# Basic File System

**CSC415-01 Operating Systems**

**PLID:**

**Pearl Anyanwu (916832727)**

**Leo Saeteurn (920928746)**

**Iza Limcolioc (922004678)**

**Denean Le (921330745)**

**Github link:** https://github.com/CSC415-2022-Fall/csc415-filesystem-ledenean.git

**Description**

The program is a simple file system that emulates the file system of a Linux operating system. The purpose of the program is the ability to manage the data storage within the Linux operating system on the disk. It allows a user to be able to manage files and directories with the ability to open and close files, move, remove, and create files. The purpose of the group project is to implement several functions that are major components of a file system that manages the storage. Moreover, the file system is also written and read on the disk.

The file system also contains information about these files and directories such as the names, the file size, the date they were created, the file type, and the location. The file system consists of various components such as structures of Volume Control Block and directory entry system, free space management, as well as various functions such as fs_setcwd, fs_getcwd, fs_isFile, fs_isDir, fs_mkdir, fs_opendir, fs_readdir, fs_closedir, fs_stat, fs_delete, fs_rmdir. With all these components and functions implemented, the given fsshell.c file tests the functions using the Linux commands such as ls, cp, mv, md, rm, touch, cat, cp2l, cp2fs, cd, pwd, history, and help.

**Approach / What we did**

We started off the project by following the pre-work of designing the file system then following the milestones, which is broken down into three phases. Starting with the design of our file system, we created the Volume Control Block (VCB) structure. We grouped together to try to understand what the VCB is and what it consists of. From referencing the *Operating System Concepts*, by Abraham Silberschatx, et al., the VCB is a structure that contains information about the volume such as the size of blocks, number of free blocks, size of blocks, and free block pointers. So our group came up with the elements that are needed within the structure of our file system, which are the number of blocks, the number of free blocks, the size of the block, the root directory, and a signature. After listing the elements within the VCB structure, we listed the elements needed in a directory entry and created a structure for it. We thought about our experiences when looking into a directory or file and came up with the elements of name of the file, the size, the location, and the date created. These were essentially the metadata that we wanted to have within our file system. Moreover, we started to come up with the logic of tracking the free space on the disk. We decided to use bitmap because it allowed us to manipulate the bits, 1 being occupied and 0 being free. This helped us keep track of where the next free block is by iterating the bitmap sequentially.

Moving onto the milestones. The first milestone required us to format our volume, and what this means is to basically implement and initialize the VCB, the Free Space Management, and the Root Directory of our file system. We followed the instructions that were given to us to implement the components properly. The fsInit.c, fsInit.h, bitmap.c, and bitmap.h files contain the initialization of the VCB, the Free Space Management, and the Root Directory. In milestone two, we implemented the functions that help maintain the files and VCB, which are fs_setcwd, fs_getcwd, fs_isFile, fs_isDir, fs_mkdir, fs_opendir, fs_readdir, fs_closedir, fs_stat, fs_delete,

fs_rmdir. These are implemented in the mfs.c and mfs.h files. In the last milestone, we implemented the routine functions which are b_open, b_read, b_write, b_seek, and b_close. The explanations to these functions will be broken down below.

fsInit.c and fsInit.h files

The fsInit.h file contains our VCB structure and a couple prototype functions that are implemented in the fsInit.c file, and a couple defined values; MAGIC_NUMBER and VCB_SIZE. The initialization of the file system starts in the fsInit.c file. This file is where we initialize the free space management and root directory and when called into memory, the system doesn't have to reinitialize the system again. The first function, initFileSystem, returns a 0 upon successfully initializing the system. We first declared a buffer for the VCB and allocated memory. We then read the VCB into the desk starting at block 0. We then check if the signature matches our defined value of the magic number, and if it doesn't match, we initialize the number of blocks, the size of the blocks, the signature - this helps keep the file system in memory, the free space value to record the next starting block, and the root directory. We then write this information back out to the disk.

The next function we created was the initFreeSpaceSystem in which we used bitmap to track our free space. The purpose of this function was to initialize the free space management and for it to return the next free block. We referenced the formula from class lecture to calculate the amount of bytes needed that is used to calculate the amount of blocks needed. The formula we used was (n + (m - 1)) / m. To calculate the bytes needed, we used the value of the number of blocks that is passed through in the first parameter of the function as *n*. We then defined a value for bit size, 8 bits is 1 byte, as *m*. Plugging in the numbers, it gave us the value for bytes needed. Using the same formula, we then used the value of bytes needed as *n*. For the variable *m*, we used the second parameter that is passed through in the function, which is block size; block size is 512 bytes. Plugging in those variables, we get the value for the number of blocks needed. These calculations give us the amount of free space needed on the disk in order to store our file system components. Next, we gave our bitmap memory by using malloc. The size of the memory was calculated using the value of the number of blocks needed multiplied by the block size, which gives us a value in bytes. This calculation gave us enough space to manage the free space based on the value of the number of blocks being passed into the function. The next thing we did was create four helper functions to manipulate, allocate, and check the bits of the bitmap - these are explained below in the bitmap.c and bitmap.h files section. After calculating the values of bytes and blocks needed to find the size of the bitmap, we then initialized all of the free space blocks to be free so that when a file or directory is being created, they can use up the blocks. We created a for loop and called the clearBit helper function to set the blocks to the value 0, meaning free blocks. Next, we allocated the first six bits on the bitmap as used using a for loop and calling the helper function setBit for the VCB starting from block 0. We then wrote this initialization back on the disk by using the given function LBAwrite, which takes in the bitmap, the number of blocks needed to write, and writes it in block 1. The last thing we did in this function was create a for loop and call the checkBit helper function which returns the location of the next free block; this is the location value in which would return back to the freeSpace element in the VCB structure.

