CSC 415-01 FILE SYSTEM

Final Project

Team Bug Catchers

Daniel Guo	Github: yiluun
Melody Ku	Github: harimku
Christopher Yee	Github: JoJoBuffalo
Matthew Lee	Github: Mattlee0610

Github Link
https://github.com/yiluun/File-System

CONTENTS

Description of File System

For this final project, our group designed and programmed a simplified version of a working file system. In this project, each block of data on the disk is 512 bytes and our whole system has a size of 9999872 bytes, which is 19531 blocks large. For our file system, we decided that each directory would hold a maximum of 50 directory entries within it. The file system is designed to allocate space in a continuous manner, so the blocks of a particular directory or file are next to each other on the disk. The file system is also able to read data from the disk and write data to the disk using the given LBAread() and LBAwrite() functions. The file system keeps a record of changes made to the disk using a bitmap. The file system also has various functionalities that allow it to create, modify and delete directories and files from our system.

Our group decided to allocate memory on the disk in a continuous manner so that the data of a particular file or directory would be in the blocks next to each other. This had the benefit of allowing us an easy way to initially allocate space on the disk. Since the blocks would be next to each other, we would just need to loop and find a section that could fit what we're writing to the disk. Having a continuous allocation also gives us the benefit of reading data from the disk. The file system would just need to keep a record of the starting block of the data and the size of the directory or file. However, the downside of contiguous allocation would be when we write data to a file using b_write(), and the file doesn't have enough space. In such a case, we would have to first look for a new location to store the data that is also large enough to hold the current data plus the amount that would be written into it. Having such a format would cause fragmentation as files would continue to get scattered across the disk.

The file system that we designed is capable of reading and writing to the disk, as well as keeping a record of any changes. We were given the functions LBAread() and LBAwrite(), which allowed us to read data into our disk and write data into our disk. Our group decided to keep track of changes made on the disk using a bitmap. The bitmap works by having each bit on the map represent a full block of space on the disk. When a bit on the bitmap is set to 0, we know that the block on the disk corresponding to the location of the bit has data and can't be used. Inversely, if a bit is set to 1, we know that the block is free to have data written into it. We also have a function that goes through this bitmap to find out what groups of blocks are free to write data into and that would fit the size of the data being written to it.

This file system has various capabilities for creating, modifying, and removing directories and files. This allows the file system to perform various actions on the directories and files that have been saved to the disk. Some of these capabilities include: making a directory, opening a directory, reading a directory, closing a directory, opening and creating files, reading and writing to files, and closing files. The file system also has multiple structures that are used to make the file system work. Many of the structures are necessary for our file system to work correctly. The most notable ones in the file system include the volume control block structure, file control block structure, directory entry structure, and parse path structure.

Volume Control Block Structure:

We have a volume control block structure that dictates and holds the data for the whole file system. The VCB holds information that many other functions will refer to. One such piece of information the VCB holds is the size of a block. This is required since we are passing in the size of a block and this value could be changed as our file system runs. If we didn't have a way to refer to the size of a block, many other functions wouldn't be able to perform their calculations to obtain values, such as the number of blocks they require.

```
typedef struct VCB {
        long numBlocks;      // total number of blocks for our system
        long numFreeBlocks;          // number of free blocks
        long freeSpaceMap;// to find the free space bitmap
        long magicNum;       // special signature
        long firstBlock;     // this marks the first block of the volume
        long rootDirectory;  // location of root dir (this is a block #)
        int directoryLength; // we will create 50 entries per directory
        int sizeOfBlock;     // the size of a single block, in bytes
        char volumeName[30];
} VCB;
```

Directory Entry Structure:

In essence, the directory entry is an array of directory entries. We have a structure representing the basis of a singular directory entry. Since each directory entry is made up of this structure, it allows us to get information about the entry in an easy way, such as the name of a particular directory entry.

```
typedef struct DEntry {
        long identifier;
        long size;           // size of the directory
        time_t createdTime;// time that it was created
        time_t modifiedTime;         // last time the directory/file was
modified
        time_t accessTime; // last access time
        int location;        // location on the disk
        char isDirectory;    // 'd' = directory, '_' for file
        char fileName[30];   // name of the directory entry
        char owner[30];      // owner of the specific directory/file
} DEntry;
```

File Control Block Structure:

The File Control Block Structure keeps track of the information required by other file-handling functions. When interacting with a file, the FCB is necessary in order to know various pieces of information regarding the file. This is because the only information that is kept in this structure is the state, location, and size of the open file. This information is required when using functions such as b_read(), b_write(), b_seek() and b_close(). For example, b_read() and b_write would need to know the location of the file on the disk and the size of the file in order to read it and write to it.

```
typedef struct b_fcb {
        char * buf;             //holds the open file buffer
        int index;              //holds the current position in the buffer
        int buflen;             //holds how large the buffer is
        int fileAccessMode;     // 0: read only        1: write only   2: read write
        off_t offset;           // where in the file you are
        int fileSize;           // size of the whole file
        long startingLoc;       // location of starting block
        char *pathname;         // stores path to change location later
        int remainingBytes;     //Remaining bytes that need to be read in close
} b_fcb;
```

Parse Path Structure:

The file system also has various other structures that are used to store and transfer multiple types of data between functions. One such instance of this is the structure the parsePath() function returns. This structure carries with it important data about a specific path. Many other functions throughout the file system rely on parsePath() and the structure it returns in order to run properly. This is because parsePath() is able to give much-needed information about a directory entry.

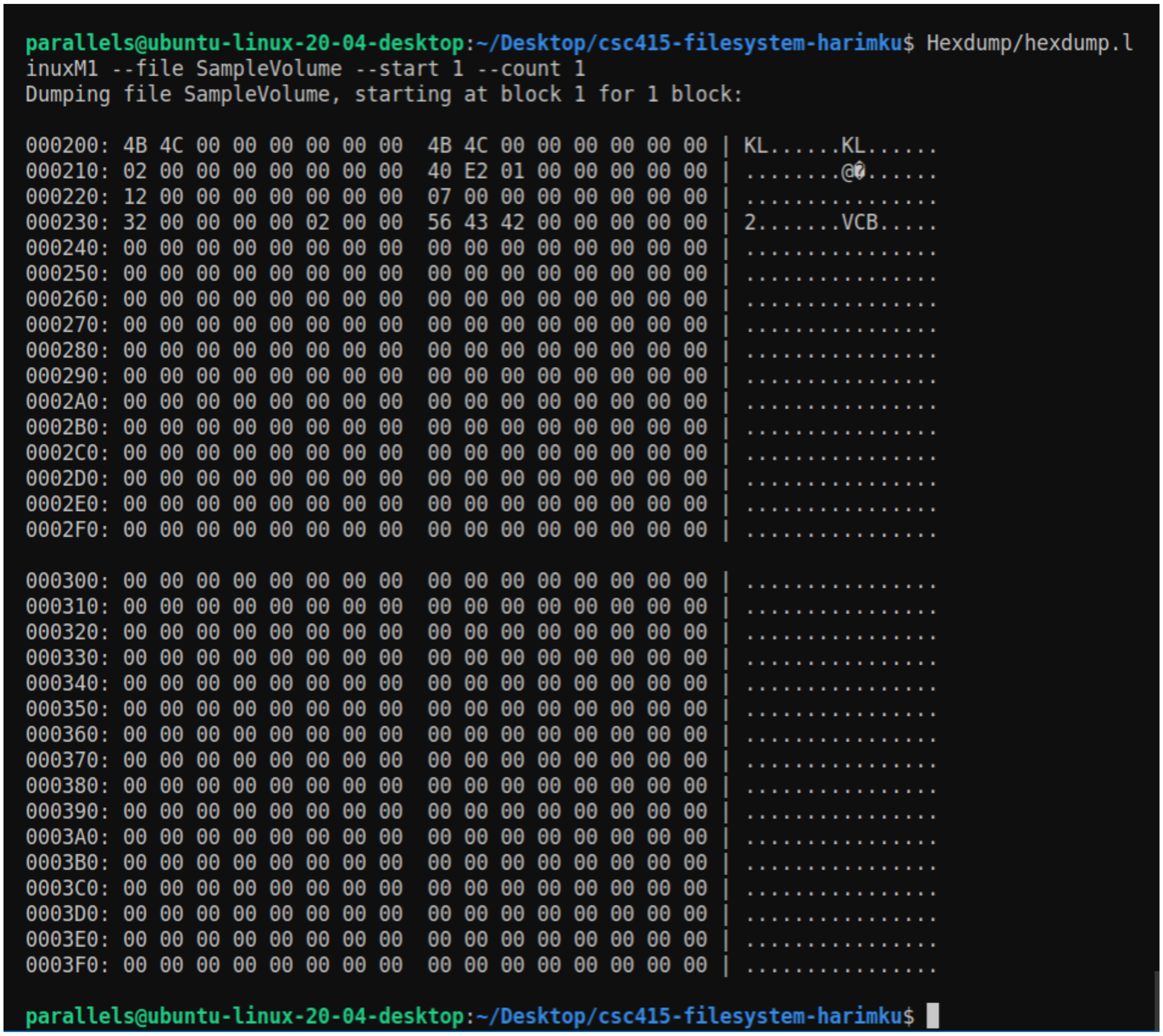
```
typedef struct parsePathStruct {
        int isValidPath;        // 1 = valid, 0 = not valid
        char isDirectory;       // d = directory, _ = file
        DEntry * dirEntry;      // the parent directory
        DEntry * lastValidLevel;        // store ptr to last valid level
        char * nonValidLevels[256];             // store tokens of each
invalid    path level
        char * lastToken;       // value of last token in the path, if path is valid
} parsePathStruct;
```

Hexdump Analysis

Block 0: Partition Record (already given)

