory or not. * If the path is a directory, the function returns 1, otherwise, it returns 0.
First I needed to Allocate memory for a struct to store the return values of the parsePath() function. And then parse the pathname using the parsePath() function and store the return values in the previously allocated struct. checks the validity of the given pathname and returns -1 to indicate an error if the path is invalid. It then retrieves the

parent directory of the target directory entry and the index of the target directory entry. If the target directory entry does not exist, the function returns -1 to indicate an error. Next, the function checks if the target directory entry is a directory or a file and returns 1 to indicate a directory or 0 to indicate a file. Finally, it frees the memory allocated for the return values of parsePath().

Milestone 3:

I have implemented an interface to seek function for a file system. The b_seek function takes three parameters: fd (file descriptor), offset (the offset to be added to the file position), and whence (specifying how the offset should be applied). I started by checking if the system was initialized by checking the value of startup. If it was 0, I called b_init() function to initialize the system. Next, I checked if the file descriptor fd was within the valid range of 0 to MAXFCBS-1. If it was outside this range, I returned an error (-1). Then I created a variable new_offset to store the new file position. I then used a switch statement to modify the file position based on the whence parameter. If whence was SEEK_SET, I set the file position to the offset. If whence was SEEK_CUR, I moved the file position by the offset. If whence was SEEK_END, I set the file position to the end of the file plus the offset. If whence was not any of the above, I returned an error (-1).

Amit:

Milestone 1:

I, Amit, worked on making the necessary calculations in order to initialize the root directory, wrote the root directory to disk using LBAWrite(), and loaded the root directory from disk. In order to make the necessary calculations for the root directory I created four different functions. First, I knew I had to calculate the total number of bytes needed for my root directory. I created a function called numBytesForEntries() which multiplied the total number of entries needed by the size of each directory entry. Next, I needed to calculate how many blocks were needed for the root directory. I created a function called numBlocksToAskFreeSpace() which took the total number of bytes needed for the root directory and divided it by the size of each block. Finally, I needed to calculate how many extra entries could be added to the root directory without exceeding the number of blocks allocated for the root directory. I created a function called numExtraEntries(), which modded the total number of bytes for the root directory by the block size. This gave me the total number of extra bytes that could still contain directory entries. I then calculated the total number of extra entries that could be added by dividing the extra bytes by the size of the directory structure. I then created a function called initializeRootDir() which leveraged both Jinwei's and Mahid's code. This function would loop through the total number of directory entries setting each data member to it's

proper value. For example the name of the root directory's first entry should be set ".". After initializing the root directory it was time to write it to disk. In order to write the root directory to disk, I needed to use LBAWrite() which takes three parameters, a buffer to write from, a count of blocks to write to, and a starting block number. I got the starting block number by leveraging a method in the free space map called getBlocks() which takes the number of blocks needed for the root directors as a parameter, and returns the starting block number. I then leveraged my numBlocksToAskFreeSpace() function, to get the count of blocks to write to. Finally, I malloced a buffer the size of the root directory. To load the root directory, I had to leverage LBARead() which takes three parameters, a buffer to read from, a count of blocks to read, and a starting block number. I got the starting block number from the VCB structure. I got the count of blocks to read by leveraging my numBlocksToAskFreeSpace() method. Finally, I malloced a buffer the size of the root directory.

Milestone 2:

I, Amit, worked on fs_opendir(), fs_readdir(), fs_closedir(), and loadDirectory(). For loadDirectory(), I had to write a function that would load a directory, given the parent directory and the position of the directory within the parent. To start I malloced a buffer for the directory that will be eventually loaded. This buffer will be returned at the end of my function. I created a variable called offset which would make sure that we are loading the proper amount of bytes into the buffer. This will be explained further in issues and resolutions. I then created a loop to loop through the extent table of the directory. Since we allocate the same number of extents for every directory entry and every directory is not of the same size, I made sure to break out of the loop when the starting block number for the extent =0. This tells me that I have read all the extents for the given directory entry. Also within the body of the loop, I made sure I loaded each extent using LBARead() into my buffer. I incremented the offset by itself + count of the extents *sizeof(block). I finally returned the buffer to the caller.

fs_opendir(): For fs_opendir(), I had to write a function that would load a directory given the pathname. The function returns a fdDir pointer, which is our file descriptor. I started by calling parsepath which will load the directory based on the given pathname. Parsepath will return to me our directory entry structure which stores all the information about the given directory. I then malloced a buffer for the fdDir structure and copied the data from my directoryEntry structure into the fdDir structure. I then returned the pointer to the fdDir structure.

fs_readdir(): For fs_readdir(), I had to write a function that took the pointer to the fdDir structure returned by fs_opendir() as a parameter, and return the info of the first file or directory that is not set to null. This function will read a single directory, so the caller will

have to call fs_opendir() multiple times, to read multiple directories. Also this function returns a pointer to a fs_diriteminfo function. I began by altering our mfs.h file. Professor mentioned in class that we had to wrap our diriteminfo structure within the fdDir structure. This will allow me to access the diriteminfo structure through the fdDir structure. Since the current working directory is already loaded because of fs_opendir(), I had access to our own directoryEntry structure. I started my code by writing a loop, this loop will loop through all the directory entries, checking to see if any of their file names were set to null. If the filename was set to null I would increment the directoryEntry position. If the filename was not null I would exit the loop. This maintains that we don't read any null directory entries. I then checked to make sure that the directoryEntry position was not greater than the total number of entries in the current directory, more on this in issues and resolutions. I then copied the data from directoryEntry structure into the diriteminfo structure within the fdDir function provided by the caller. I then returned the diriteminfo structure.

fs_closedir(): For fs_closedir(), I had to write a function that took the fdDir pointer as a parameter and essentially freed it. This function was very simple, all I needed to do was to free the pointer to the fdDir structure returned by fs_opendir(). I called free on the given fdDir pointer and then I set the pointer to null.