The last function in the fsInit.c file is the initRoot function. This function initializes the root directory. Before we are able to initialize the root directory, we need to calculate the amount of memory needed and use that to allocate memory to the directory entries. Similar to the variables used in the initFreeSpaceSystem, we calculated the number of bytes needed by defining the number of entries, which in our system is 50 entries, and multiplying that to the byte size of the directory entry structure. We then used that value to allocate memory to a pointer of the directory entry structure. Next, we used the (n + (m-1)) / m formula and the value that we calculated for the number of bytes needed as *n*, and the block size that's passed in through the function parameter, 512 bytes, as *m*, to calculate the number of blocks needed. Next, we initialized each element of the directory entry structure as known free state by using a for loop, looping through the 50 entries. We then allocated the free block spaces by using the helper function allocFreeSpace. Then we moved on to setting the indexes 0 and 1 as the root directories; '.' and '..', as well as all the elements within the directory entry structure for the root directory. The . and .. root directories are both the same but just have different names. After initializing the root directories, we use the LBAwrite function to write the root directories onto the disk. The last function in this file is just the exitFileSystem function that exits the system when called.

bitmap.c and bitmap.h files

We decided to make a separate file that holds all functions concerning our bitmap so we can gain access to it from other files. In this file, we implemented functions that allow us to set, clear, and check the bits in the bitmap. The function setBit sets the bits to zero which signifies that those bits are used, meanwhile, the function clearBit sets the bits to one which signifies that those bits are free. We also created a function called checkBit that allows us to check what value the bits are in the bitmap. These functions are used in allocFreeSpace and freeBlocksInFreeSpace. To allocate blocks in the free space system, we had to start by finding the first free bit in the bitmap to figure out where our starting block is. This is done by looping through the bitmap and using checkBit to check each individual bit. The loop ends when we find the first free bit, then we start another loop that checks if the bits after the starting position are free and if there are enough bits available according to the number of blocks we need to allocate. If there are enough free blocks available, we then set the bits as used by using setBit and returning the starting point of the free bits. Unlike allocFreeSpace, freeBlocksInFreeSpace is a function that frees the blocks in the free space system. This is implemented by using clearBit in a loop that starts from the starting block of the given directory and ends when we reach the size of the directory. To make sure it worked, we looped through it again and used checkBit to check if the bits were set to free.

mfs.c and mfs.h files

First, we created the parsePath function to split up the given pathname and return the last known element in the directory. We did this by checking if the path is absolute or relative, where if the path is relative, we would load the current working directory, otherwise, tokenize the path and load each token. We created a helper function to load the directories, called loadDir, which would allocate space to read the directory from disk and return directory read from disk.

By doing so, we are able to gain access to the directory and check if the directory exists and whether or not it is a directory or a file. When parsePath reaches the last element in the pathname, it returns a pointer to that directory.

After creating parsePath, we are able to implement the fs_mkdir function, where the purpose of the function is to create a new directory. First, we use parsePath to check if the given pathname exists. If the path does not exist, then we create a directory by allocating enough blocks in the free space for the new directory, and populating the directory entry structure with the given pathname, location, file type, and date created. Since we made changes to the directory, we also had to write the directory onto the disk with the appropriate blocks needed, and the location.

Unlike fs_mkdir, the function fs_rmdir removes a directory under the condition that the directory is empty before removing and that the given pathname is a directory. To implement this, we used parsePath to check if the directory exists and used fs_isDir to check if it is a directory. To find out if the directory is empty, we calculated the number of entries by dividing the size of the directory to the size of one entry. If the result of the calculation is zero, then the function will remove the directory by freeing the blocks allocated for the directory and marking it as unused. This process is repeated inside fs_delete, but the only difference is instead of checking if the given pathname is a directory, we check if it is a file by using fs_isFile. Since fs_delete only works with files, we do not need to find out if it is empty.

For the directory iteration functions, we had to implement three functions that opened the directory, read the entries in the directory, and closed the directory. For fs_opendir, we used parsePath to get the directory entry of the last element, where we used loadDir to load the directory and set a pointer to it in the fdDir structure. Then, we allocated space of the directory, and set the variables in the fdDir structure to the appropriate values. After doing so, fs_opendir returns the pointer to the structure which is then accessed by fs_readdir and fs_closeDir. In fs_readdir, we loop through the number of directory entries in the given pointer, and check if the entry is used. If the entry is used, then we return the next directory entry. In fs_closeddir, we are simply freeing everything that we allocated space for in fs_opendir.

When implementing fs_getcwd and fs_setcwd, we created a global variable to hold the pathname given in fs_setcwd. Before assigning the pathname to the global variable, we needed to check if it exists by using the parsePath function. If it exists, then the pathname is assigned to the global variable, which is returned by fs_getcwd.

When implementing fs_isFile and fs_isDir, we used parsePath to check if the pathname exists, then used the pointer returned from parsePath to check if the file type of the directory entry is a file or a directory. We then return a 1 if it’s the respective file type or a 0 if it’s not.

The function fs_stat passes a pointer to a structure and a pathname, in which we use to populate the pointer with information from our directory entry. Before populating the structure, we had to find where the directory entry was in our array of entries by using our helper function

called findEntry. This helper function loops through the array of directory entries and finds the entry that matches the given pathname, then returns the index of where it is in the array.

b_io.c and b_io.h files

Starting with the b_io routine functions, we have implemented b_open, b_read, b_write, b_seek, and b_close to emulate file I/O functions in linux and how they work in our file system.

For b_open, we made a getFileInfo function in which uses parsePath to get the directory entry and use a given pointer to populate our file information. We have also provided a check to see if the file isn't found when opening the buffered file. Afterwards, we malloc'd our buffer using B_CHUNK_SIZE and if we're not able to malloc that would indicate our open function failing. We then got our own file descriptor using b_getFCB and checked if all FCB's are used. We also set the pointers and variables from our struct to its initial state (meaning not reading anything yet). For numBlocks especially, we set it to the block size of the file added to the new chunk of bytes and then we divided by the original B_CHUNK_SIZE. At the end of the function, we test to see if it works with a printf statement and we return using the returnFd variable.

For b_read, we split our logic into three parts in order to fill the callers request: part 1 being what's filled from the existing buffer, part 2 as the filled number of bytes in multiples of block size, and part 3 as the leftover bytes filled from the refilled buffer. We also check if the specified file control block is in use, and if this fails it would indicate the file not opening. We created a variable remainBytesInBuffer to hold the number of bytes that is available to copy from our buffer, and set it to the valid bytes in our buffer minus our current position in buffer. Also, we made another variable amountDelivered to limit the count of the file length.