Block 1: Volume Control Block

The volume control block (VCB) struct is defined in the following order (refer to Page 2), with the following values in **the initFileSystem** function within **fsInit.c**. The values are to be read in the Little Endian order, while strings are read in the normal order.



```
parallels@ubuntu-linux-20-04-desktop:~/Desktop/csc415-filesystem-harimku$ Hexdump/hexdump.l
inuxM1 --file SampleVolume --start 1 --count 1
Dumping file SampleVolume, starting at block 1 for 1 block:

000200: 4B 4C 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | KL......KL......
000210: 02 00 00 00 00 00 00 00  40 E2 01 00 00 00 00 00 | ........@�......
000220: 12 00 00 00 00 00 00 00  07 00 00 00 00 00 00 00 | ................
000230: 32 00 00 00 00 02 00 00  56 43 42 00 00 00 00 00 | 2.......VCB.....
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 | ................

parallels@ubuntu-linux-20-04-desktop:~/Desktop/csc415-filesystem-harimku$
```

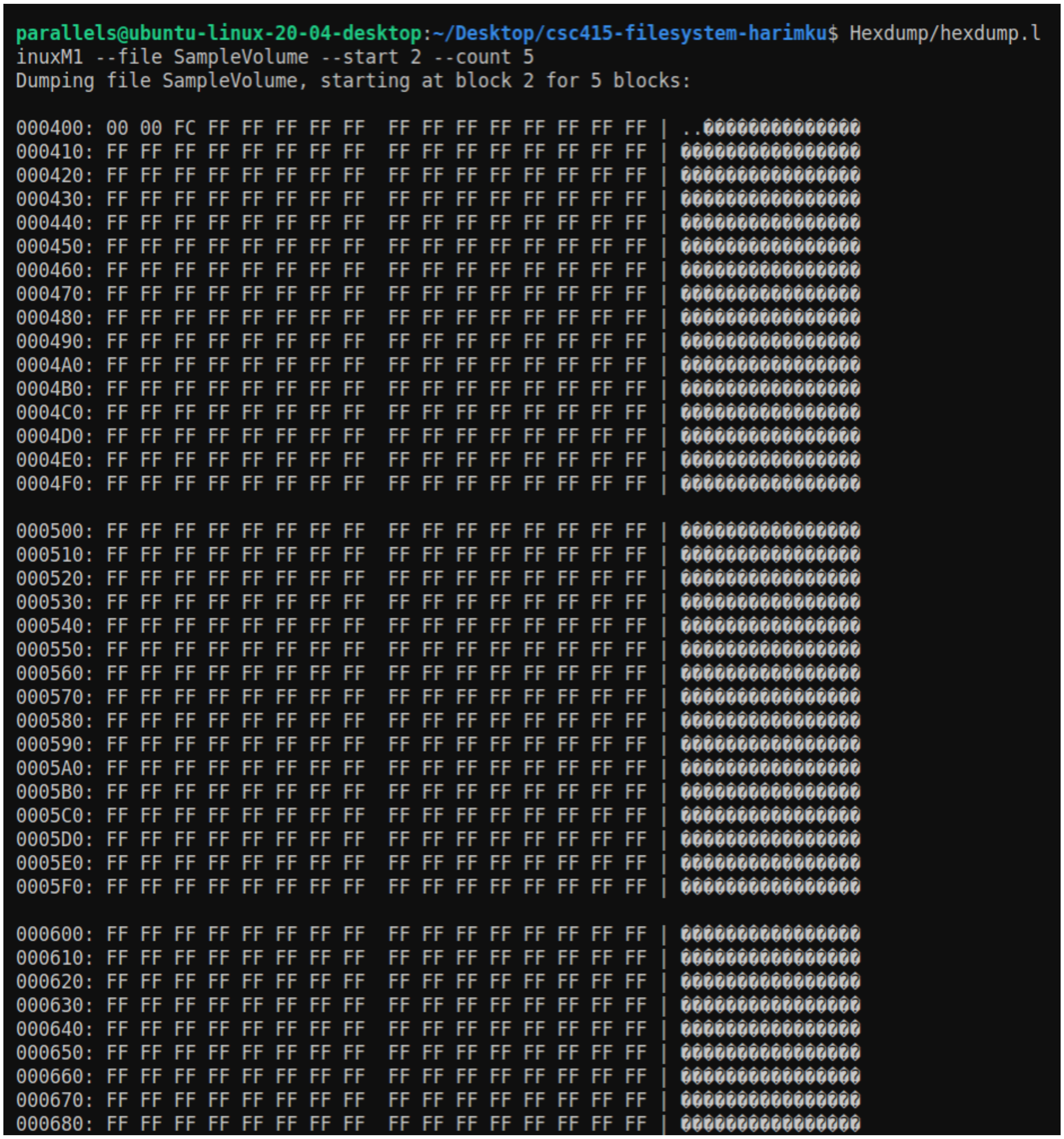
1. long numBlocks = 19531 (4C4B → 19531)
    a. The value stored in the first 8 bytes (size of Long) is 4C4B. This is 19531 in decimal, and matches the value passed in by the test code for our system size.
2. long numFreeBlocks = 19531 (4C4B → 19531)
    a. The value stored in the next 8 bytes is also 4C4B. This is 19531 in decimal. This value should match numBlocks, as the number of free blocks upon initialization is equal to the total number of blocks in the system.
3. long freespacemap = 2 (02 → 2)
    a. The next 8 bytes starting from address 0x000210 is the location of the free space map. This is the block # where the free space map begins. This should be 1, as block 1 is the VCB, and the free space map begins from block 2.
4. long magicNum = 123456 (0001E240 → 123456)
    a. The next 8 bytes are storing our magic number, which I defined as 123456. The value from hexdump is 00 01 E2 40, which is 123456 in decimal.
5. long firstBlock = 18 (0012 → 18)
    a. The next 8 bytes are storing the location of the first free block of the system. This block would be stored after the VCB(block 1), free space map(block 2-6), and finally the root directory (block 7-17) (These block allocations will be explained in more detail in the following section). That means the location of the first block where normal data we create can be stored is the beginning of block 18, which is what we see from the hexdump.
6. long rootDirectory = 7 (0007 → 7)
    a. The next 8 bytes are storing the location of the root directory. The root directory starts from block 7 as explained above, and that is the value we see.
7. int directoryLength = 50 (0032 → 50)
    a. The next 4 bytes (size of int) stores the number of entries in each directory. This is the number that is used in the **createDir** method to create a new directory. We have defined this number as 50.
8. int sizeofBlock = 512 (0200 → 512)
    a. The next 4 bytes store the block size of the system. The number that gets passed in is 512 bytes. That is the value we see in the hexdump.
9. char volumeName = “VCB” (564342 → “VCB”)
    a. Lastly, the volume’s name is stored in the next 3 bytes. We allocated 30 bytes of space, but In the **initFileSystem** method, I strcpy the string “VCB” into this variable. This value also matches expectations.

Block 2-6: Free Space Map

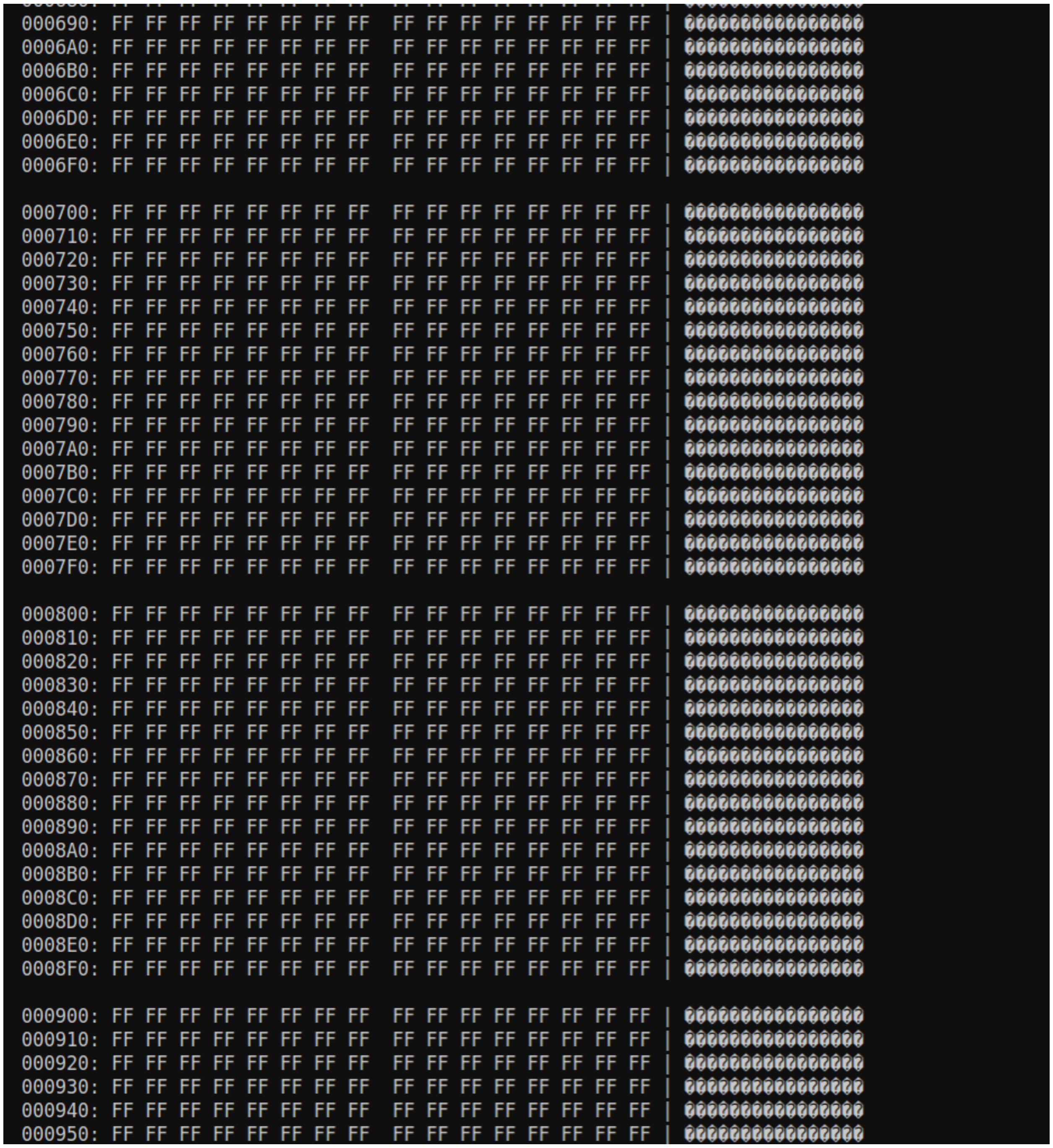
As mentioned already, the value passed in at runtime for our system is 19531 blocks. Our free space map is a bitmap that tracks whether each block is allocated or free. This means we need to have 1 bit for every block in the system.