Milestone 3: I, Amit, worked on b_read(), which takes three parameters, a file descriptor, a pointer to a char buffer, and the count of bytes to read. My approach to this function was very similar to my approach for assignment 5. In order to properly read a file, I need to be able to account for a fragmented file. A fragmented file will use multiple extents, this will force me to create a function that maintains the file position and the total number of bytes read. This is why there are three parts to b_read().

Part 1: The first part occurs if the user buffer is smaller than the file system buffer. This occurs when the user asks for a count of bytes less than a block (512 bytes). If this is the case, then copy the user requested number of bytes from the os buffer to the user buffer. Then return the total number of bytes the user requested.

Part 2: The second part occurs if the user is requesting more bytes than are in the file system buffer. This means that the file system buffer already has some bytes left in it (most likely from the last call to b_read()). But since I do not want to reread the bytes that are already contained in our file system buffer, I need to copy the bytes already contained in the file system buffer to our user buffer. I will do a memcpy of the total bytes in the filesystem buffer to the user buffer. If I have read the entire file then return total bytes read otherwise loop.

Part 3: The third part occurs if the user buffer is larger than the file system buffer. This will occur when the user asks for a count of bytes larger than the size of the first extent entry. I will need to loop through my extents table, making sure I'm reading in the total number of extents I need to satisfy the user's request. I will then have to load the extents by calling LBARead() into the filesystem buffer. I will then need to do a copy of the user requested number of bytes from the filesystem buffer to the user buffer. If I have reached the end of the file then return total bytes read otherwise loop.

Jinwei:

Milestone 1:

I, Jinwei, worked on setting the root directory's first entry to be "." and second entry to be "..". Since we are doing our part separately for the root directory, I need to figure out what information I can expect to gain from the precedent implementation. In class, we have discussed that "." represents the current directory and ".." represent the parent directory. So I need to figure out what information can be stored in their fields. First, no matter when we set up the directory for a root directory or normal directory, their original states are the same, their sizes are always stored in the first directory entry, the time is always the time when we create the instance directory, the starting block and length are always gained from the free space map. So I can set up a method that takes an int startingBlockNum, int size, and directory entry array and sets up the first directory entry "." dynamically. Since the ".." represents the parent directory, the setting of ".." should base on the parent directory. But for the root directory, since it's the top directory of our file system, its parent directory is NULL. Thus, the ".." of the root directory will just inherit from the ".". Since the setting of the root directory is once and forever, the volume control block should hold the information of the root directory. Thus, I add another parameter VCB to the parameter list. Also, since this method is just to set up the first two directory entry, the file type of both of them are set to the directory.

Milestone 2:

parsePath():

Before we can implement our fs functions, we first need to get the parsePath function to analyze the input path, search the directory in our file system and return the result. After reviewing the class recording, the group comes up with four steps to approach parsePath - tokenize the path, locate the index of the token in the directory, load the directory, and a function to integrate the above three functions. I work on tokenizing the path. I define a function named tokenizationOfPath. It will take a path and a holder for the tokens. Initializing the holder before filling in tokens to get rid of the tokens from the previous call. I set "/" and " "to trim the space and separate the tokens on the delimiter

slash since the slash is the directory separator. After tokenizing the path and filling the token into the holder. The tokenizationOfPath() will return the number of tokens for further retrieving tokens for the other implementation.

fs_setcwd():

For the fs functions, I work on fs_setcwd and fs_getcwd. The fs_setcwd will change the current working directory to the target directory if the path to the target directory is valid and the target directory exists. Firstly, I am not quite sure how is the current working directory to be recorded. Fortunately, Essa gives me the solution by having a global directory entry pointer to store the current working directory. Then, I have a way to keep track of the current working directory. I call parsePath to return the parse result on the path. Also, on the start-up of our system, the current working directory will initially set to the root directory. By calling parsePath and passing the current working directory as a starting point, the parsePath will analyze the path to see if the intermediate directory exists. If yes, then check if the last element is a directory or not. If the last element is also a directory, I will call our directory loader loadDirectory() and point the current working directory to the returned directory.

fs_getcwd():

After setting the current working directory, I also need to figure out some way to know the absolute path of the current working directory. Since the absolute path is changing over time when we call fs_setcwd() and we also need to consider not to overuse the resource, a good practice is to have a structure to store the information over time. Thus, I create a structure that has the field char pathTokens[][] and int numOfTokensInPath. The pathTokens[][] will be filled and modified by the fs_setcwd. The numOfTokensInPath will keep track of how many tokens are in the path to make sure it won't exceed our path depth of 30. During implementing fs_setcwd, I also need to consider the case when the directory entry name is "." and "..". Since "." represents the current directory, I don't need to change tokens in the pathTokens[][]. But ".." will move the current working directory to the parent directory, I need to remove the last token in the pathTokens[][] only if the pathTokens[][] is not empty.

The fs_getcwd() is pretty straightforward. It retrieves the pathTokens and formalizes it into an absolute path string and returns it to the caller's buffer. I set it to return a string at least starting from a single slash if the current working directory is the root directory.

Milestone 3:

For milestone 3, I work on b_open(),b_write(), and b_close().