Additionally, we also check if the count is negative, and provide a error that it cannot exceed end of file. We continue to check if we have enough in our buffer to give, and if we do satisfy the caller's request, we set part 1 to the requested count as it's completely buffered. We also set part 2 and part 3 to 0 as the "next" bytes aren't needed. If not satisfied, we have to give more than what is in the buffer so we have to set part 1 to whatever's left in our buffer, part 3 to how many more bytes we need to give, and we find how many blocks there are and how many whole blocks worth of bytes to give by using numBlocksCopied and B_CHUNK_SIZE. Then we give the remainder which is how many bytes after given the whole blocks. Lastly, we made a set of checks for each part if it goes past zero.

For part 1's condition, we used memcpy to copy those to our buffer and incremented the index by that much amount. For part 2's condition, we used the blocks that are copied directly to the caller's buffer with LBAread and limited the blocks to the number of blocks left. We also set part 2 to the number of bytes read in this condition. For part 3's condition, we refill our buffer to copy more blocks and with LBAread we read B_CHUNK_SIZE bytes into our buffer. Then, we reset the offset and how many bytes are actually read, and check if there's not enough bytes left to satisfy the request. We implemented a check to also memcpy bytesRead and adjust the index for the copied bytes. After these part conditions, we would add all parts and set it to the variable bytesReturn and return this variable at the very end.

For b_write, the purpose of the function is to write contents into a file, which can be used for the copy file and move file Linux commands. The function returns an integer with three parameters; b_io_fd fd, char * buffer, and int count. We first error handled the count to make sure it is not zero, if it is, then we return with an error. Next we allocated memory for the buffer. Then we declared some variables: int blocksWritten, int freeBlocks, int numOfBlocks, and int startBlocks. We initialized the freeBlocks to 0 and numOfBlocks to 50. We then called the function, allocFreeSpace, passing the numOfBlocks into the parameter, and the function returns the next free space, which we initialized to startBlock. Then we check if the startBlock value is 0, if it is, we check how many free blocks there are by calling the checkBit function to check the bits. We increment freeBlocks which we used as a counter, then we set the numOfBlocks to the incremented value of freeBlocks. Next, we created a while loop to dynamically increment the numOfBlocks by doubling it. We used the function LBAwrite to write the buffer onto the disk with the count size, starting at the next free block and initializing that return value to blocksWritten. Then the last thing we did was free up the space not needed, a couple edge cases. The first is if the count is less than the numOfBlocks, if it is, we would calculate the number of blocks left by subtracting blocksWritten by numOfBlocks along with looping through the bitmap and setting them free using the function clearBit. The next edge case is if the count is greater than the numOfBlocks, we would write the first block of the count written to the disk and initialize the return value while incrementing it to blocksWritten. And finally, we return the number of blocks written.

For b_seek, we came up with the logic to use the nth byte by changing the location of the pointer of a fd. As it allows the file offset to be set beyond the end of the file, we created a temporary variable to hold the new position of the pointer called newPos. With this, we used a switch case to cover all 3 cases of the function: SEEK_SET, SEEK_CUR, and SEEK_END. In SEEK_SET, we have the file offset set to offset bytes. In SEEK_CUR, we have the file offset set to the current location added to the offset bytes. In SEEK_END, we have the file offset set to the block size of the file added to the offset bytes. As a default case, we return an invalid statement when it can't be reached.

For b_close, we free all allocated memory in our buffer and set both pointers in our struct back to null. This would place the file control block to the unused FCB pool so we can close the file successfully with the return statement. We have also provided a printf statement to test if the file actually closed with the file descriptor.

**Issues and Resolutions**

The first issue we ran into was during milestone one in the initRoot function within the fsInit.c file. First thing was that the formula we used for calculating the number of blocks needed was incorrect. We initially calculated it by dividing the size of our directory entry by the block size that's passed in from the function parameter. We had received feedback that it was incorrect, so we went back and started using test values to test the formula and found out that the value was returning 0 each time. We then went back to the class lecture videos and realized that the formula that we used in the initFreeSpaceSystem, (n + (m - 1))/m, was the correct

formula that we should've used. So we modified that formula, used some values to test the calculation and ended up getting the value that we wanted.

The next issue was also in the initRoot function, which is using the dot method to access elements within the directory entry function. This was another feedback given to us that was an error that we had to fix. We created a pointer to the directory entry structure and used that pointer to access the elements. What we forgot was when we're using a pointer, we need to use the pointer method to access and not the dot. This was a minor issue, but still an issue that needed to be fixed.

The third issue we had, which might be the toughest issue we had throughout the project, was the parse path helper function. Our initial approach to implementing the parse path was to try to understand what it's supposed to do from the lecture videos but the more we watched the lecture videos and came up with our logic to implement, the more confused we got. Although it seemed simple, it was more difficult to implement it to align with the other functions that called the parse path function. Our understanding of the parse path was it loads the directory entry using the given LBAread function, use the pathname that gets passed into the parameter to find that directory, parse the path using strtok() function and by delimiter, check if the path is an absolute or relative path, then if it's a file or a directory, then return the pointer to the directory entry. So following that logic, after implementing it, we ran into the issue of being confused with what to do for the absolute and relative path - we knew how to check it but were a bit confused on the next step. The next issue within the parse path was checking if the entry is a file or a directory - we tried using the isFile and isDir functions that we had to implement, but at the end, ended up not using it because it just didn't fit, so that was another confusion we had with when and where to use those functions. Another issue was how to load the directory entry - we initially thought that we were able to add the header file from the fsInit.c where the directory entry structure was originally stored and accessing it directly, but we ended up clarifying with Professor Bierman that we needed to read from the disk in order to load it. So we kept debugging, rearranging the codes, changing up the logic a bit, and went back to old lectures from previous assignments for some tips. We haven't seen any errors come from our parse path function anymore, so assuming it worked correctly and we were able to resolve our confusion.