1. # of bits required = 19531
    a. Since we cannot store individual bits, we must calculate the number of bytes needed to cover this many bits: (# of bits + 8 - 1) / 8 = (19531 + 7) / 8 = 2442.25. → We need 2442 bytes to store all 19531 bits. This calculation is somewhat similar to the way we calculate # of blocks needed for certain data structures. This is needed so we round up to the nearest byte, in order to cover all 19531 bits.
2. # of bytes required = 2442
    a. Now we have the # of bytes needed for our free space map, we need to calculate the number of blocks needed to store it. This is (2442 + blocksize -1) / blocksize. This would be: (2442 + 512 -1) / 512 = 5 blocks in our case.
3. # of blocks needed = 5. This is why the free space map occupies blocks 2,3,4,5,6.

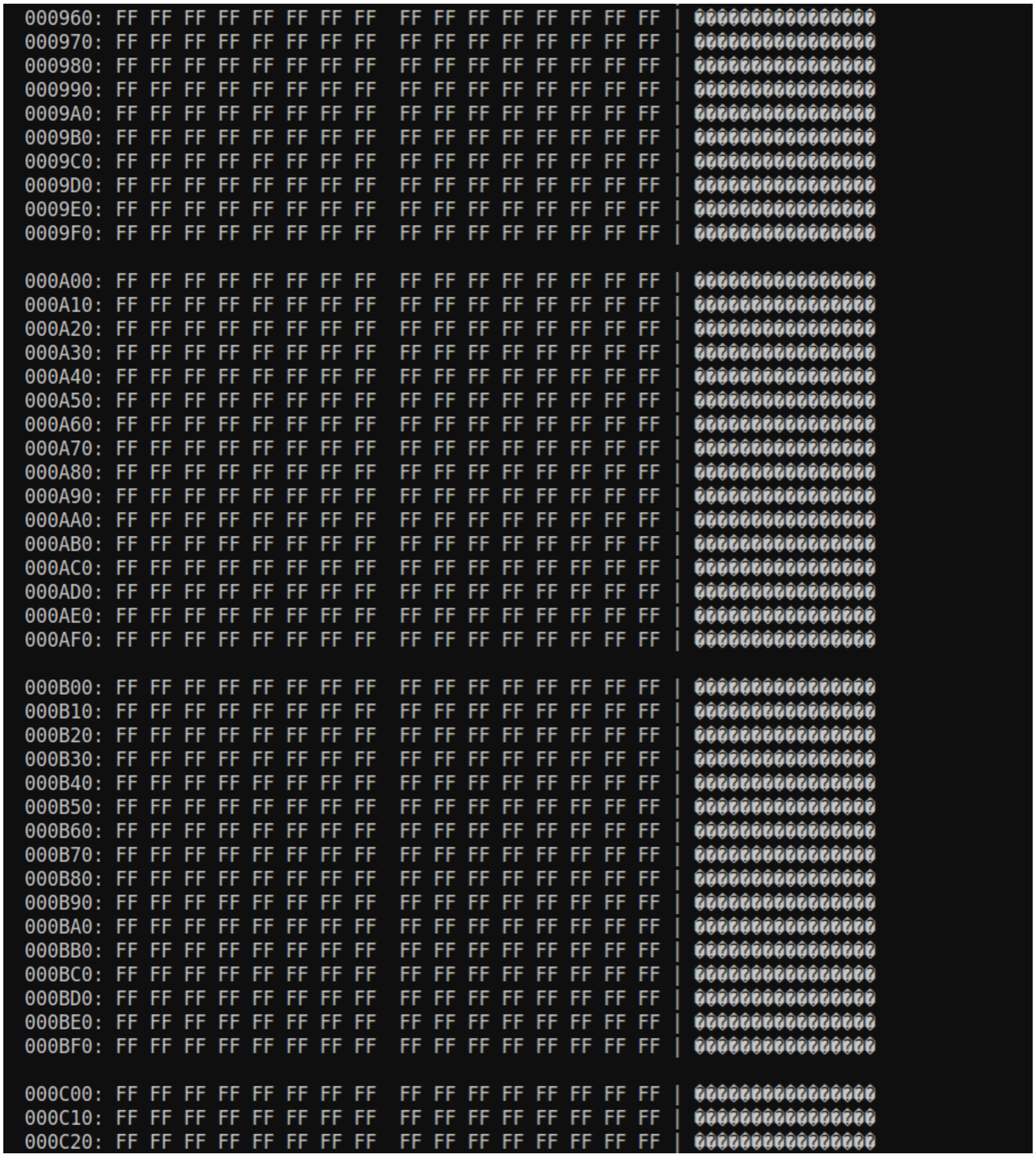
This calculation is performed in the **initFreeSpaceMap** method in fsInit.c. Now let’s look at the hexdump for these blocks. The hexdump is from block 2 to block 6 (see next page).



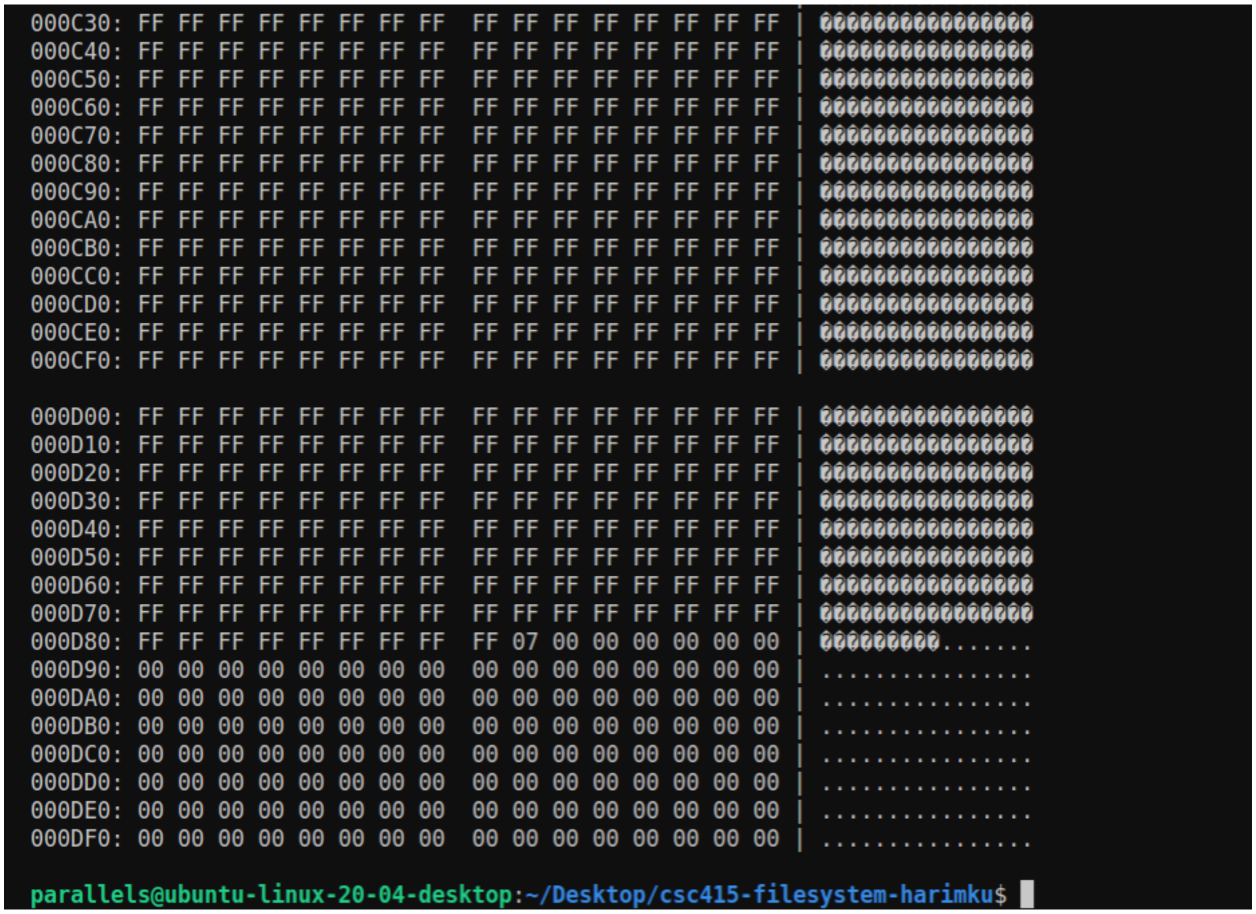
```
parallels@ubuntu-linux-20-04-desktop:~/Desktop/csc415-filesystem-harimku$ Hexdump/hexdump.l
inuxM1 --file SampleVolume --start 2 --count 5
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 FC 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 | ����������������
```