Before I start planning on implementing, I check the man page first to see how exactly they should work.

b_open():

b_open() will take a filename in a path, and a flag as parameters. First of all, I called b_getFCB() to obtain a free slot in the file control table. If no slots are available, return -1 and exit the routine. Then, I will call parsePath to make sure the intermediate path is valid until the last element in the path. Also, I have some variables to hold the result of analyzing the last element. I have int fileExists and int fileNameIsUsed to store if the file exits in the directory and if the file name has been used. Once the intermediate path is proven to be valid, I check if the directory entry of the last element is a directory of a file. If it's file, set fileExists to TRUE. I check if the name has been used by some directory entry. If the name is used, set fileNameIsUsed to TRUE. These two variables are utilized to determine if the file can be opened or if the file can be created by calling the flag O_CREATE. Then I check if the flag O_CREATE is being called. If yes, only when fileNameIsUsed is FALSE can the file be created. While creating a new file, I obtain the parent directory and obtain a free directory entry slot, filling in the information of an empty file and tracking the pointer to the file entry in b_fcb. If the O_CREATE is not called, I check if fileExists is TRUE. If the file is not found, exit and return -1. If either the file is found in the directory, track the pointer to the file entry in b_fcb. Later, fill in all the related information in b_fcb. I also include int canRead and int canWrite in the struct b_fcb to keep track of the flags O_RDONLY, O_WRONLY, O_RDWR. If flag O_TRUNC is called, I will erase the content of the block allocated to the file on the disk and release the allocated block to the free space map. Then set the size of the file to zero. If flag O_APPEND is called, I will change the file offset to the size of the file. Before I exit the routine, I free the malloced unused resource. Finally, return the file descriptor.

b_write():

b_write() is a complicated routine as b_read(). It needs to consider many situations depending on the file's allocated blocks and the size of the caller's buffer. b_write is a routine similar to the Linux file routine write(). It will write the buffer to the file starting from the file offset. If the file's allocated block is not big enough to hold the buffer, we need to allocate more blocks to the file. First, I need to plan out the logic of how to write the buffer to the correct position in the file. Since the data are stored in a unit of the block (usually 512 bytes), in some cases, I need to read the block to my own buffer first and then write the caller's buffer into my own buffer and write my own buffer to disk. There are also situations, I need to allocate more blocks to the file to hold the exceeding bytes.

Then I start to implement the b_write by checking if the file is opened for write. If not, return -1. Then first, I check if the file has some allocated block. If true, go to the block where the file offset is currently in. There are two cases - either the file offset is within the file's allocated block or the offset is beyond the file's allocated block. If the offset is beyond the file's allocated block, we need to allocate more blocks to hold the rest of the caller's buffer. If the file offset is within the file's allocated block, read the block into my own buffer, write the caller's buffer from the "hasWritten" (a local variable to keep track of how many bytes have been written to the file, initially is zero) for the size of caller's buffer or the remaining bytes from the file offset in the current block (whichever is smaller). Update hasWritten, count, and file offset. Write my own buffer back to disk. If the caller's buffer does not fully write to the file, check if the file has more allocated blocks. If yes, check if the caller's remaining buffer is large enough to directly write to a block. If true, write the caller's remaining buffer directly to those available file blocks. Update hasWritten, count, and file offset. Check if there are still some available blocks of the file. If no more blocks are found, it means the caller's buffer will write beyond the file's allocated blocks. Then we need to allocate more blocks to hold the rest of the caller's buffer. If there are still blocks in the file, it means the rest of the caller's buffer can only write to partial block. Read the file block to my own buffer, write the remaining buffer to my own buffer, and write my own buffer back to disk.

For the situation when we need to allocate more blocks, I create a temporary extentNode array to hold the returned free blocks from blockAllocator() of our file system. Then I write the remaining buffer to the temporary blocks and merge the temporary extentNode array to the file's extentNode array. Also, the merging process will check if the starting block of the temporary extentNode is adjacent to the file's allocated block. The change to the file's extent will be written to disk by calling b_close().

Lastly, the b_write will return the hasWritten to indicate how many bytes are actually written to the disk.

b_close:

b_close() is a simple one. I write the parent directory back to disk because there may be some change on the directory entry, such as creating a new file, resizing the file while calling b_write, etc. Also, I release the unused resource while utilizing the b_fcb.

Issues and Resolutions:

Essa:

Milestone 1:

I, Essa, as I said before, worked on the free space map and worked with Amit on the initFileSystem() function of the fsInit.c file where we formatted the volume and wrote the checks to see whether or not we have already formatted the volume (for subsequent boots of the system). I had no issues with completing the latter. However, there were quite a few issues I encountered when completing the former. Thankfully, I had a resolution to every single issue I did have. My first issue was figuring out how to manipulate bits in order to initialize the free space map. My first instinct was to rewatch one of the recorded in-class lectures, as you, Professor, did go over how to do so. That video really helped me understand how the process works, but the implementation details were left to us, the students, to decide. So I experimented, implementing the techniques covered in that lecture, and after all of this experimentation, learning how to shift bits using the bitwise operators and using a whole lot of printfs, I was able to successfully figure out how to manipulate the bits to initialize the map and to write the allocation function. So my resolution was simply rewatching the lecture video and performing lots of testing and abuse printfs.

My second issue revolved around writing the allocation function for the free space. I knew what to do but not how to do it. So I resolved this issue by coming up with a plan. I figured, “Hmm, I can iterate across every byte in the free space map, and in every byte, I can iterate across every bit. I can track how many consecutive 1s (indicating a free block) I encounter, and if we encounter at least one 1, we store the starting block where we found that first 1. If we encounter a 0 before having the number of consecutive 1s equaling the number of blocks the caller wants, we reset the starting block number to 0. However, if the number of consecutive 1s matches the number of blocks the caller wants, we keep that starting block number, clear those bits starting at that starting block number, and return the starting block number of that number of consecutive 1s.” It was a plan that took me a couple of hours to create, but it was well worth it. My resolution was simply to plan the process out on the drawing board.

