File Systems Project Writeup

Team: JODA

Members: James Dixon, Justin Hellweg, Devon Huang, Omar Thiongane

Student IDs: 922440659, 922692586, 916940666, 922164097

Github Usernames: JD499, Jus1927, Novedh, ot409

Github Link:
https://github.com/CSC415-2024-Spring/csc415-filesystem-Novedh

Table of Contents

Plans:

Directory Entry:

This is the planning that was done for the "What is a Directory Entry?" assignment that was done as a group in class. At this stage, the main goals were creating a valid C structure that would compile and contain the key pieces of information a directory entry should have. There was minimal planning for future stages of the project at this point as our group was still in the process of learning about the file system.

Our strategy to complete this assignment was to compare our notes on directory entries from the lectures, video recordings, the textbook, and other research about directory entries. After compiling the information we had as a group, we decided to include the following variables in our directory entry: name (with max of 255 characters), location, size (in bytes), isDirectory, permissions (using the octal system), the most recent modification date, the most recent access date, and the date the directory entry was created. We shared our contact info and created a Discord server for the File System group project.

After defining the initial C structure and verifying that it compiles, we submitted this stage of the project. We initially used the wrong variable types for our directory entry. After receiving the feedback for this step, we made sure to avoid using pointers in the directory entry definition and to change the type of our dates to be binary values so that they will be more compact and more straightforward to work with.

File System Design:

This is the planning that was completed for the "File System Design" assignment. The main goal for this assignment was to make some decisions about how our group will go about implementing the file system including important aspects, such as the volume control block, free space management, our directory entry, and any meta data that we are planning to track.

For this assignment, we made most of the major decisions together in a meeting. First, we discussed how we would handle our free space management as this would impact other parts of our file system. We initially considered extents as they would be able to handle discontiguous memory allocation more eloquently than some of the other solutions for free space management and also would take up less room in memory. In the end, we decided to use a bitmap as we thought it would be the simplest solution to implement. We decided that if a bit is set to 0 that represents that the block is free and if the bit is 1 that means that it is being used in the file system. While making decisions for our file system we prioritized making a simple design that would be more feasible to implement within the time constraints of the class. As a group, we briefly discussed the contents of the volume control block, agreed to update our directory entry with the feedback we received, and mentioned some of the metadata we might want to track.

After meeting as a group, we split up different sections of the assignment and fleshed out the specifics individually. If a section had C code, we verified that it compiled without throwing an error and added comments to clearly explain the purpose and meaning of each variable. After reviewing our section, we confirmed on our group's Discord server with any updates or changes for that section.

We implemented the feedback we received in this section for the Milestone One phase of the project. The major changes we needed to make were to define the signature in our volume control block as well as to specify the location of the free space management.

Milestone One:

After working on the initial File System group submissions together, we began to work more asynchronously for Milestone One. We frequently communicated on our group's Discord server as questions popped up for a section we were working on or about the file system project in general.The majority of the coding for the project was done separately, but the Discord server provided a way for team members to reach out and help each other. People worked on tasks for the Milestone One on a voluntary basis with reaching out to the group as needed.

Devon took the lead in Milestone One in making and sharing the GitHub repo and implementing some steps of the project including the VCB and root directory. Following the Steps for milestone 1.pdf we started by looking at the fsInit.c file and writing down our structs for directory entries and the volume control block from our file system design group activity. We first needed to see if the volume even needed to be formatted by reading block 0 and checking the signature. If it wasn't ours then we would have to initialize the volume.

James then wrote the free space management system using a bitmap as our group planned. Then we would have to initialize the free space map by finding out how many blocks we need for it and setting it to use on the free space map.

Justin made a Notion page to help track the requirements for the project and made a template for the PDF writeup. From there, team members could comment on which tasks they were currently working on and easily update each other on which tasks had been completed. After we would have to initialize the root directory by using the same function used to create directories we can check if it has a parent if not then it would be the new root directory. We set up . and .. to point to itself. The specific sections group members worked on was recorded in the group table and all members helped to review and edit the document.