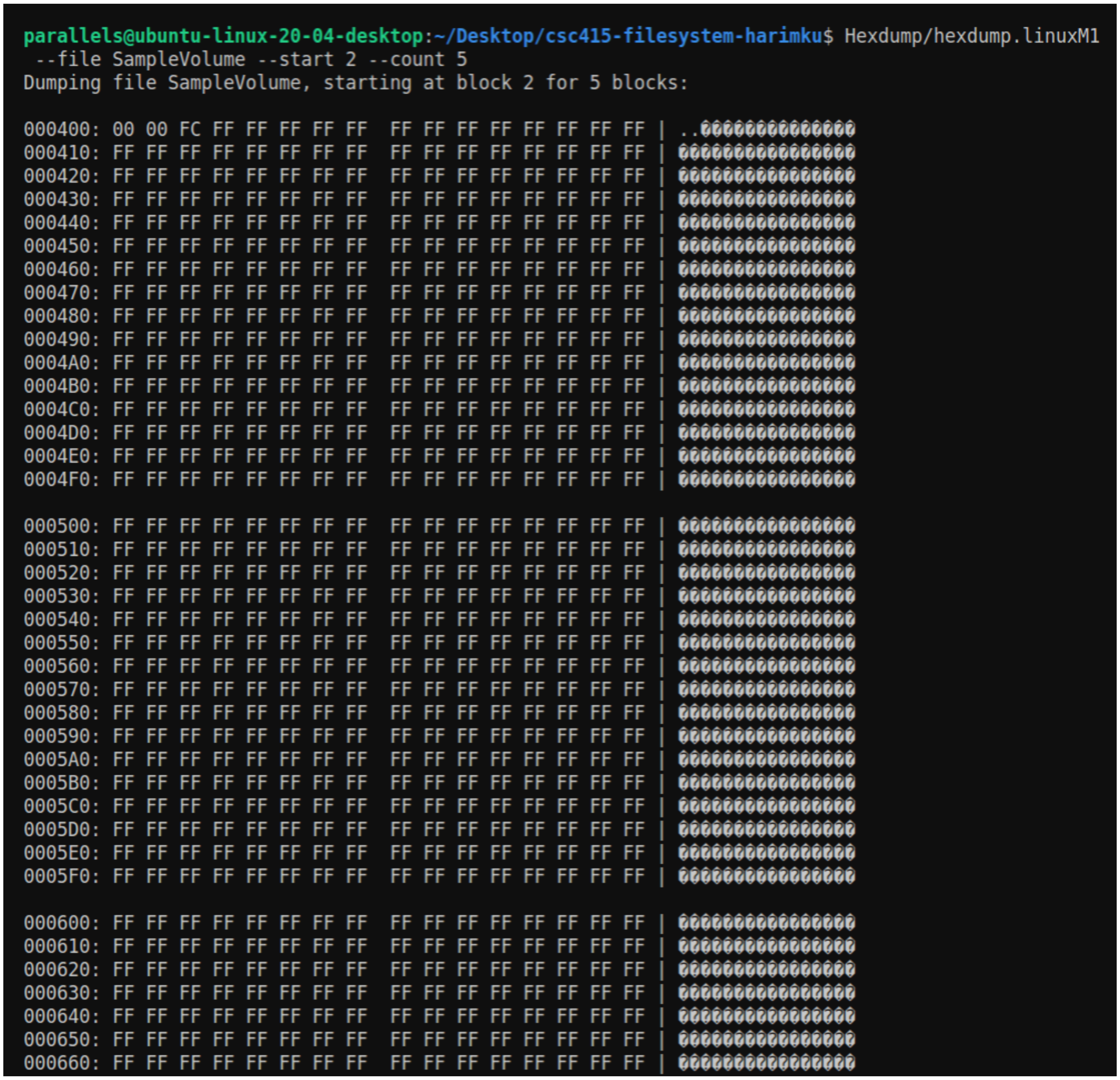


```
000C30: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
000C40: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
000C50: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
000C60: FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF | ����������������
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 07 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 | ................

parallels@ubuntu-linux-20-04-desktop:~/Desktop/csc415-filesystem-harimku$
```

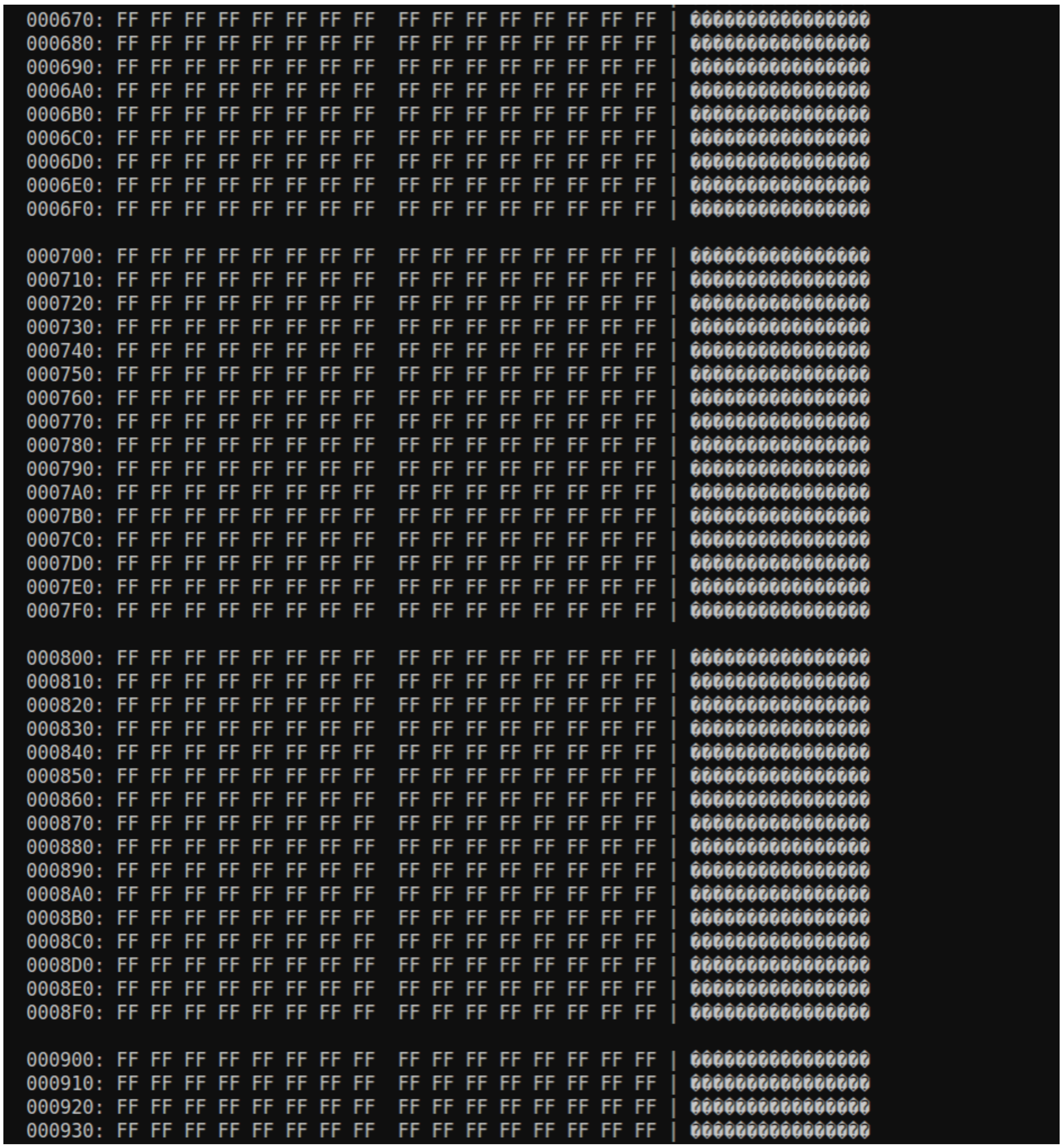
When we initialized the free space map in **initFreeSpaceMap** method, the first 7 blocks (for the partition record, the VCB, and the free space map itself) were marked as allocated, while the rest of the bits were set to be free. Then the **initRootDir** method also marks the blocks occupied by the root directory (11 blocks) as allocated upon the creation of the root directory. In total, this is 1+1+5+11 = 18 blocks allocated from system initialization. Therefore, the next free block in the system is block #18. We can check if free space allocation is performed correctly by checking how many bits are set in our hexdump.

Each byte (8 bits, 2 hex digits) represents 8 blocks of our system.