Milestone 2:

I, Essa, did not have too many issues during Milestone 2. Writing the free blocks function for the free space, revising the directory system, planning out parse path for the group (as well as implementing my parse path integration function, parsepath()), fs_rmdir, and fs_stat, were all pretty simple and straight forward to implement. The only two things I struggled with, as described in the "Approach/What I Did" section, were writing the extent allocation function, blockAllocator(), and handling the special cases in mkdir where I needed to allocate more space for a parent directory in which I want to create a new directory.

The allocation function that accepts an extents table as a parameter was kind of a doozy to implement. The first part of it was easy, as I described in the "Approach/What I Did" section, as all I had to do was first check if we can just give the caller a contiguous set of free blocks of a desired number. If so, just fill the first extent with the starting block number and a count. The next part, handling the situation where this is not the case, where we can't find a contiguous set of free blocks of a desired number, took a lot of thinking. I eventually realized, "Okay, so I'm going to need to get a minimum number of contiguous blocks. Let's have a #define called "MINIMUM_NUMBER_OF_EXTENT_BLOCKS" where we assign each extent a minimum number of extent blocks (I set it to a value of 3). Then, to get a count, I'm going to first find how many consecutive bits of value 1 there are past this point. I can write a helper function for that." And that's what I did. I wrote a helper function called "getNextBit" which returns the number of 1s after "MINIMUM_NUMBER_OF_EXTENT_BLOCKS" number of blocks. I then added this number to "MINIMUM_NUMBER_OF_EXTENT_BLOCKS" and got the actual count for this extent. And then I'd have to decrement a count of how many blocks we have left to allocate called "numOfBlocksLeft". I then repeated this process until all of the desired number of blocks were allocated. So the resolution was just a series of drawing out the problem, visualizing the free space, and planning to get this difficult part done.

The next issue I had revolved around two special cases for fs_mkdir. The first case was the case in which we can find blocks adjacent to the range of the parent directory's LBA extent positions, meaning, we can just add to the count of one of the extents since the range of the extent has increased. The second case is the case in

which the previous case doesn't apply, where we'll unfortunately need to resort to creating a new extent entry with a new starting block number and a new count. Let's begin with the first case. To go about allocating more space, I figured that I'll need to iterate across the extent table of the "." entry of the parent directory (since "." stores info about the current directory) and see if we can find a starting block number of "EXTRA_BLOCKS_FOR_NEW_DIR" number of free blocks that is adjacent to an extents range. Once I do encounter such a starting block number, I can increment that extent's count by "EXTRA_BLOCKS_FOR_NEW_DIR" number of blocks. I then figured that I'm going to have to call realloc() to allocate more space for the parent directory, clear some bits (marking them used), creating a new directory, and finally, loading each of the parent's children and modifying their ".." entries as, since ".." stores the extents of the parent directory, and since that parent is now modified, we want to reflect those changes in the children so that "cd .." would properly load the entire parent directory. So, if we had a situation where an already existing extent entry had a starting block number of 20 and a count of 3, the new count would be 3 + EXTRA_BLOCKS_FOR_NEW_DIR, which in our system, we assigned a value of 3, so the new range of that extent would now be 20-25. And I handled the second case a similar way albeit I didn't modify the count of an extent, I actually had to create a new extent entry since we don't have adjacent blocks to allocate. So I'd say that just like for the previous issue, the resolution was just lots and lots of thinking and planning in order to get this function done.

Milestone 3:

The issue I had with implementing the "mv" command lied in the fact that metadata between two directory entries were being switched and rearranged from one another. For example, when I wanted to move "students" into a directory I called "wow", for some reason, the new entry in "wow" was called "wow". It took me a while to realize that I had to call parse path not one after the other, but once I had obtained the name of the final element of the source. Only then did things start working properly and data wasn't being intermingled. So the issue was resolved with a little bit more thinking and strategizing.

Mahdi:

Milestone 1:

I had an error in determining the correct size of the dirEntries array or in the loop bounds for iterating through the array, leading to out-of-bounds access errors. I also encountered errors when allocating memory using malloc(), such as failing to allocate enough memory or allocating memory that is not freed later, leading to memory leaks. encountered errors when initializing the dirEntries array or modifying its contents, leading to data corruption errors or undefined behavior. And the most difficult part of this milestone was combining everybody's code together, for my part, my code needed some variables that were initialed in someone else's code, and making everything compatible was the hardest part. We all created our own separate branches and pushed our code so if there was anything wrong with our code won't break the whole code, and after talking to the professor we realized that wasn't the best thing to do because we were working on the same file, and only 1 person was pushing to the main branch and we were all relying on him. for the next milestone, we are going to create different files instead of different branches, so we can all push to the main branch.