The fourth issue we had was implementing a helper function that we would call in the function that opens a directory, our loadDir. We wanted to create a function where we can load the directory from the disk and return the pointer to the directory entry. One of the difficult parts about doing that was finding a way to actually access the directory. Our original logic was to create a pointer that points to our directory entry structure and thought this was the proper way to access the entries. After testing a couple linux commands, we realized that we weren't accessing the entries at all. After having office hours with Professor Bierman, it was clear that we needed to read from the disk in order to access the entries. So we did this by using the LBAread. The LBAread reads directly from the disk - as it was written on the disk - which allowed us to be able to access the directory entry.

Another issue we experienced was that our program wasn't writing any blocks onto the disk. After investigating our calculations for blocksNeeded and the allocFreeSpace function, we

found that our calculation for blocksNeeded was correct, but the loops inside allocFreeSpace were incorrectly written. Originally, we had a loop that started setting the bits from the startBlock (int j = startBlock) and ending when the number of blocks is reached (j <= numOfBlocks). This was incorrect because the numOfBlocks could be less than the startBlock and the loop wouldn't iterate the same amount of times as the number of blocks. To solve this, we wanted the loop to start at the startBlock, but also end when the loop count is equivalent to the number of blocks, so we added startBlock to the numOfBlocks variable (j <= startBlock + numOfBlocks) to set the correct about of bits in the bitmap.

For the b_io functions, especially b_seek and b_write, an issue we had with those was how exactly we were going to implement it based on our logic. Because we used assignment 5 as a stepping stone for b_open, b_close, and b_read in our file system, it was more difficult to start the other functions from scratch with little understanding. However, after researching the man pages for lseek and going back to lecture recordings for reference, we saw that b_seek is basically making use of a "nth" byte to change the location of a pointer of a file descriptor. As a solution, we realized that just like the lseek file I/O function, our b_seek function should be able to reposition the read/write file offset of the open "fd" with whence as the directive. This would let the file's offset be set beyond the end of the file, and we would have 3 separate cases: SEEK_SET, SEEK_CUR, SEEK_END. With these conditions, we understood that it is necessary to use so we can successfully change the current file offset to a new position in the file.

**Analysis / How our driver works**

Our driver starts in the fsInit.c file where the implementation and initialization of the free space management and the root directory are created. This is the initial starting point of any driver because this is what's booted up for the file system when a computer is booted up. Once the initialization is set, the free space management can stay in memory so when accessing the file storage, we don't have anything deleted when turning off a computer. For our particular program, we use the given fsshell.c file to test out the file system using the data from the file SampleVolume that are attached to the program. So within our program, after initializing the free space management and the root directory, then some of the functions and structures are stored within the mfs.c file. That file contains the functions that are called from the fsshell.c file. This allows the test functions to make a directory, remove a directory, open a directory, read a directory and returning the information, close a directory, get the current working directory, set the current working directory, check if the directory entry is a file or a directory, delete a file, and has structures that contains the variables that store the directory information. Moving onto our b_io.c file, this is where the program allows for a directory to be opened, read, write, seek within the file, and closing the file.

The fsshell.c is an object file that has implemented functions of the Linux commands. This file is the driver of the program where it allow us to test the commands, which includes ls that lists the files within a directory, cp that copies a file from another, mv which renames or moves a file, md which makes a new directory, rm which removes a file or a directory, touch which creates a new file, cat that displays the contents of a file, cp2l that copies a files from a

test file system to the Linux file system, cp2fs which copies a file from a Linus file system to the test file system, cd which changes the directory, pwd which prints the working directory, history that displays the command history, and help which prints out some instructions.

To see the data that the file system stores, there is also a separate folder within the program called Hexdump. It contains a text file called dump.txt where the data can be dumped into. We can run this by going into the Hexdump folder, and running the command ./hexdump SampleVolume > dump.txt. If the command runs correctly, we should be able to open up the dump.txt file to see the data - the written blocks are shown down below. Each line of the data is one block of the bitmap, the data is in hex, and the numbers are paired as one byte.

**References:**


Silberschatz, Abraham, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*, 10th Ed., published by John Wiley & Sons, Inc.

Arora, Himanshu. The Geek Stuff, 2012, Jun 15, https://www.thegeekstuff.com/2012/06/c-directory

Kerrisk, Michael. “Linux Programmer's Manual.” *Lseek(2) - Linux Manual Page*, 22 Mar. 2021, https://man7.org/linux/man-pages/man2/lseek.2.html.

Die.net. “write(2) - Linux man page”. *Open source forum.* https://linux.die.net/man/2/write