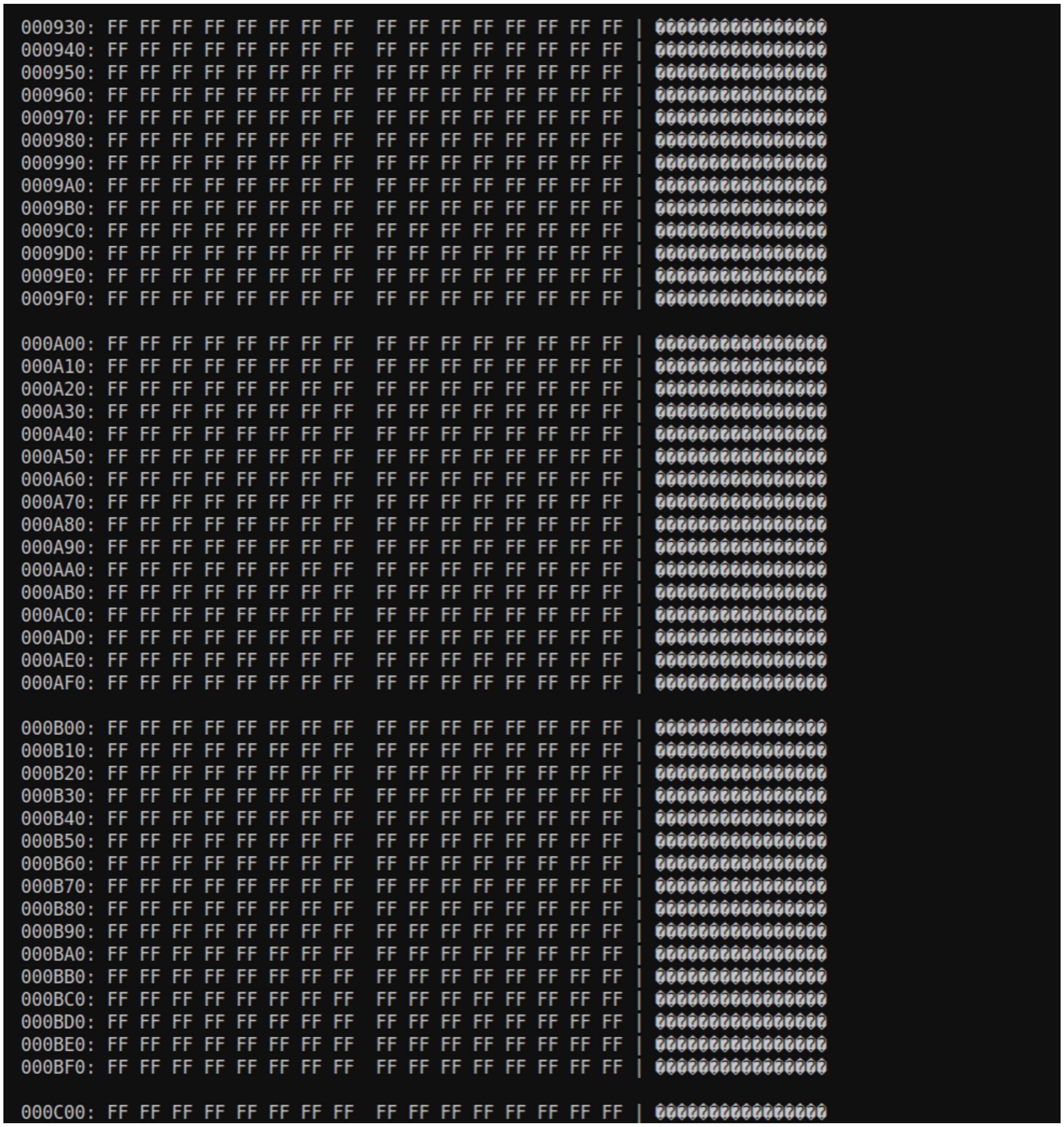
- The first byte is 00 → 00000000 (all 8 bits allocated).
- The next byte is also 00 → 00000000 (all 8 bits allocated).
- Then the next byte is FC → 11111100 (only the first 2 bits are allocated). After this byte is all FFs → 11111111 (all bits are free). In total, our free space map shows the first 8 + 8 + 2 = 18 blocks are allocated upon system start.



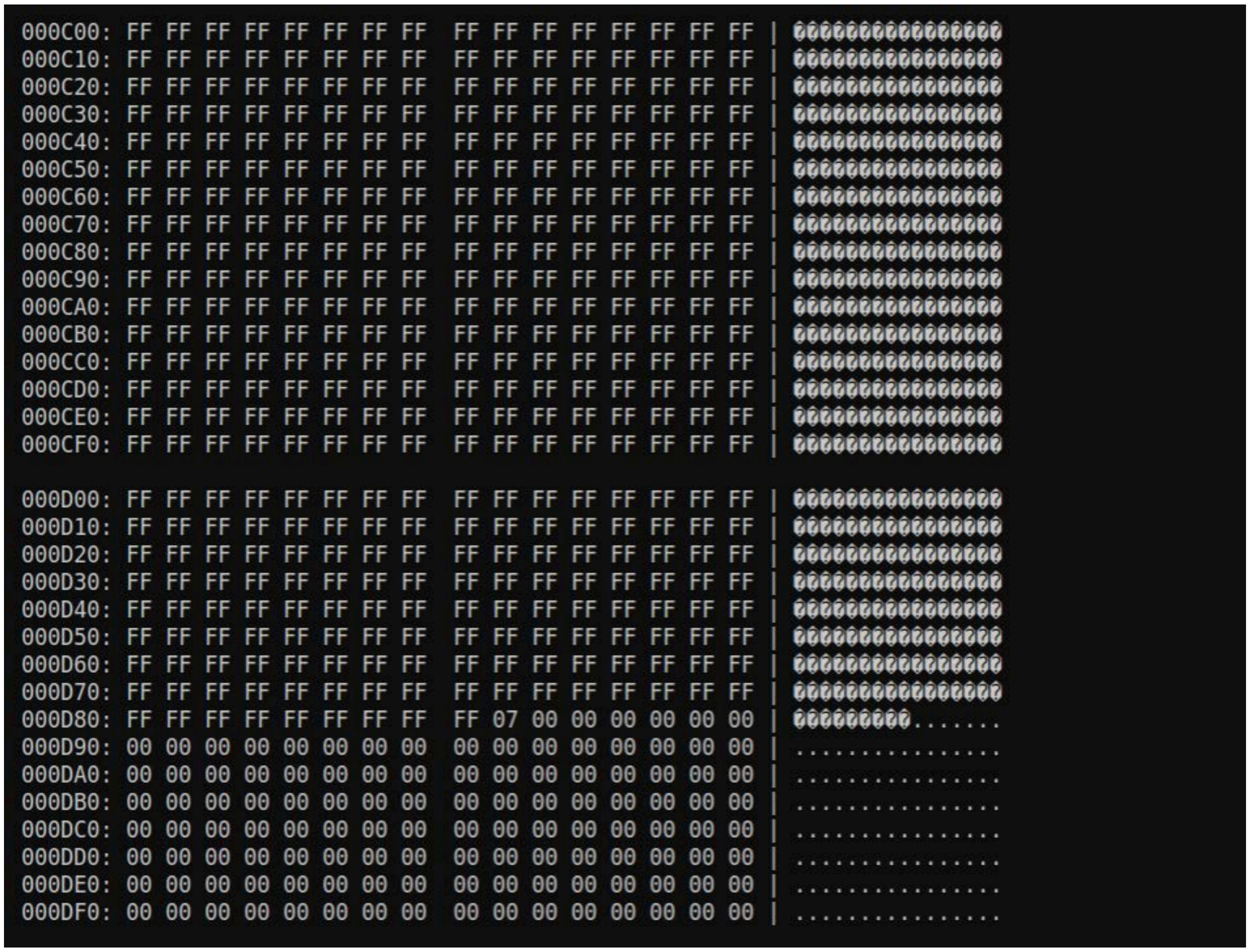
```
parallels@ubuntu-linux-20-04-desktop:~/Desktop/csc415-filesystem-harimku$ Hexdump/hexdump.linuxM1
 --file SampleVolume --start 2 --count 5
Dumping file SampleVolume, starting at block 2 for 5 blocks:

000400: 00 00 FC 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 | ����������������
```



```
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 | ����������������
```



```
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 07 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 | ................
```

The reason why values from 0x000D8A are all 0s again, even though we have marked all remaining blocks as free(1) is because the 19531 bits (2442 bytes) that make up our free space map do not fill the last block entirely. There are 512*5 - 2442 = 118 bytes that are part of the block storing the free space map but aren't actually utilized.

Block 8-18: Root Directory

The root directory, like any directory, consists of directory entries. The first entry stored in the beginning will be the "." entry, which is pointing to the root directory. The next entry will be the ".." entry, but also pointing to itself since this is the root directory.

Each directory entry struct (DEntry) is defined in this order, but let's look at the first entry, which is the '.' entry:

1. Long identifier = 3 (03 → 3)

a. We weren't sure what the unique identifier should be, so we set it to 3 in initRootDir method.

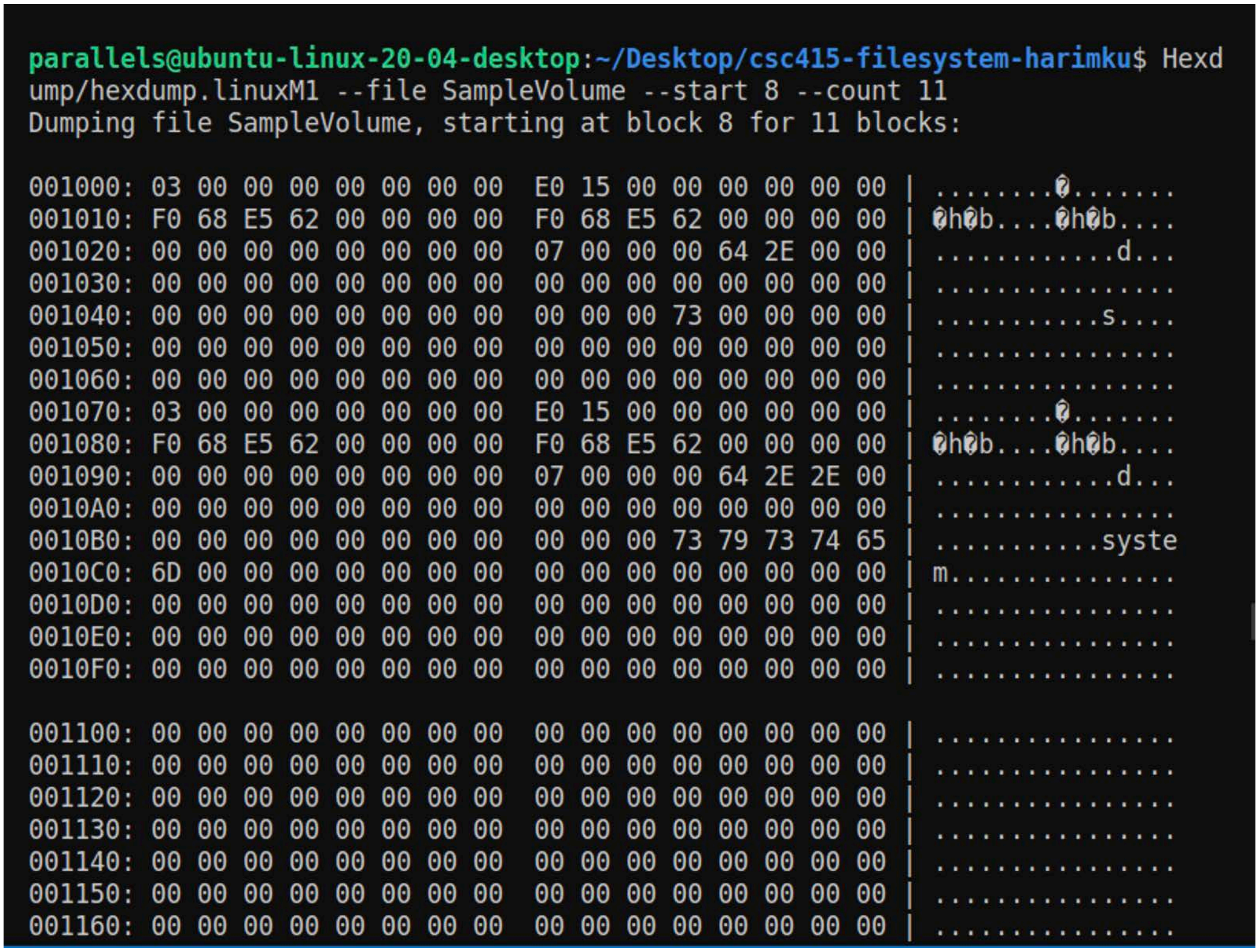
2. Long size = 50 * sizeof(DEntry) = 50*112 = 5600 (15E0 → 5600)