Milestone 2:

Fs_delete:

As part of my task to remove files from the file system, I needed to free the blocks associated with the file using the freeBlocks() function with the extents array of the directory entry. Since one of my group members had already worked on the freeBlocks() function in milestone 1, I took the time to review his code thoroughly to ensure that I fully understood its functionality. During this process, I also had a few questions about his code, which I approached him with and clarified any doubts I had. This allowed me to confidently use the freeBlocks() function in my code, knowing that it was well-implemented and met the necessary requirements. I also needed to ensure that the correct blocks were being freed when deleting a file. And I used the freeBlocks function with the extents array of the directory entry to free the blocks associated with the file. Also, I needed to locate the correct directory entry for the file to be deleted. So I used the parsePath function to get the directory entry of the parent directory for the file to be deleted, then used the locateEntry function to locate the correct entry for the file.

fs_isFile:
I had difficulty in determining if a given path refers to a directory or file. I had to use a helper function fs_isDir() to determine whether the path refers to a directory or not, and then use that information to decide whether the path refers to a file or not. The other issue I had was Invalid path names or errors in the parsePath() function that could lead to incorrect results. And I fixed that by testing the parsePath() function and ensuring that it returns the expected results for a variety of valid and invalid paths.

fs_isDir:
I had some issues with Memory allocation. parsePath() might allocate memory dynamically using malloc(), which could lead to memory allocation errors if not handled properly. To fix this issue, I had to make sure that the memory allocated by parsePath() is properly freed after it was no longer needed. Also, in Input validation, parsePath() might accept invalid or malformed input, which could lead to unexpected results or errors. To fix this issue, I had to add an input validation code to check if the input is valid and handle errors appropriately. The next issue I had was error handling, parsePath() might encounter errors during execution, such as if the input path is invalid or if a file or directory doesn't exist. To fix this issue, I had to make sure that parsePath() handles errors properly by returning appropriate error codes or messages to the caller.

Milestone 3:
At the beginning, I was overthinking the b_seek function, and after talking to one of my group members I realized that b_seek() should have similar functionality as lseek(). It simply modifies the filePosition (which means the offset from the start of the file) in our b_fcb structure. It should implement three different operations on the filePosition depending on the "whence".

Amit:

Milestone 1: While writing the root directory to disk, I had an issue where LBAWrite() would not write the root directory to disk. I couldn't figure out why because I had just malloced my buffer to the size of the root directory which is 13 blocks or 6,656 bytes. The pointer itself was not null, otherwise I would've gotten a segmentation fault. I passed the correct block count and position. I went on a deep dive into the C pointers online and realized that I was passing in the address of the pointer into LBAWrite(). This caused LBAWrite to write to disk a buffer that was only 8 bytes, the size of the pointer instead of the size of the root directory which is 6,656 bytes.

My second issue occurred with storing the results of my methods so that they could exist outside of my file. I went on a deep dive online, and found that a structure would

be the best way to store the results of my calculations. I created a struct called directoryEntryCalcs which contains three variables. each of int type. The first variable is called totalBytesForEntries, this variable stores the result of my numBytesForEntries() method. The second variable is called totalBlocks to ask, this variable stores the result of my numBlocksToAskFreeSpace() method. The third and final variable is called totalExtraEntries, this variable stores the result of my numExtraEntries() method. This structure allowed me to reference the results of my calculations anywhere in our program.

Milestone 2:

Issue 1 (Load directory): While I was testing my code I realized that I had not accounted for loading a fragmented directory or a directory with multiple extent entries. For example, I may read part of the directory from the first extent entry, and read the other part from the second extent entry.

Resolution: In order to resolve this issue I will maintain an offset which stores how many bytes I have already copied to my buffer. I will increment the offset every time I call LBARead(). The offset will be incremented by the total number of bytes read + the total number of bytes in the next extent entry.

Issue 2 (fs_readdir()): While I was writing my code I realized that I needed to make sure that I wasn't going beyond the length of my directory. Since the length of every directory is the same this was easy to rectify.

Resolution: In order to resolve the issue, I will check that the directoryEntry position variable is not greater than the total number of entries in the current directory. I will have to make this check every time directoryEntry position is incremented and every time fs_readdir() is called.

Milestone 3:

Issue 1(b_read()): While I was testing my code for b_read(), I realized that there wasn't a way for me to test my code against a fragmented file. Since our freespace map is always going to try and allocate as many contiguous blocks as possible there isn't any easy way to test my b_read() against a file with multiple extent entries.

Resolution: In order to resolve this issue, I decided to hard code the extent table. My test file takes a total of 23 blocks. I decided to make the first entry start at block 20 and have a count of 11 blocks. The second entry will start at block 31 and have a count of 12 blocks. This will allow me to test with a fragmented file.

Issue 2(b_read()): While I was testing my code, b_read() kept returning a -1. I found that when I checked to see if I had permission to read the file, that I didn't have permission. Even though the flags were set properly in b_open().

Resolution: In order to resolve this issue, I contacted Jinwei, who had been working on b_open(). Together, we debugged his code and found that the read only flag was not being properly set.

Jinwei:

Milestone 1:

While implementing the method, I have an issue getting the total size of the directory entry array. Usually, we will just use sizeof() to get the length of the malloc length. But after I pass the directory entry array to my method as the argument, the sizeof() always returns 8. So I assume that it is returning the size of the pointer since the array is passed by passing as a pointer to the memory address of the array. I search on Google to see if there are similar occurrences. Then my assumption was proved to be right. Thus, I add another parameter to hold the size of the array so I need not get the size depending on the argument.

Milestone 2:

While implementing my routine tokenizationOfPath(), we have a declaration in our header file that my routine will return a pointer to the array of some character pointers. But later I find out, I am not able to return such a pointer that can correctly point to an array. The returned pointer does not allow us to access the char * other than the first char *. I try to browse solutions and explanations. Finally, it turns out that c is unlikely to allow such logic to be implemented. Thus, I change my strategy. Instead of returning a pointer to an array of some character pointers I will just take a holder in the parameter

list and fill in the holder after tokenizing the path. Then the problem is solved and all the tokens can be read from the holder.