Steps for milestone.pdf

**Screenshots of each of the command from the README**

```
student@student-VirtualBox:~/csc415-filesystem-ledenean$ 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 > exit
System exiting
student@student-VirtualBox:~/csc415-filesystem-ledenean$
```

```
student@student-VirtualBox:~/csc415-filesystem-ledenean$ 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 plid
Could not change path to plid
Prompt > exit
System exiting
student@student-VirtualBox:~/csc415-filesystem-ledenean$
```

```
student@student-VirtualBox:~/csc415-filesystem-ledenean$ 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 > ls

Prompt > ls -a

.
..
Makefile:67: recipe for target 'run' failed
make: *** [run] Segmentation fault (core dumped)
student@student-VirtualBox:~/csc415-filesystem-ledenean$
```

```
student@student-VirtualBox:~/csc415-filesystem-ledenean$ 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 plid
Makefile:67: recipe for target 'run' failed
make: *** [run] Segmentation fault (core dumped)
student@student-VirtualBox:~/csc415-filesystem-ledenean$
```

```
student@student-VirtualBox:~/csc415-filesystem-ledenean$ 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 plid
Makefile:67: recipe for target 'run' failed
make: *** [run] Segmentation fault (core dumped)
student@student-VirtualBox:~/csc415-filesystem-ledenean$
```

```
student@student-VirtualBox:~/csc415-filesystem-ledenean$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o bitmap.o fsLow.o -g -I. -lm -l readline -l pthread
fsshell.o: In function `cmd_touch':
/home/student/csc415-filesystem-ledenean/fsshell.c:258: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:262: undefined reference to `b_close'
fsshell.o: In function `cmd_cat':
/home/student/csc415-filesystem-ledenean/fsshell.c:292: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:303: undefined reference to `b_read'
/home/student/csc415-filesystem-ledenean/fsshell.c:307: undefined reference to `b_close'
fsshell.o: In function `cmd_cp':
/home/student/csc415-filesystem-ledenean/fsshell.c:347: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:348: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:351: undefined reference to `b_read'
/home/student/csc415-filesystem-ledenean/fsshell.c:352: undefined reference to `b_write'
/home/student/csc415-filesystem-ledenean/fsshell.c:354: undefined reference to `b_close'
/home/student/csc415-filesystem-ledenean/fsshell.c:355: undefined reference to `b_close'
fsshell.o: In function `cmd_mv':
/home/student/csc415-filesystem-ledenean/fsshell.c:397: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:398: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:401: undefined reference to `b_read'
/home/student/csc415-filesystem-ledenean/fsshell.c:402: undefined reference to `b_write'
/home/student/csc415-filesystem-ledenean/fsshell.c:404: undefined reference to `b_close'
/home/student/csc415-filesystem-ledenean/fsshell.c:405: undefined reference to `b_close'
fsshell.o: In function `cmd_cp2l':
/home/student/csc415-filesystem-ledenean/fsshell.c:492: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:496: undefined reference to `b_read'
/home/student/csc415-filesystem-ledenean/fsshell.c:499: undefined reference to `b_close'
fsshell.o: In function `cmd_cp2fs':
/home/student/csc415-filesystem-ledenean/fsshell.c:536: undefined reference to `b_open'
/home/student/csc415-filesystem-ledenean/fsshell.c:541: undefined reference to `b_write'
/home/student/csc415-filesystem-ledenean/fsshell.c:543: undefined reference to `b_close'
collect2: error: ld returned 1 exit status
Makefile:61: recipe for target 'fsshell' failed
make: *** [fsshell] Error 1
student@student-VirtualBox:~/csc415-filesystem-ledenean$
```

**Hexdump:**

The hexdump provided below are the blocks that are written on. Block 0 was supposed to be the Volume control Block. Block 1 was where we stored our bitmap for the free system management. Blocks 7-11 are where the root of the directory is stored. Each row is a block of the bitmap.

In block 0, the yellow highlight numbers are in bytes that are translated from the characters on the right. So to translate a couple characters, capital C is 43 is in the ASCII table, capital S is 53, and so on. In block 1, the bytes highlighted in cyan is our magic number that is a global variable in the fsInit.h file. For the rest of the data, we are not able to translate them because there are characters not part of the numeric alphabet. But to assume what we were supposed to store, blocks 7-11 should be our root directory, which should contain the structure and the elements stored for the root directory metadata.

Block 0

000000: 43 53 43 2D 34 31 35 20 2D 20 4F 70 65 72 61 74 | CSC-415 - Operat

000010: 69 6E 67 20 53 79 73 74  65 6D 73 20 46 69 6C 65 | ing Systems File

000020: 20 53 79 73 74 65 6D 20  50 61 72 74 69 74 69 6F |  System Partitio

000030: 6E 20 48 65 61 64 65 72  0A 0A 00 00 00 00 00 00 | n Header........

000040: 42 20 74 72 65 62 6F 52  00 96 98 00 00 00 00 00 | B treboR.��.....

000050: 00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | ........KL......

000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000070: 52 6F 62 65 72 74 20 42  55 6E 74 69 74 6C 65 64 | Robert BUntitled

000080: 0A 0A 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0000D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000100: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000110: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000120: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000130: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000140: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000150: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000160: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000170: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000180: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000190: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0001A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0001B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0001C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0001D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0001E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

0001F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

Block 1

000200: 4B 4C 00 00 00 02 00 00  06 00 00 00 06 00 00 00 | KL..............

000210: 44 49 4C 50 00 00 00 00  00 00 00 00 00 00 00 00 | DILP............

000220: 00 2D 42 65 FF 7F 00 00  0D DC 88 47 D5 55 00 00 | .-Be□...·G□U..

000230: 00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | ........KL......

000240: 40 2E 42 65 FF 7F 00 00  70 C6 3A 49 D5 55 00 00 | @.Be□..p□:I□U..

000250: 60 2D 42 65 FF 7F 00 00  78 D9 88 47 D5 55 00 00 | `-Be□..x□G□U..

000260: 48 2E 42 65 FF 7F 00 00  04 00 00 00 04 00 00 00 | H.Be□...........

000270: 01 00 00 00 00 00 00 00  00 96 98 00 00 00 00 00 | .........□□.....

000280: 00 02 00 00 00 00 00 00  7D 43 42 65 FF 7F 00 00 | ........}CBe□..

000290: 40 EC 88 47 D5 55 00 00  90 D2 88 47 D5 55 00 00 | @□G□U..□□G□U..

0002A0: 40 2E 42 65 FF 7F 00 00  00 08 AD 97 D6 A1 13 E1 | @.Be□....□□.□

0002B0: 40 EC 88 47 D5 55 00 00  87 3C 56 C1 9D 7F 00 00 | @□G□U..□<V□□..

0002C0: 00 00 00 00 20 00 00 00  48 2E 42 65 FF 7F 00 00 | .... ...H.Be□..

0002D0: 00 00 00 00 04 00 00 00  7B D8 88 47 D5 55 00 00 | ........{  G□U..

0002E0: 00 00 00 00 00 00 00 00  8C 98 67 D7 8F D0 5D F6 | ........ ..g   ]

0002F0: 90 D2 88 47 D5 55 00 00  40 2E 42 65 FF 7F 00 00 | ...G.U..@.Be..

000300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000310: 8C 98 07 55 1A 95 09 A2  8C 98 99 77 32 DD CC A2 | ...U.....w2...

000320: 00 00 00 00 FF 7F 00 00  00 00 00 00 00 00 00 00 | ................

000330: 00 00 00 00 00 00 00 00  D3 98 14 C2 9D 7F 00 00 | ........ .. ..

000340: 88 7D 13 C2 9D 7F 00 00  B0 CB B4 C1 9D 7F 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  90 D2 88 47 D5 55 00 00 | ........ ...G.U..