. The space occupied by each DEntry struct is 112 bytes. Multiplied by 50 entries, the directory size is 5600 bytes. This is stored in the "." entry.

3. time_t createdTime (8 bytes storing the time that root was initialized)

4. time_t modifiedTime (8 bytes storing the time that root was initialized)
5. time_t accessTime (8 bytes storing the time that root was initialized)
6. Int location = 7 (07 → 7)
7. Char isDirectory = ‘d’ (64 → ‘d’)
8. Char fileName[30] = “.” (2E → ‘.’)
9. Char owner[30] = “system” (73 79 73 74 65 6D → ‘system’)

The next entry has similar values, as the root directory’s “..” entry stores the same information except for the fileName being “..”. We can see this at address 0x00108D. The value stored there after 64 (‘d’) is 2E2E, which is “..”.



```
parallels@ubuntu-linux-20-04-desktop:~/Desktop/csc415-filesystem-harimku$ Hexd
ump/hexdump.linuxM1 --file SampleVolume --start 8 --count 11
Dumping file SampleVolume, starting at block 8 for 11 blocks:

001000: 03 00 00 00 00 00 00 00  E0 15 00 00 00 00 00 00 | ........�.......
001010: F0 68 E5 62 00 00 00 00  F0 68 E5 62 00 00 00 00 | �h�b....�h�b....
001020: 00 00 00 00 00 00 00 00  07 00 00 00 64 2E 00 00 | ............d...
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 73 00 00 00 00 | ...........s....
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: 03 00 00 00 00 00 00 00  E0 15 00 00 00 00 00 00 | ........�.......
001080: F0 68 E5 62 00 00 00 00  F0 68 E5 62 00 00 00 00 | �h�b....�h�b....
001090: 00 00 00 00 00 00 00 00  07 00 00 00 64 2E 2E 00 | ............d...
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 73 79 73 74 65 | ...........syste
0010C0: 6D 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | m...............
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 | ................
```

Issues and Problems

One of the first issues we faced was determining what would be a good way to manage the free space for our file system. We needed a way to maintain the free space list and keep track of the blocks on the disk that have and have not been allocated. Our team decided between using a bitmap, using grouping, or using a linked list to keep track of this information. Ultimately, we decided to utilize a bitmap due to its simplicity to understand, as we didn't want to overly complicate our file system. Thinking further upon our decision to use a bitmap, we realized that the time complexity for the bitmap search would be O(n). We think that we could increase the search speed in future iterations of our file system with multithreading, to help with the linear search speed through the bit array, if that ever proves to be an issue.

Our team also had issues with overthinking the code. We sat down to discuss the logic behind each function without fully understanding each and every structure. We also started discussing the logic of a function and went down a rabbit hole of dealing with every special case we could think of, instead of first dealing with the main issues. Modifying basic code to deal with a few special cases is easier than jumping into a complex conditional function at the start. At first, we also failed to break down our logic into smaller pieces to be used in helper functions. This would have greatly helped both our debugging and understanding of the functions. Eventually, after repeatedly failing to progress in our implementation of our parsePath() function, we decided to visually draw out the logic of the basic functions and break them down into smaller pieces to simplify the complexity. Visualizing the flow of code on paper was extremely helpful in understanding our function and even debugging our code before implementation.

Another issue that our group faced was time constraints from being bottlenecked by the two most important functions. Being stuck on parsePath() and loadDir() implementations meant that it was difficult for us to move on to other functions. It took a lot of trial and error to figure out what other structure members we needed for parsePath() to return. Not knowing what the parsePathStruct needed to contain initially caused us to continuously go back into parsePath to add small bits of code to accommodate the new members. parsePathStruct was completed when we had gone through each of the functions that relied on parsePath() and gave the struct the necessary members to store. With parsePath() and loadDir() done, the logic was greatly simplified, and we just had to implement all the functions and debug them.

Our group successfully debugged a lot of issues that popped up, and our program seemed to compile perfectly fine. Issues began coming up when we tried testing out the driver program commands. Most of the commands failed to work and our group had a hard time figuring out what was wrong.

Functions

Milestone 1 (fsinit.c):

1. **void allocateBlock(long index)**
   allocateBlock() is a function that takes in an index to set a particular bit on the bitmap to 0. This indicates that the block, corresponding to the index of that bit, has already been allocated. This function allows us to keep our bitmap updated on which blocks on the disk have already been written to and can't be used for something else.

2. **void freeBlock(long index)**
   freeBlock() is similar to the allocateBlock() function in that it takes in an index to set a bit in our bitmap. However, it instead sets the bit to 1, indicating that the corresponding block on the disk is free to be used and allocated. This is useful for when we move or delete directories and files.

3. **long findFreeBlock(long numberOfBlocks)**
   findFreeBlock() is a function that takes in a parameter that indicates the number of free, continuous blocks that are needed. When this function finds a group of contiguous blocks that are free to use, it will return a long integer that would indicate the starting block of this group of blocks. This function, along with allocateBlock() and freeBlock(), is used throughout the file system to keep our bitmap updated about which blocks on our disk are free to use and which have already been allocated to other things.

4. **long initFreeSpaceMap(long numberOfBlocks, int blockSize)**
   initFreeSpaceMap() takes in two parameters, one long integer to indicate the number of blocks the free space bitmap will represent, and another is an integer to represent the size of a single block. This function is what allocates the appropriate amount of space for our free space bitmap.

5. **DEntry * createDir(DEntry * destDir, char * newDirName)**
   createDir() is a function that performs the allocation and initialization of a directory entry. To do so, the function is passed a pointer to the parent directory and a name for the new directory entry. First, this function will allocate the appropriate amount of space for the directory entry. createDir() also initializes the directory entry to include the directory entry of itself, indicated by a dot(.), and its parent, indicated by two dots(..). It would also perform the function of setting the parent's directory entry to include the information of this newly created entry.

6. **long initRootDir(int blockSize)**
   initRootDir() is a function that initializes the root directory of the file system. This function is passed an integer value that indicates the size of a single block on the disk, in bytes. The purpose of this function is to initialize the root directory, and it

will be called at the start of the program to do so. This function is valuable as it is the one to create the root directory, which will act as the parent directory for all subsequent directory entries.

7. **int initFileSystem (uint64_t numberOfBlocks, uint64_t blockSize)**
   initFileSystem() is one of, if not the, most important functions for this file system. This function gets passed an 8-byte integer value to indicate the number of blocks our file system will have. It also gets passed an 8-byte integer value to indicate the size of each block. This function performs the task of initializing the volume control block and populating the structures' fields with their appropriate values. That volume control block structure is what represents and holds all the information that is needed about the file system.

Milestone 2 (msf.c):

8. **int fs_setcwd(char *buf)**
   fs_setcwd() takes a path parameter and then sets a parsePathStruct pointer to the value returned by parsePath(). fs_setcwd() checks the structure member isValidPath in parsePathStruct to make sure that the value is 1, indicating that the path is valid. We use our helper function, findIndexOfToken() which takes in the parent directory, the last token we want to look for that we store in the parsePathStruct structure and the max number of entries in the directory. fs_setcwd() then checks if the global cwdPtr is being used and frees it if it is. We then set the cwdPtr to the element in the parent directory where the token index indicates. The function also sets the other global variable, cwdName to the path name using our concatPath function.

9. **char * fs_getcwd(char *buf, size_t size)**
   fs_getcwd() takes the global variable, cwdName, and performs strncpy on it with buf and size from the function parameters. This function returns the buffer after being called.

10. **int fs_isFile(char * path)**
    fs_isFile() takes the path provided by the caller and puts it in parsePath() to get back a temporary parsePathStruct. The function then checks if the member isValidPath is set to 0 to represent invalid. If so, it will free the structure and return an error. Else, it will then proceed to check if the final path name is a file or not by utilizing the isDirectory member. fs_isFile() returns a 1 if it is a file, a 0 if it isn’t, and a -1 if the path is invalid.

11. **int fs_isDir(char * path)**
    fs_isDir() takes the path provided by the caller and puts it in parsePath() to get back a temporary parsePathStruct. The function then checks if the member isValidPath is set to 0 to represent invalid. If so, it will free the structure and return an error. Else, it will then proceed to check if the final path name is a directory or not by utilizing the isDirectory member. fs_isDir() returns a 1 if it is a directory, a 0 if it isn’t, and a -1 if the path is invalid.