While implementing the fs_setcwd, I have an issue with the return result of parsePath. I need to know the parent directory of the last element of the path in order to check if the last element is a directory or not. Thus, I ask Essa for help and finally have an idea to approach my desired information. The parsePath will return a struct containing the information of the parent directory and the name of the last element. Then I can continue the implementation. Also, I have an issue storing the absolute path of the current working directory. Firstly, I try to keep storing a char array of absolute paths in my own structure. But it turns out very complicated and redundant to handle appending and trimming characters from the char array. Thus, I change my approach to storing the tokens in the structure. Then when the fs_getcwd is called, the tokens will be utilized to generate an absolute path for the current working directory.

Milestone 3:

After implementing b_open(), I try to test it with the command “touch” in fsshell. I find out the new file I just created does not show up after I restart the system. I look at my code and realize I didn’t write the change to disk. Therefore, I need to write the directory back to disk so that the change in the directory entries can be stored.

The most challenging thing in b_write is to organize the logic and decide the steps. I rewrite the whole b_write function twice to finally have it consider all the situations. One big challenge is since we are using contiguous extend blocks as our allocating logic, I need to figure out a way how to retrieve the block number of the file’s allocated blocks. Thus, I write a helper function to pass in an ententNode array and the current block number. The helper function will check if the current block number is included in the extentNode array. If true, then the helper function will return me the next available block number if exists.

After implementing b_write, I use “cp2fs” to check the running status of b_write. Later I make a helper function to print out the content that has been written to disk. While I write to an empty file, the outcome is correct. The file from the Linux system can be copied into our file system correctly. But once I try to truncate an existing file and write to it, the previous content still exists in the block. Thus, our free block function doesn’t correctly release the block. Then, I utilize a buffer as a helper. I malloc it with the size of a block, call memset() to set the buffer to empty and write the buffer to the blocks that I need to free. Later I initialize the file’s extentNode Array to all zero. Then finally, I can correctly rewrite the file.

Screenshot of the Compilation:

```
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o freespace.o freespace.c -g -I.
gcc -c -o volumecontrol.o volumecontrol.c -g -I.
gcc -c -o directories.o directories.c -g -I.
gcc -c -o parsepathintegration.o parsepathintegration.c -g -I.
gcc -c -o parsepathtokenization.o parsepathtokenization.c -g -I.
gcc -c -o parsepathtraversal.o parsepathtraversal.c -g -I.
gcc -c -o parsepathload.o parsepathload.c -g -I.
gcc -c -o fs_mkdir.o fs_mkdir.c -g -I.
gcc -c -o fs_getcwd.o fs_getcwd.c -g -I.
gcc -c -o fs_setcwd.o fs_setcwd.c -g -I.
gcc -c -o fs_rmdir.o fs_rmdir.c -g -I.
gcc -c -o fs_stat.o fs_stat.c -g -I.
gcc -c -o fs_isDir.o fs_isDir.c -g -I.
gcc -c -o fs_isFile.o fs_isFile.c -g -I.
gcc -c -o fs_delete.o fs_delete.c -g -I.
gcc -c -o fs_openrdclose.o fs_openrdclose.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -c -o b_io_helper.o b_io_helper.c -g -I.
gcc -c -o fs_move.o fs_move.c -g -I.
gcc -o fsshell fsshell.o fsInit.o freespace.o volumecontrol.o directories.o pars
epathintegration.o parsepathtokenization.o parsepathtraversal.o parsepathload.o
fs_mkdir.o fs_getcwd.o fs_setcwd.o fs_rmdir.o fs_stat.o fs_isDir.o fs_isFile.o f
s_delete.o fs_openrdclose.o b_io.o b_io_helper.o fs_move.o fsLow.o -g -I. -lm -l
 readline -l pthread
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > exit
System exiting
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$
```

Tables of What Do and Do Not Work:

FUNCTIONS THAT WORK	FUNCTIONS THAT DO NOT WORK
fs_mkdir	
fs_rmdir	
fs_stat	
fs_isFile	
fs_isDir	
	fs_delete
fs_opendir	
fs_readdir	
fs_closedir	
fs_getcwd	
fs_setcwd	
fs_move (created to implement “mv”)	
b_open	
b_read	
b_seek	
b_write	
b_close	

COMMANDS THAT WORK	COMMANDS THAT DO NOT WORK
md	
rm (does not work for fs_delete)	
cd	
pwd	
mv	
cp	
ls	
cp2l	
cp2fs	
touch	
cat	

Screenshots of the Execution:

The "md" command in action (as well as "cd" and "pwd" by consequence):

Creating directories named "home" and "wow" via *relative* paths:

First creating “home”:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > md home
Prompt > cd home
Prompt > pwd
/home
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Next creating “wow” within “home”:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > cd home
Prompt > pwd
/home
Prompt > md wow
Prompt > cd wow
Prompt > pwd
/home/wow
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Creating directories named “student”, “documents”, and “csc415” via ***absolute*** paths:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > md /home/student
Prompt > md /home/student/documents
Prompt > md /home/student/documents/csc415
Prompt > pwd
/
Prompt > cd home
Prompt > pwd
/home
Prompt > cd student
Prompt > pwd
/home/student
Prompt > cd documents
Prompt > pwd
/home/student/documents
Prompt > cd csc415
Prompt > pwd
/home/student/documents/csc415
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Here is the beginning of the free space map ***<u>before</u>*** creating all of these directories:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 5 --start 2
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 1F FF FF FF FF FF  FF FF FF FF FF FF FF FF | ...◆◆◆◆◆◆◆◆◆◆◆◆◆
000410: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000420: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000430: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000440: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000450: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000460: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000470: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000480: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000490: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000500: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000510: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000520: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000530: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

Here is the beginning of the free space map ***after*** creating all of these directories to prove that “md” works:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 5 --start 2
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 00 00 00 00 00 00  00 00 0F FF FF FF FF FF | ............◆◆◆◆◆
000410: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000420: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000430: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000440: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000450: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000460: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000470: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000480: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000490: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

000500: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000510: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000520: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000530: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

As can be seen, multiple bits have been cleared (from addresses 000 402 - 000 40A), or set to values of 0 where 0 = used, after creating these directories. The original state after creating the VCB, free space map, and the root directory, which is displayed in the “before” screenshot, has now been altered and more bits have been cleared as a result of allocation for these directories. Every directory is 13 blocks long in the logical block array which is why there’s much space taken per directory.