000370: 40 2E 42 65 FF 7F 00 00  BA D2 88 47 D5 55 00 00 | @.Be..  ...G.U..

000380: 38 2E 42 65 FF 7F 00 00  1C 00 00 00 00 00 00 00 | 8.Be............

000390: 04 00 00 00 00 00 00 00  73 43 42 65 FF 7F 00 00 | ........ sCBe..

0003A0: 7D 43 42 65 FF 7F 00 00  8A 43 42 65 FF 7F 00 00 | }CBe..  .CBe..

0003B0: 93 43 42 65 FF 7F 00 00  00 00 00 00 00 00 00 00 | .CBe............

0003C0: 97 43 42 65 FF 7F 00 00  AE 43 42 65 FF 7F 00 00 | .CBe..  .CBe..

0003D0: C3 43 42 65 FF 7F 00 00  EC 43 42 65 FF 7F 00 00 | .CBe..  .CBe..

0003E0: FE 43 42 65 FF 7F 00 00  12 44 42 65 FF 7F 00 00 | .CBe...  DBe..

0003F0: 22 44 42 65 FF 7F 00 00  33 44 42 65 FF 7F 00 00 | "DBe..  3DBe..

Block 7

000E00: 40 03 A4 63 79 55 00 00  00 9C 7A B9 71 6D 0F A5 | @..cyU....z.qm..

000E10: 06 00 00 00 8A 09 00 00  00 00 00 00 00 00 00 00 | ................

000E20: D0 39 C0 F1 FC 7F 00 00  DB 7B 23 62 79 55 00 00 | .9.....  .{#byU..

000E30: 00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | ........ KL......

000E40: 10 3B C0 F1 FC 7F 00 00  70 F6 A3 63 79 55 00 00 | .;���..p��cyU..

000E50: 30 3A C0 F1 FC 7F 00 00  78 79 23 62 79 55 00 00 | 0:���..xy#byU..

000E60: 18 3B C0 F1 FC 7F 00 00  04 00 00 00 04 00 00 00 | .;���.....

000E70: 01 00 00 00 00 00 00 00  00 96 98 00 00 00 00 00 | .....��.....

000E80: 00 02 00 00 00 00 00 00  7D 53 C0 F1 FC 7F 00 00 | ........}S���..

000E90: 40 8C 23 62 79 55 00 00  90 72 23 62 79 55 00 00 | @�#byU..�r#byU..

000EA0: 10 3B C0 F1 FC 7F 00 00  00 9C 7A B9 71 6D 0F A5 | .;���...�z�qm.�

000EB0: 40 8C 23 62 79 55 00 00  87 0C 46 70 88 7F 00 00 | @�#byU..�.Fp��..

000EC0: 00 00 00 00 20 00 00 00  18 3B C0 F1 FC 7F 00 00 | .... ....;���..

000ED0: 00 00 00 00 04 00 00 00  7B 78 23 62 79 55 00 00 | ........{x#byU..

000EE0: 00 00 00 00 00 00 00 00  B5 58 3D E6 68 20 0F FE | ........X=h .

000EF0: 90 72 23 62 79 55 00 00  10 3B C0 F1 FC 7F 00 00 | r#byU...;���..

000F00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000F10: B5 58 3D 8A AF 07 04 AB  B5 58 C3 E6 A3 04 ED AB | X=..X.X.

000F20: 00 00 00 00 FC 7F 00 00  00 00 00 00 00 00 00 00 | ....�.........

000F30: 00 00 00 00 00 00 00 00  D3 68 04 71 88 7F 00 00 | ........h.q��..

000F40: 88 4D 03 71 88 7F 00 00  B0 9B A4 70 88 7F 00 00 | M.q��..p��..

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  90 72 23 62 79 55 00 00 | ........r#byU..

000F70: 10 3B C0 F1 FC 7F 00 00  BA 72 23 62 79 55 00 00 | .;���..r#byU..

000F80: 08 3B C0 F1 FC 7F 00 00  1C 00 00 00 00 00 00 00 | .;���.........

000F90: 04 00 00 00 00 00 00 00  73 53 C0 F1 FC 7F 00 00 | ........sS���..

000FA0: 7D 53 C0 F1 FC 7F 00 00  8A 53 C0 F1 FC 7F 00 00 | }S���..S���..

000FB0: 93 53 C0 F1 FC 7F 00 00  00 00 00 00 00 00 00 00 | ﷺ.....

000FC0: 97 53 C0 F1 FC 7F 00 00  AE 53 C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

000FD0: C3 53 C0 F1 FC 7F 00 00  EC 53 C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

000FE0: FE 53 C0 F1 FC 7F 00 00  12 54 C0 F1 FC 7F 00 00 | ﷺ...T...

000FF0: 22 54 C0 F1 FC 7F 00 00  33 54 C0 F1 FC 7F 00 00 | "T...3T...

Block 8

001000: 43 54 C0 F1 FC 7F 00 00  70 54 C0 F1 FC 7F 00 00 | CT...pT...

001010: 82 54 C0 F1 FC 7F 00 00  93 54 C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

001020: E8 54 C0 F1 FC 7F 00 00  FE 54 C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

001030: 1E 55 C0 F1 FC 7F 00 00  86 55 C0 F1 FC 7F 00 00 | .U...U...

001040: BA 55 C0 F1 FC 7F 00 00  24 56 C0 F1 FC 7F 00 00 | ﷺ..$V...

001050: 43 56 C0 F1 FC 7F 00 00  58 56 C0 F1 FC 7F 00 00 | CV...XV...

001060: 6F 56 C0 F1 FC 7F 00 00  5B 5C C0 F1 FC 7F 00 00 | oV...[\...

001070: 76 5C C0 F1 FC 7F 00 00  9F 5C C0 F1 FC 7F 00 00 | v\...\...

001080: AA 5C C0 F1 FC 7F 00 00  BD 5C C0 F1 FC 7F 00 00 | \...\...

001090: D0 5C C0 F1 FC 7F 00 00  FD 5C C0 F1 FC 7F 00 00 | \...\...