12. **int fs_mkdir(const char *pathname, mode_t mode)**
    fs_mkdir() takes a path that we will pass into parsePath(). fs_mkdir() also takes in a mode, but for this assignment, we do not have to implement permissions for the directories, so we will not be utilizing whatever is passed into mode. If the path is valid after passing the pathname into parsePath(), we return -1 because we can’t create a directory that already exists. If it is not valid, we store a lastValidLevel member in the structure that is returned from parsePath(). We then use that lastValidLevel and call loadDir() to store that directory into a DEntry pointer. The function then creates a new directory in the loaded directory and then loads the newly made directory for the next token. The function returns 0 upon success.

13. **fdDir * fs_opendir(const char *name)**
    fs_opendir() mallocs a fdDir struct and puts the name parameter into parsePath(). fs_opendir() checks if the path is valid and also checks if the last path is a directory. If any is false, we set the fdDir struct member, isValid, to 0 and return that structure. If they are both true, then we can load the parent directory. fs_opendir() then, call findIndexOfToken() to store that directory into a DEntry struct. The function then populates the members into the malloc'd fdDir structure, which we then return to the caller.

14. **struct fs_diriteminfo *fs_readdir(fdDir *dirp)**
    fs_diriteminfo() takes a fdDir pointer and checks if the path is valid and also a directory. It then loads the directory into a DEntry pointer and starts a for loop that iterates through the record length starting from the directory entry's starting position. If fileName is not empty, then we can perform a strcpy on it with the directory item info name member. We then populate the other members of dirp and increment the directory entry position before returning the fs_diriteminfo struct stored in dirp.

15. **int fs_closedir(fdDir *dirp)**
    fs_closedir() frees and closes all the memory associated with dirp. We check to make sure that dirp is not NULL before we proceed to free the pointer, or else we return -1.

16. **int fs_stat(const char *path, struct fs_stat *buf)**
    fs_stat() takes a path and calls parsePath() with it to get the parsePathStruct. We check for path validity and then load the parent directory into DEntry. We find the index of the token of our last path, with the help of findIndexOfToken(), and then populate the members of the fs_stat struct, but, with the token index inside the parent directory. We return 0 upon completion.

17. **int fs_delete(char* filename)**
    fs_delete() is our file deletion function. fs_delete() calls parsePath() with the filename and then checks to make sure the path is not invalid and is also a file and not a directory. It then loads the parent directory and finds the index of the last token. It calls the helper function deleteEntry(), which accepts a parent directory and a token index and frees the blocks before deleting the entry.

18. **int fs_rmdir(const char *pathname)**
    fs_rmdir() calls parsePath() with the pathname parameter and returns an error if the path is invalid or is a file. If the path is valid and a directory, we load the parent and find the last directory indicated in the pathname. The function then iterates through the final directory, skipping the first two elements (for root and parent), and checks if the directory is empty. If the directory is empty, we call our deleteEntry() helper function to delete the directory.

19. **parsePathStruct *parsePath(char * path)**
parsePath() is a function that takes in a path as a parameter and returns a structure containing information that various other functions will use. This function does so by first tokenizing the given path into each element level. For example, if given the path of "/foo/bar/this/that", it will result in an array containing the values foo, bar, this, and that. This function then determines whether to start from the current working directory or the root directory, based on the given path. It would then proceed to attempt to consecutively load each directory to reach the end. Based on the results of this attempt, parsePath() will populate the parsePathStruct with various information regarding the status of the given path. Some such information includes a pointer to the parent directory, the type of directory or file, and the last element level of the path.

Milestone 3 (b_io.c):

20. **b_open(char * filename, int flags)**
    b_open() returns a file descriptor, which is the main way we track the file we are using for our other functions such as write, read, seek, and close. Our b_open() is passed both the name of a file and a set of flags. We pass the said file name to our parsepath(), which in the case of a file, checks if it exists or not. If the file exists, we check the set of flags that are passed in. Our system can modify and set the flags depending on the combination of flags. If the create flag is set, and the file is not already created, then we will want to call our setFileEntry(), which essentially creates the file. One thing we had to take into consideration is that we could not have the read-only flag set while the truncate or create files are set since we are not allowed to modify a file that is set to read-only.

21. **b_close(b_io_fd_ fd)**
    b_close() closes the file and returns the fcb back to a free state. We first want to use our file descriptor to obtain which file we are going to close. However, before we close our file, we must first ensure that our buffer does not have any unread bytes in it. We check if there are unread bytes by setting a flag to 1 every time we call memmove in b_write(). If the buffer is dirty, we must save the remaining bytes to disk and update our parent directory. In either instance, whether or not our buffer is dirty, we then want to free the buffer that we malloced for our b_fcb structure. After calling free, we then set it to NULL so it can be used again for another file.

22. **b_write (b_io_fd fd, char * buffer, int count)**
    b_write() on success, returns the number of bytes that were written to the disk. b_write() first determines whether the number of blocks to be written to the file is greater than the amount of available space. If there is not enough space, we will allocate a new block within our bitmap which can hold the required amount of file data. Once we have the required amount of space, we can then write our information to our buffer. Subsequently, after the information in our buffer is filled all the way, we can LBAwrite() to disk.

23. **b_read (b_io_fd fd, char * buffer, int count)**
    b_read similarly to the other functions, requires a file descriptor, a buffer, and the requested number of bytes to be read. After checking that we were passed in an acceptable number of bytes to be read, we LBAread a block to our file buffer. Even if the number of bytes to be read is acceptable, we must consider edge cases in which the number of bytes requested exceeds a block. If the amount requested is larger than a block, then we will want to return the remainder of the file and then break out of our while loop. We will return bytes read. Otherwise, we must check if the requested bytes are larger than the remaining space in our buffer. If it is, then we memmove however many bytes we can fit in our buffer, and then we return the bytes read.

24. **b_seek (b_io_fd fd, off_t offset, int whence)**
b_seek() seeks through our file and changes the position of our file. We have three cases for our whence parameter. In case 0, where we want to change the position of our file to the beginning of our file, in case 1, we want to set our position to the current position of our file pointer. Finally, in case 2, we want to set the position of our pointer to the end of the file.

25. **int setFileEntry(DEntry *parentDir, char *fileName)**
setFileEntry() sets an empty directory entry within the parent directory that serves as a file entry and returns the index of the set entry. We first want to loop through our parent directory and find an empty entry location where we can create our file. Once we find the empty entry location, we then want to set the members of our DEntry struct to the proper values that will designate the item in the directory entry as a file. We will then set values such as the location where the file will start and the time it was created and accessed. Once our entry is set, we will want to save the file to disk by calling LBAwrite(). Immediately after saving the file, we will mark the file's block as being allocated to prevent any other files from being written into that block, and we will return the index of our newly created file.

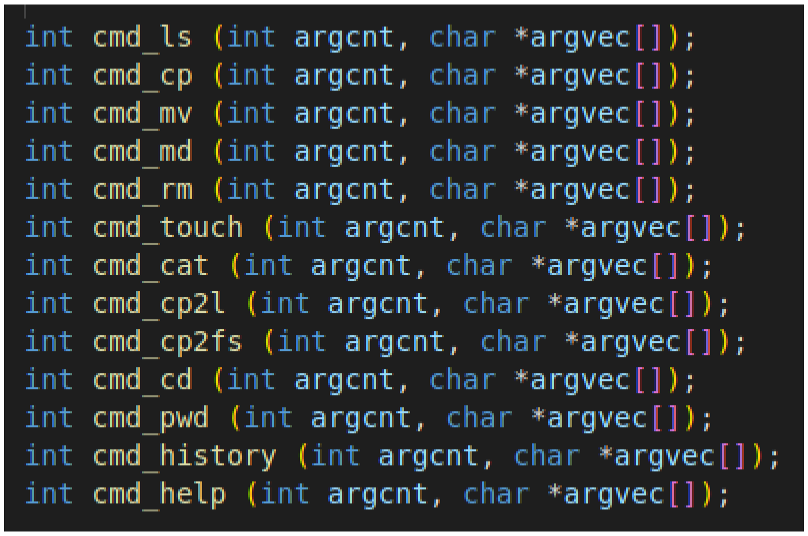
26. **flagStruct *giveFlags (int flags)**
giveFlags() is where we determine the flags that will be set for each file. We will have five different types of flags that can be set; a flag for write only, read-only, read-write, create and truncate. We will use bit operations to shift which bits are set for certain flags. After setting our flags, we will then return a structure that contains the flags that are set. This structure will be used by our b_open when determining whether or not we should do certain operations, such as the creation of a file, truncation of a file, and whether or not we can read or write to a file.

27. **void changeLocation(char *path, long newLocBlock, int newSize)**
changeLocation() takes in a file in which we change the location of the file within our parent directory entry. We must first call the parsed path to obtain the structure that contains the location of the parent directory entry. After we obtain this structure, we use a for loop to go through the parent directory entry to find the file, which is the last token. Once the file is found, we set the location of the file to the new location passed into our function and set the file size to the new file size, also passed into our function. This function is called b_write, and if we need more space for our file, we can move the file into a new block location where there is enough space to satisfy our requirements.

Driver Program

Compile & Run Instructions:
For cleanup purposes, before running the program, delete the sample volume file in the file system folder. A *make clean*, followed by a *make* will make sure that a new volume is created and formatted by our program specifications. *Make run* will initialize our file system and proceed to prompt the user for commands.