<u>The “cd” and “pwd” commands in action (more specific actions):</u>

Switching from the root directory to /home/student/documents/<u>csc415</u> via *<u>relative</u>* paths, and then back to the root directory via a series of “cd ..” commands:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > cd home
Prompt > pwd
/home
Prompt > cd student
Prompt > pwd
/home/student
Prompt > cd documents
Prompt > pwd
/home/student/documents
Prompt > cd csc415
Prompt > pwd
/home/student/documents/csc415
Prompt > cd ..
Prompt > pwd
/home/student/documents
Prompt > cd ..
Prompt > pwd
/home/student
Prompt > cd ..
Prompt > pwd
/home
Prompt > cd ..
Prompt > pwd
/
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Switching from the root directory to /home/student/documents/csc415 via *one absolute path*, and then back to the root directory via the “cd /” command:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > cd /home/student/documents/csc415
Prompt > pwd
/home/student/documents/csc415
Prompt > cd /
Prompt > pwd
/
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

<u>The “rm” command in action (as well as “cd” and “pwd” by consequence):</u>

Deleting directories “home”, “student”, “documents”, and “csc415” via *<u>relative</u>* paths:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > cd home/student/documents
Prompt > pwd
/home/student/documents
Prompt > rm csc415
Prompt > cd csc415
Could not change path to csc415
Prompt > cd ..
Prompt > pwd
/home/student
Prompt > rm documents
Prompt > cd documents
Could not change path to documents
Prompt > cd ..
Prompt > pwd
/home
Prompt > rm wow
Prompt > cd wow
Could not change path to wow
Prompt > rm student
Prompt > cd student
Could not change path to student
Prompt > pwd
/home
Prompt > cd ..
Prompt > pwd
/
Prompt > rm home
Prompt > cd home
Could not change path to home
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Deleting directories “home”, “student”, “documents”, and “csc415” via ***absolute*** paths:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > rm /home/student/documents/csc415
Prompt > rm /home/student/documents
Prompt > rm /home/student
Prompt > rm /home/wow
Prompt > rm /home
Prompt > cd home
Could not change path to home
Prompt > pwd
/
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Here is the beginning of the free space map ***before*** removing all of these directories after using the “md” command from the previous screenshots:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 5 --start 2
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 00 00 00 00 00 00  00 00 0F FF FF FF FF FF | ............◆◆◆◆◆
000410: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000420: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000430: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000440: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000450: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000460: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000470: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000480: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000490: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

Here is the beginning of the free space map ***after*** removing all of these directories to now prove that “rm” works:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ ./Hexdump/h
exdump.linux SampleVolume --count 5 --start 2
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 1F FF FF FF FF FF  FF FF FF FF FF FF FF FF | ...◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000410: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000420: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000430: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000440: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000450: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000460: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000470: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000480: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
000490: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004A0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004B0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004C0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004D0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004E0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
0004F0: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

As can be seen, multiple bits have been freed (addresses 000 402 - 000 40A), or set to values of 1 where 1 = free, after removing these directories. This state is the state of the free space map at its very beginning if you recall the hexdump of the free space we included when describing it. The reason there is 0x00 00 1F, or 0b00000000 00000000 00011111 in binary, is because, as we explained before in the analysis of the free space, there are blocks allocated for the VCB, the free space map itself, and the root directory. Therefore, directories, in fact, have been removed from our system.

<u>The “mv” command in action:</u>

We create three directories: “wow”, “home”, and “student” (which is contained in “home”). The we move “student” from “home” to “wow” via *<u>relative</u>* paths, as can be seen here:

```
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > md home
Prompt > md /home/student
Prompt > md wow
Prompt > cd home
Prompt > ls -a

.
..
student
Prompt > mv student /wow/student
Prompt > ls -a

.
..
Prompt > cd ..
Prompt > cd wow
Prompt > ls -a

.
..
student
Prompt > pwd
/wow
Prompt > ls -a

.
..
student
Prompt > exit
System exiting
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$
```

Now, let’s move “student” back into “home”:

```
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > cd wow
Prompt > ls -a

.
..
student
Prompt > mv student /home/student
Prompt > ls -a

.
..
Prompt > cd ..
Prompt > pwd
/
Prompt > cd home
Prompt > ls -a

.
..
student
Prompt > exit
System exiting
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$
```

Now, let's move "student" again to "wow" via an *absolute* path, as can be seen here:

```
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > mv /home/student /wow/student
Prompt > cd home
Prompt > ls -a

.
..
Prompt > cd ..
Prompt > pwd
/
Prompt > cd wow
Prompt > ls -a

.
..
student
Prompt > exit
System exiting
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$ make run
```

One more time, let's move "student" again to "home" via an ***absolute*** path, as can be seen here:

```
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > mv /wow/student /home/student
Prompt > cd wow
Prompt > ls -a

.
..
Prompt > cd ..
Prompt > pwd
/
Prompt > cd home
Prompt > ls -a

.
..
student
Prompt > exit
System exiting
student@student-VirtualBox:~/repoclone/csc415-filesystem-EssaHusary$
```