0010A0: 0C 5D C0 F1 FC 7F 00 00  1F 5D C0 F1 FC 7F 00 00 | .]...]...

0010B0: 3B 5D C0 F1 FC 7F 00 00  71 5D C0 F1 FC 7F 00 00 | ;]...q]...

0010C0: 93 5D C0 F1 FC 7F 00 00  A3 5D C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

0010D0: CF 5D C0 F1 FC 7F 00 00  DA 5D C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

0010E0: EE 5D C0 F1 FC 7F 00 00  FF 5D C0 F1 FC 7F 00 00 | ﷺ..ﷺ..

0010F0: 07 5E C0 F1 FC 7F 00 00  27 5E C0 F1 FC 7F 00 00 | .^...'^...

001100: 38 5E C0 F1 FC 7F 00 00  45 5E C0 F1 FC 7F 00 00 | 8^...E^...

001110: 66 5E C0 F1 FC 7F 00 00  7B 5E C0 F1 FC 7F 00 00 | f^���..{^���..

001120: 86 5E C0 F1 FC 7F 00 00  8E 5E C0 F1 FC 7F 00 00 | ^^���..^^���..

001130: A6 5E C0 F1 FC 7F 00 00  B9 5E C0 F1 FC 7F 00 00 | ^^���..^^���..

001140: DA 5E C0 F1 FC 7F 00 00  EC 5E C0 F1 FC 7F 00 00 | ^^���..^^���..

001150: 0D 5F C0 F1 FC 7F 00 00  63 5F C0 F1 FC 7F 00 00 | ._���..c_���..

001160: 80 5F C0 F1 FC 7F 00 00  8D 5F C0 F1 FC 7F 00 00 | _���..._���..

001170: A5 5F C0 F1 FC 7F 00 00  BD 5F C0 F1 FC 7F 00 00 | _���..._���..

001180: CE 5F C0 F1 FC 7F 00 00  E2 5F C0 F1 FC 7F 00 00 | _���..._���..

001190: 00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00 | ........!.......

0011A0: 00 F0 CF F1 FC 7F 00 00  10 00 00 00 00 00 00 00 | .����.........

0011B0: FF FB 8B 17 00 00 00 00  06 00 00 00 00 00 00 00 | ���.............

0011C0: 00 10 00 00 00 00 00 00  11 00 00 00 00 00 00 00 | ................

0011D0: 64 00 00 00 00 00 00 00  03 00 00 00 00 00 00 00 | d...............

0011E0: 40 60 23 62 79 55 00 00  04 00 00 00 00 00 00 00 | @`#byU..........

0011F0: 38 00 00 00 00 00 00 00  05 00 00 00 00 00 00 00 | 8...............

Block 9

001200: F0 12 A4 63 79 55 00 00  00 9C 7A B9 71 6D 0F A5 | ..cyU...z.qm.

001210: 06 00 00 00 8A 09 00 00  00 00 00 00 00 00 00 00 | ................

001220: D0 39 C0 F1 FC 7F 00 00  0D 7C 23 62 79 55 00 00 | 9���...|#byU..

001230: 00 02 00 00 00 00 00 00  4B 4C 00 00 00 00 00 00 | ........KL......

001240: 10 3B C0 F1 FC 7F 00 00  70 F6 A3 63 79 55 00 00 | .;���..p..cyU..

001250: 30 3A C0 F1 FC 7F 00 00  78 79 23 62 79 55 00 00 | 0:���..xy#byU..

001260: 18 3B C0 F1 FC 7F 00 00  04 00 00 00 04 00 00 00 | .;���..........

001270: 01 00 00 00 00 00 00 00  00 96 98 00 00 00 00 00 | .........�.....

001280: 00 02 00 00 00 00 00 00  7D 53 C0 F1 FC 7F 00 00 | ........}S����..

001290: 40 8C 23 62 79 55 00 00  90 72 23 62 79 55 00 00 | @�#byU..�r#byU..

0012A0: 10 3B C0 F1 FC 7F 00 00  00 9C 7A B9 71 6D 0F A5 | .;���...�z�qm.�

0012B0: 40 8C 23 62 79 55 00 00  87 0C 46 70 88 7F 00 00 | @�#byU..�.Fp��..

0012C0: 00 00 00 00 20 00 00 00  18 3B C0 F1 FC 7F 00 00 | .... ....;���..

0012D0: 00 00 00 00 04 00 00 00  7B 78 23 62 79 55 00 00 | ........{x#byU..

0012E0: 00 00 00 00 00 00 00 00  B5 58 3D E6 68 20 0F FE | ........�X=�h .�

0012F0: 90 72 23 62 79 55 00 00  10 3B C0 F1 FC 7F 00 00 | �r#byU...;���..

001300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

001310: B5 58 3D 8A AF 07 04 AB  B5 58 C3 E6 A3 04 ED AB | �X=��..��X��.��

001320: 00 00 00 00 FC 7F 00 00  00 00 00 00 00 00 00 00 | ....�.........

001330: 00 00 00 00 00 00 00 00  D3 68 04 71 88 7F 00 00 | ........�h.q��..

001340: 88 4D 03 71 88 7F 00 00  B0 9B A4 70 88 7F 00 00 | ..M.q��..���p��..

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  90 72 23 62 79 55 00 00 | ........r#byU..

001370: 10 3B C0 F1 FC 7F 00 00  BA 72 23 62 79 55 00 00 | .;���..r#byU..

001380: 08 3B C0 F1 FC 7F 00 00  1C 00 00 00 00 00 00 00 | .;���.........

001390: 04 00 00 00 00 00 00 00  73 53 C0 F1 FC 7F 00 00 | ........sS���..

0013A0: 7D 53 C0 F1 FC 7F 00 00  8A 53 C0 F1 FC 7F 00 00 | }S���..S���..

0013B0: 93 53 C0 F1 FC 7F 00 00  00 00 00 00 00 00 00 00 | S���.........

0013C0: 97 53 C0 F1 FC 7F 00 00  AE 53 C0 F1 FC 7F 00 00 | S���..S���..

0013D0: C3 53 C0 F1 FC 7F 00 00  EC 53 C0 F1 FC 7F 00 00 | S���..S���..