Milestone Two:

As Milestone Two was not a graded deadline, our group did not plan to have all of the features included in Milestone Two finished by the recommended date. We continued to pick up tasks for this phase of the project on a voluntary basis. We primarily used our Discord server to ask questions and communicate about the project as needed, although we also used GitHub and Notion to track remaining tasks of the project and communicate about what we were working on. Devon produced a lot of the code for this phase of the project and James contributed as well.

Final Deliverable:

This section of the project is expected to have a complete file system that can transfer data between the Linux file system and ours, in addition to supporting a few basic Linux functions. This involves a lot of interaction between our fsDir functions and the b_io functions. As such the completion of this stage needs a strong Milestone Two foundation, that functions with the io operations.

Our group maintained a similar method of dividing the responsibilities of the project and continued to communicate on our Discord server as needed. The frequency of contributions continued to increase as the assignment deadline became closer.

Descriptions:

Directory Entry Structure:

The directory entry structure or DE struct is defined in our global.h file. Our file system's directory entry contains 8 variables ordered where the largest variables in memory are at the top of the struct, so that the structure is packed efficiently in memory.

Our directory entry contains the variable name which is a user-readable filename with a maximum number of characters of MAX_FILENAME_LEN which we have set at 255. Next, we have three time_t variables for createTime, modTime, and accessTime. After that, we have fileID which is a unique integer used to refer to the file internally within the file system. We have another int loc which represents the location of the file on the disk (and is specifically not a pointer). Finally, we have two unsigned ints size (in bytes) and isDir (0 if it is a file, 1 if it is a directory).

Volume Control Block Structure:

The VCB is the Volume control Block which contains information such as signature, the numberOfBlocks in the volume, blockSize (in bytes), when free space starts (freeStart) and when the root directory starts (rootStart), and now the size of the free space (freeSize). Each of the variables in the VCB struct are unsigned integers with a fixed size of 64 bits. The volume control block or VCB structure is defined in the global.h file of our file system. From there, our specific file system volume signature '415JODA' is defined in the fsInit.c file along with a pointer to the VCB.

Free Space Structure:

The free space structure is responsible for managing and allocating free blocks in the file system. Our group implemented our free space structure using a bitmap. An outline of the 8 functions in our bitmap is given in our bitMap.h file and the functions are implemented in the bitMap.c file.

There are three key functions that allow us to manipulate the bits in the bitmap where each bit represents a block of 512 bytes: setBit, clearBit, and getBit. The function setBit(int blockNum) marks the blockNumth bit as used. The function clearBit(int n) marks the nth bit as free. The function getBit(int n) returns the value of the nth bit in the bitmap.

There are several other functions used to maintain the bitmap, store it, and retrieve it from memory properly. The function loadFSM() loads the free space map into memory. The function initFreeSpaceMap initializes the free space map. The function allocateBlocks(int numBlocksRequested) will allocate numBlocksRequested blocks. The function freeBlocks(int index, int numBlocks) will free numBlocks blocks starting at index. The function exitFreeMap() is used to update the free space map on disk and free any memory that was allocated for it.

File System Structure:

The Directory system contains information about a file or directory in the file system. It includes name of directory, directory entry, location on disk, size, and a flag indicating if it is a directory or not.

A bitmap is then used to track the available free space. Every bit in the bitmap represents a block of 512 bytes. If a bit is set to 0, it is free and if a bit is set to 1 it is currently being used by the file system.

As a user gives the file system commands, creates files and directories, moves directory entries, and other available commands, the file system will maintain the bitmap to be an accurate representation of which blocks are being used and which are not. Then, before the file system is fully closed and freed from memory it is imperative that the current working version of the bitmap is saved to the permanent disk. This will give the file system permanency and make sure that a user can reopen their files after closing them.