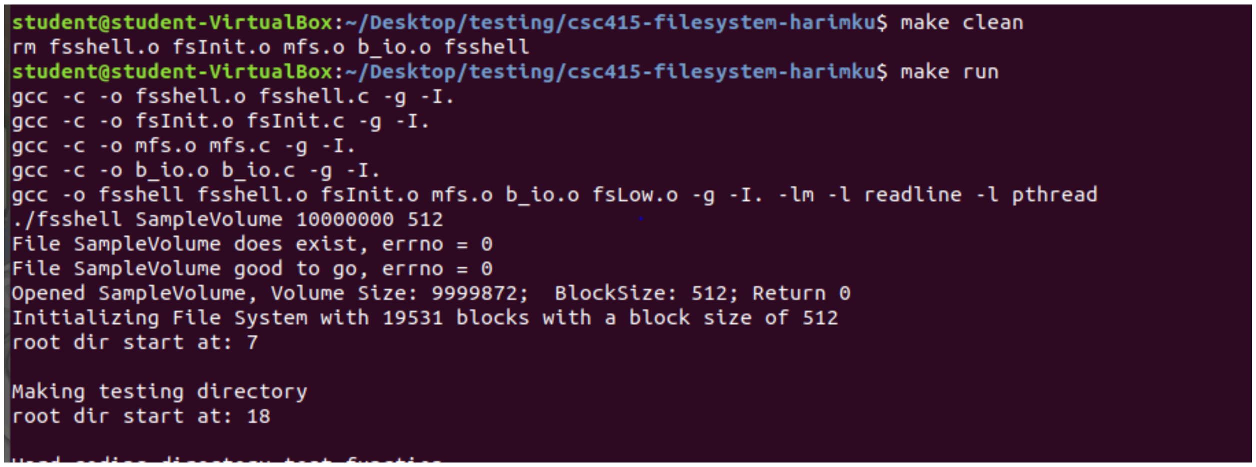
Driver Program:
The driver program itself utilizes our functions and allows the user to input commands when prompted to navigate and use the file system.



```
int cmd_ls (int argcnt, char *argvec[]);
int cmd_cp (int argcnt, char *argvec[]);
int cmd_mv (int argcnt, char *argvec[]);
int cmd_md (int argcnt, char *argvec[]);
int cmd_rm (int argcnt, char *argvec[]);
int cmd_touch (int argcnt, char *argvec[]);
int cmd_cat (int argcnt, char *argvec[]);
int cmd_cp2l (int argcnt, char *argvec[]);
int cmd_cp2fs (int argcnt, char *argvec[]);
int cmd_cd (int argcnt, char *argvec[]);
int cmd_pwd (int argcnt, char *argvec[]);
int cmd_history (int argcnt, char *argvec[]);
int cmd_help (int argcnt, char *argvec[]);
```

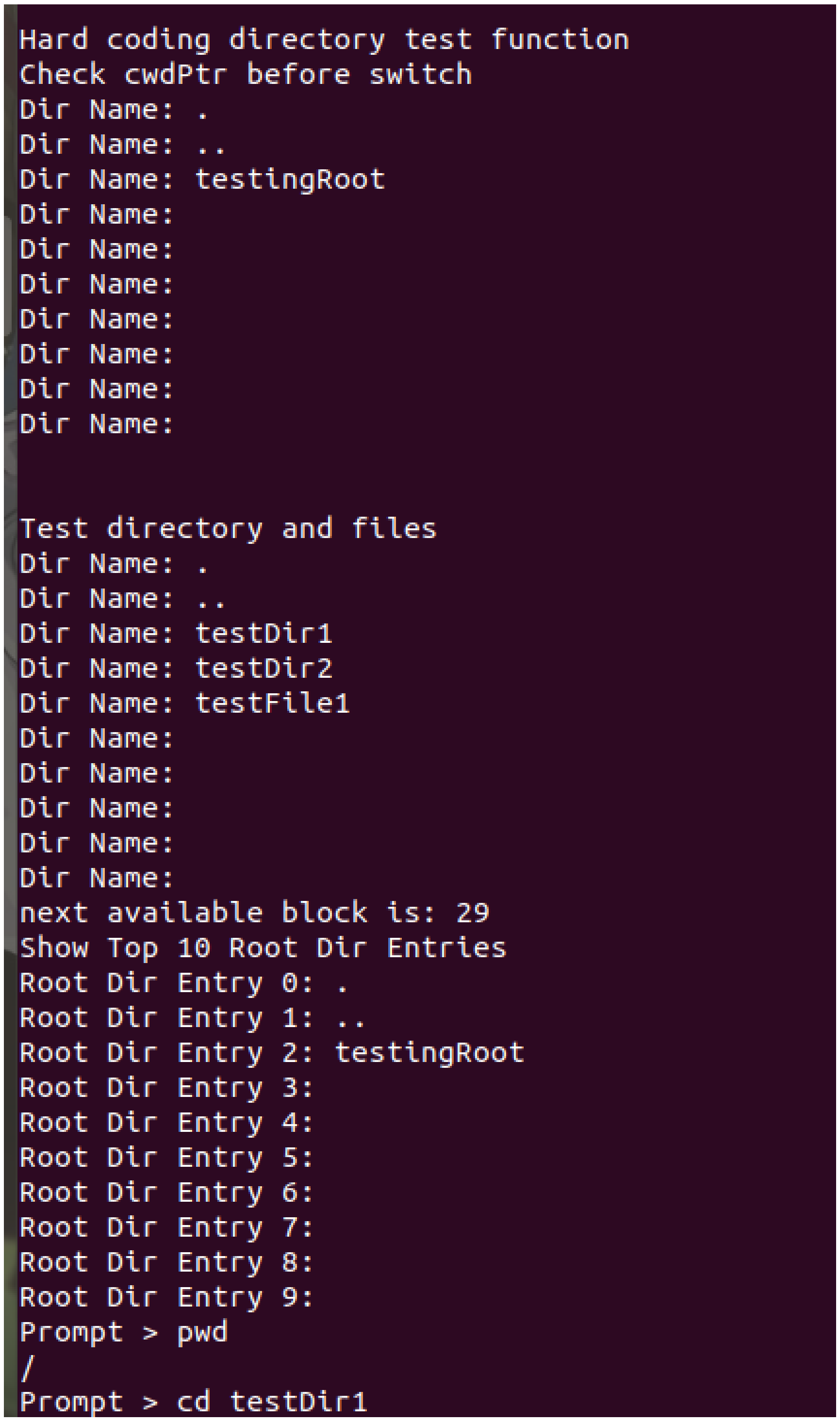
Display of Working Commands

Compiling:



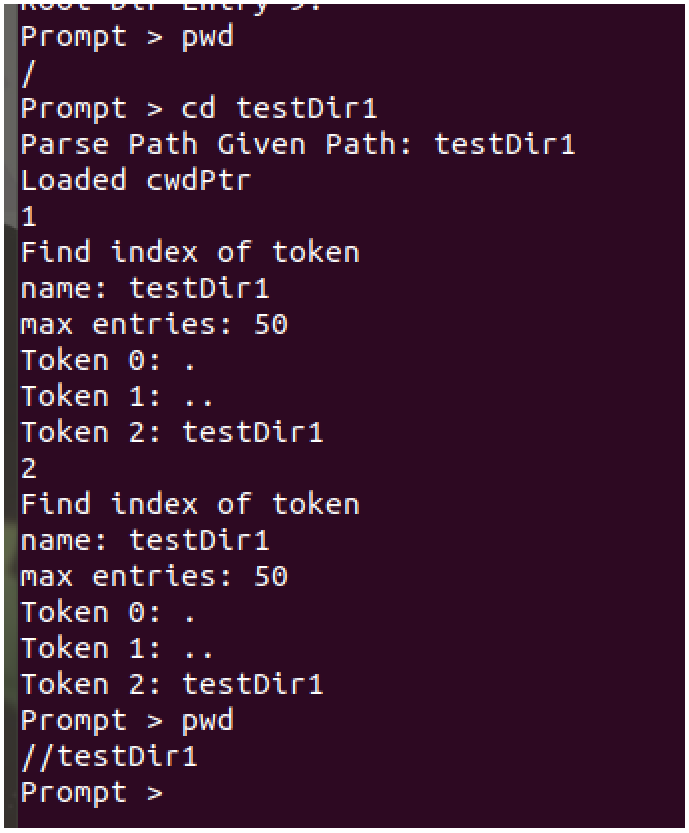
```
student@student-VirtualBox:~/Desktop/testing/csc415-filesystem-harimku$ make clean
rm fsshell.o fsInit.o mfs.o b_io.o fsshell
student@student-VirtualBox:~/Desktop/testing/csc415-filesystem-harimku$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o b_io.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
root dir start at: 7

Making testing directory
root dir start at: 18
```

Directory Creation:



```
Hard coding directory test function
Check cwdPtr before switch
Dir Name: .
Dir Name: ..
Dir Name: testingRoot
Dir Name:
Dir Name:
Dir Name:
Dir Name:
Dir Name:
Dir Name:
Dir Name:


Test directory and files
Dir Name: .
Dir Name: ..
Dir Name: testDir1
Dir Name: testDir2
Dir Name: testFile1
Dir Name:
Dir Name:
Dir Name:
Dir Name:
Dir Name:
next available block is: 29
Show Top 10 Root Dir Entries
Root Dir Entry 0: .
Root Dir Entry 1: ..
Root Dir Entry 2: testingRoot
Root Dir Entry 3:
Root Dir Entry 4:
Root Dir Entry 5:
Root Dir Entry 6:
Root Dir Entry 7:
Root Dir Entry 8:
Root Dir Entry 9:
Prompt > pwd
/
Prompt > cd testDir1
```

<u>Usage of pwd and cd:</u>



```
Prompt > pwd
/
Prompt > cd testDir1
Parse Path Given Path: testDir1
Loaded cwdPtr
1
Find index of token
name: testDir1
max entries: 50
Token 0: .
Token 1: ..
Token 2: testDir1
2
Find index of token
name: testDir1
max entries: 50
Token 0: .
Token 1: ..
Token 2: testDir1
Prompt > pwd
//testDir1
Prompt >
```