0013E0: FE 53 C0 F1 FC 7F 00 00  12 54 C0 F1 FC 7F 00 00 | S���...T���..

0013F0: 22 54 C0 F1 FC 7F 00 00  33 54 C0 F1 FC 7F 00 00 | "T����..3T����..

Block 10

001400: 43 54 C0 F1 FC 7F 00 00  70 54 C0 F1 FC 7F 00 00 | CT����..pT����..

001410: 82 54 C0 F1 FC 7F 00 00  93 54 C0 F1 FC 7F 00 00 | T����..T����..

001420: E8 54 C0 F1 FC 7F 00 00  FE 54 C0 F1 FC 7F 00 00 | T����..T����..

001430: 1E 55 C0 F1 FC 7F 00 00  86 55 C0 F1 FC 7F 00 00 | .U����..U����..

001440: BA 55 C0 F1 FC 7F 00 00  24 56 C0 F1 FC 7F 00 00 | U����..$V����..

001450: 43 56 C0 F1 FC 7F 00 00  58 56 C0 F1 FC 7F 00 00 | CV����..XV����..

001460: 6F 56 C0 F1 FC 7F 00 00  5B 5C C0 F1 FC 7F 00 00 | oV����..[\����..

001470: 76 5C C0 F1 FC 7F 00 00  9F 5C C0 F1 FC 7F 00 00 | v\����..\����..

001480: AA 5C C0 F1 FC 7F 00 00  BD 5C C0 F1 FC 7F 00 00 | \����..\����..

001490: D0 5C C0 F1 FC 7F 00 00  FD 5C C0 F1 FC 7F 00 00 | \����..\����..

0014A0: 0C 5D C0 F1 FC 7F 00 00  1F 5D C0 F1 FC 7F 00 00 | .]����...]����..

0014B0: 3B 5D C0 F1 FC 7F 00 00  71 5D C0 F1 FC 7F 00 00 | ;]����..q]����..

0014C0: 93 5D C0 F1 FC 7F 00 00  A3 5D C0 F1 FC 7F 00 00 | ]����..]����..

0014D0: CF 5D C0 F1 FC 7F 00 00  DA 5D C0 F1 FC 7F 00 00 | ]����..]����..

0014E0: EE 5D C0 F1 FC 7F 00 00  FF 5D C0 F1 FC 7F 00 00 | ]����..]����..

0014F0: 07 5E C0 F1 FC 7F 00 00  27 5E C0 F1 FC 7F 00 00 | .^����..'^����..

001500: 38 5E C0 F1 FC 7F 00 00  45 5E C0 F1 FC 7F 00 00 | 8^����..E^����..

001510: 66 5E C0 F1 FC 7F 00 00  7B 5E C0 F1 FC 7F 00 00 | f^����..{^����..

001520: 86 5E C0 F1 FC 7F 00 00  8E 5E C0 F1 FC 7F 00 00 | ^����..^����..

001530: A6 5E C0 F1 FC 7F 00 00  B9 5E C0 F1 FC 7F 00 00 | ^����..^����..

001540: DA 5E C0 F1 FC 7F 00 00  EC 5E C0 F1 FC 7F 00 00 | ^����..^����..

001550: 0D 5F C0 F1 FC 7F 00 00  63 5F C0 F1 FC 7F 00 00 | ._���..c_���..

001560: 80 5F C0 F1 FC 7F 00 00  8D 5F C0 F1 FC 7F 00 00 | _���..._���..

001570: A5 5F C0 F1 FC 7F 00 00  BD 5F C0 F1 FC 7F 00 00 | _���..._���..

001580: CE 5F C0 F1 FC 7F 00 00  E2 5F C0 F1 FC 7F 00 00 | _���..._���..

001590: 00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00 | ........!.......

0015A0: 00 F0 CF F1 FC 7F 00 00  10 00 00 00 00 00 00 00 | .....�.........

0015B0: FF FB 8B 17 00 00 00 00  06 00 00 00 00 00 00 00 | ................

0015C0: 00 10 00 00 00 00 00 00  11 00 00 00 00 00 00 00 | ................

0015D0: 64 00 00 00 00 00 00 00  03 00 00 00 00 00 00 00 | d...............

0015E0: 40 60 23 62 79 55 00 00  04 00 00 00 00 00 00 00 | @`#byU..........

0015F0: 38 00 00 00 00 00 00 00  05 00 00 00 00 00 00 00 | 8...............

Block 11

001600: 09 00 00 00 00 00 00 00  07 00 00 00 00 00 00 00 | ................

001610: 00 60 03 71 88 7F 00 00  08 00 00 00 00 00 00 00 | .`.q.....�......

001620: 00 00 00 00 00 00 00 00  09 00 00 00 00 00 00 00 | ................

001630: 90 72 23 62 79 55 00 00  0B 00 00 00 00 00 00 00 | .r#byU..........

001640: E8 03 00 00 00 00 00 00  0C 00 00 00 00 00 00 00 | ................

001650: E8 03 00 00 00 00 00 00  0D 00 00 00 00 00 00 00 | ................

001660: E8 03 00 00 00 00 00 00  0E 00 00 00 00 00 00 00 | ................

001670: E8 03 00 00 00 00 00 00  17 00 00 00 00 00 00 00 | ................

001680: 00 00 00 00 00 00 00 00  19 00 00 00 00 00 00 00 | ................

001690: 59 3E C0 F1 FC 7F 00 00  1A 00 00 00 00 00 00 00 | Y>���..........

0016A0: 00 00 00 00 00 00 00 00  1F 00 00 00 00 00 00 00 | ................

0016B0: EE 5F C0 F1 FC 7F 00 00  0F 00 00 00 00 00 00 00 | ._���...........

0016C0: 69 3E C0 F1 FC 7F 00 00  00 00 00 00 00 00 00 00 | i>���.....

0016D0: 00 00 00 00 00 00 00 00  00 2B 9C 7A B9 71 6D 0F | .....+�z�qm.

0016E0: A5 5E FF 17 F2 7E AA 5A  AC 78 38 36 5F 36 34 00 | ^.~Z.x86_64.

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 | ................