Issues:

Memory Management:

It seems like our main struggle revolved around memory management. To tackle this issue, we found ourselves resorting to a series of printf statements to meticulously trace the source of memory-related problems as well as the use of valgrind. This approach allowed us to pinpoint precisely where memory was being allocated, deallocated, or potentially leaked within the codebase.

Memory management can be a tricky aspect of software development, especially when dealing with dynamic memory allocation and deallocation. Without proper management, memory leaks or segmentation faults can occur, leading to unexpected behavior and program crashes.By strategically placing printf statements throughout the code, we were able to track the lifecycle of allocated memory blocks. This included identifying where memory was allocated, ensuring it was properly deallocated when no longer needed, and detecting any instances of memory leakage.

Additionally, these printf statements served as invaluable tools for debugging, providing insights into the flow of execution and uncovering any unforeseen interactions or dependencies between different parts of the code.While printf-based debugging may seem rudimentary, it proved to be an effective technique in my case, offering a straightforward yet powerful means of diagnosing memory-related issues. Moving forward, I plan to incorporate more robust debugging tools and techniques to streamline the process and enhance my proficiency in memory management.

Debugging:

I am very used to using larger ide's such as Intellij products which allow proper step debugging where I can have lots of breakpoints and see all the variables and their values in play. In this setup I can only really use vscode and hope to get by using print statements to track variables. This is very messy and makes debugging take a lot of time.

Functions:

fsInit:

int initFileSystem(int numberofBlocks, int blockSize)

Responsible for starting our file system. The function takes the number of blocks the file system will have access to as well as the size of those blocks. It will check if our file system is already initialized by reading a volume control block to memory, and then comparing the signature in the control block with the one associated with our file system. If the comparison is true, this means our file system is already initialized here and we just need to load in the free space map and root directory. Otherwise we need to initialize our file system, setting up a VCB with a root directory and free space map, saving it to disk.

void exitFileSystem()

This function is responsible for cleanly exiting the file system, it frees any remaining memory, and saves relevant structures such as the free map and volume control block to disk.

bitMap:

void loadFSM()

The function first checks if the freeSpaceMap already has space in memory allocated, if it does then it frees it to allocate new clean space. This space is then filled with the freeSpaceMap that is read from disk.

void setBit(int blockNum)

This is one of 3 crucial functions for managing our freeSpaceMap. The function uses bitwise operations to set a bit representing a block, to 1. This represents a block as being in use.

void clearBit(int blockNum)

This function uses bitwise operations to set a bit representing a block to 0. This sets a block as being unused.

void getBit(int blockNum)

This function uses bitwise operations to return the value of a bit in our freeSpaceMap. The function gets called whenever we need to check if a block in the map is used or free.

int initFreeSpaceMap(int numBlocks, int blockSize)

This function is called by initFileSystem to create our initial freeSpaceMap. It determines the amount of space needed for the map based on the number of blocks and the size of the blocks given to our file system. It will set the newly allocated freeSpaceMap's values to 0, setting them as free initially. Then the function marks the first block as used for the volume control block, and additionally the next n blocks needed for the map itself. It will then write the map to disk starting at block 1, and return its size in blocks.

int allocateBlocks(int numBlocksRequested)

allocateBlocks is used whenever anything needs to be saved in our filesystem. The function will receive the number of blocks the caller needs and then search through every block in our free space map linearly until it finds a continuous number of blocks that matches the caller's request. This series of blocks matches the amount of space the caller needs to allocate, this space is marked as used in the map and then written to disk. The function finally returns the starting block of the continuous series to the caller.

int freeBlocks(int index, int numBlocks)

This function is called whenever something is deleted from our file system. The function takes the starting index aka block number, and then the number of blocks to delete. It will read in the map, and then mark the series as free by clearing the bits associated with the blocks (index + numblocks). Finally it writes the updated map to disk.

void exitFreeMap()