<u>The “ls” command in action (as well as “pwd” and “md” by consequence):</u>

We start by calling “pwd” to check the current working directory. Then call “ls” to check the child inside the current working directory. It shows no child exist. Then we call “md” to create some directory “home”, “student”. Then we call “ls” again. Now we can see the current working directory has two children. By default, “ls” will not output the “.” and “..”. By calling “ls -a”, it will output the hidden children. “ls” is working.

```
student@student-VirtualBox:~/main/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > ls

Prompt > md home
Prompt > md student
Prompt > ls

home
student
Prompt > ls -a

.
..
home
student
Prompt > pwd
/
Prompt > exit
System exiting
student@student-VirtualBox:~/main/csc415-filesystem-EssaHusary$
```

<u>The “cp2fs” command in action (as well as “pwd”, “cat” and “ls” by consequence):</u>

We start by checking the file and the content in the Linux file system. Then we run our program. We call “pwd” to check the current working directory and “ls” to check the child. It shows there is no child currently. Then we call “cp2fs” to copy the file from Linux file system to our file system and name it “testcp2fs”. We call “ls” and “cat” to check the content. It shows the two file are identical. “cp2fs” is working.

```
student@student-VirtualBox:~/test$ pwd
/home/student/test
student@student-VirtualBox:~/test$ ls
testfile
student@student-VirtualBox:~/test$ cat testfile
/*************************************************************
* Class:  CSC-415-03 Spring 2023
/*************************************************************
student@student-VirtualBox:~/test$ cd ../Documents/csc415-filesystem-EssaHusary/
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > ls

Prompt > cp2fs /home/student/test/testfile /testcp2fs
Prompt > ls

testcp2fs
Prompt > cat testcp2fs
/*************************************************************
* Class:  CSC-415-03 Spring 2023
/*************************************************************
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

<u>The “cp2l” command in action (as well as “pwd”, “cat” and “ls” by consequence):</u>

We start by check the Linux file system and it show there is no file in the folder “test”. Then we run our program and check current working directory. We call “cat” to check the content. After calling “cp2l”, we go back to the Linux file system and locate the new file “testcp2l”. The result shows two files are identical. “cp2l” is working.

```
student@student-VirtualBox:~/test$ pwd
/home/student/test
student@student-VirtualBox:~/test$ ls
student@student-VirtualBox:~/test$ cd ../Documents/csc415-filesystem-EssaHusary/
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > ls

testfile
Prompt > cat testfile
/**************************************************************
* Class:  CSC-415-03 Spring 2023
/**************************************************************
Prompt > cp2l /testfile /home/student/test/testcp2l
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ cd ..
student@student-VirtualBox:~/Documents$ cd ..
student@student-VirtualBox:~$ cd test
student@student-VirtualBox:~/test$ pwd
/home/student/test
student@student-VirtualBox:~/test$ ls
testcp2l
student@student-VirtualBox:~/test$ cat testcp2l
/**************************************************************
* Class:  CSC-415-03 Spring 2023
/**************************************************************
student@student-VirtualBox:~/test$
```

<u>The “touch” command in action (as well as “pwd”, “ls” and “cd” by consequence):</u>

We start by checking the current working directory and checking if there are childs in it. Then we call “touch” to create a file “testtouch”. Call “ls” again to check if the new file exists. Now the file is created. We call “cd” to check if the child is directory or a file.The result shows “”touch successfully create a file. “touch” is working.

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./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
Prompt > pwd
/
Prompt > ls

Prompt > touch testtouch
Prompt > pwd
/
Prompt > ls

testtouch
Prompt > cd testtouch
Could not change path to testtouch
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

<u>The “cat” command in action (as well as “pwd”, and “ls” by consequence):</u>

We start by check the current working directory and list the child. Then we call “cat” to print out the content of the file “testfile”. The file system successfully prints out the text of the file. “cat” is working.

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > ls

testfile
Prompt > cat testfile
/**************************************************************
* Class:  CSC-415-03 Spring 2023
/**************************************************************
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

<u>The “cp” command in action (as well as “pwd”, “cat” and “ls” by consequence):</u>

We start by checking the current working directory and list the child. Then we call “cat” to print the content of “testfile”. After, we call “cp” to make copy this file to generate another file called “testcp”. Call “cat” to print the new file. It shows that two files are identical. “cp” is working.

```
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > pwd
/
Prompt > ls

testfile
Prompt > cat testfile
/*************************************************************
* Class:  CSC-415-03 Spring 2023
/*************************************************************
Prompt > cp testfile testcp
Prompt > ls

testfile
testcp
Prompt > cat testcp
/*************************************************************
* Class:  CSC-415-03 Spring 2023
/*************************************************************
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-EssaHusary$
```

Works Cited Page:

When describing our file system, it helped us to reference the textbook used in this course to define terms such as a “volume” or talking about the layers of the file system. Of course, the lecture videos were used as well. And, we used the Linux Programmer’s Manual by typing in, for example, “man strtok_r” into the Ubuntu terminal to review (and even learn) how to use certain functions such as strtok_r, strcmp, strcpy, time, strlen, memcpy, memset, malloc, realloc, and free. Of course, we already knew how to use most of them, but the man pages are cited here anyways. Also, we used the man pages to aid in our implementations of some of the directory functions and b_io functions, as the man pages gave us some more insight as to how to use commands such “md” and b_io functions such as “read”, in order to implement them accurately.

Citations:

“Linux Programmer’s Manual.” 15 Sept. 2017. (Man pages)

Silberschatz, Avi, et al. *Operating System Concepts*. 10th ed., e-book ed., John Wiley & Sons, Inc., 2018.