The exit function just exits the map cleanly, writing everything to disk and freeing the map in memory.

fsDir:

void loadRoot()

This function is similar to the loadFreeSpaceMap function as it is called when the file system is marked as already initialized. The function is responsible for reading the root directory from disk, and setting up the cwd at initialization which should be pointing to root.

int createDirectory(int numEntries, DE *parent)

createDirectory is our low level function to create a directory. It is used in creating the root and all sub directories. It accepts the number of entries the directory should be able to hold, to initialize their locations and to determine the size of the directory. Additionally it accepts the parent which is used to determine if the directory is root (ie parent == NULL) or a sub directory in which case the '..' should point to the DE *parent directory. The function allocates space for the directory, sets up the '.' and '..' pointers, updates any metadata such as the isDir check, and then writes the newly created directory to disk, returning the location of the directory to the caller.

int createFile(int fileSize, DE *parent)

createFile is nearly identical to createDirectory in structure. The significant differences are that size is determined by the caller's input instead of directory entries as a file can not have directory entries. Following this principle the function does not need to worry about ‘.’ or ‘..’ pointers. Instead the DE only has a default name, filesize, location, and marks isDir as 0 signifying that this is a file not a directory. The file is then written to disk and the location of the file is returned to the caller.

int makeFile(char *pathname, int fileSize)

This is the higherLevel file to create a file. The function takes the initial size of a file and the pathname of the file it will create. Using parsePath to get info about the parent directory as well as to determine if a file can be made in the location. The file will be created using the createFile function, then the function will call findUnusedDE to find space in the parent to save the file to and have its metadata updated here, and its metadata updated in the parentDirectories respective index. The updated parent is then written to disk and the location of our new file is returned.

DE *getDEInfo(char *filename)

This function is responsible for finding a Directory entry, using parsePath, if it exists and returning it to the caller. It is mostly used in b_io operations.

int findInDir(DE *parent, char *string)

This function looks through the given parent directory to find a directory entry in it whose name matches the given string. If it finds this entry it will return the index of the entry, otherwise it returns -1.

DE *loadDir(DE *de)

This function will allocate memory for a directory entry, look for its location on disk and then load it into memory.

DE *loadDirByLoc(int loc)

This function is similar to loadDir but instead of taking a directory entry as input, it will take the block index location of the directory on disk.

int parsePath(const char *path, ppRetStruct *ppInfo)

parsePath is the vital function that is responsible for 90% of the actual work in fsDir.c functions. The function receives a path and a custom ppInfo struct that is used to store most of the relevant info the function finds. ppInfo contains a parentDirectory, lastElementName, and lastElementIndex. This information can be used to find all the information about a directoryEntry and its parent directoryEntry. The function first checks if it is provided a path and ppInfo. It then determines if the path it is given is absolute or relative. It then tokenizes the string based on the "/" symbol. Then it follows the path down till it finds the desired directory entry.

int findUnusedDE(DE *dir)

This function similarly to findInDir scans through the entries of a DE, in this case instead of comparing the entries name to a path name, it compares the name to "" which represents an empty entry, returning the index of the empty entry position.

void writeDir(DE *de)

This function takes a directory entry, and writes it to disk.

int fs_mkdir(const char *pathname, mode_t mode)

Mkdir is the high level function to create a directory, similar to makeFile. This function takes a pathname and mode. The function uses parse path to check if the path is valid and to make sure the file can be created. The directory is then created using the lower level createDirectory function. This directory is then loaded into memory and has its parents metadata updated to reflect that the subdirectory now belongs to it. Finally the updated parent is written to disk.

char *fs_getcwd(char *pathname, size_t size)

This function returns the current working directory path string, it's mostly used for the print working directory command.

int fs_setcwd(char *pathname)

The fs_setcwd function sets the current working directory of the filesystem to the specified directory path. It involves updating a global variable or state within the filesystem to reflect the new current working directory.

int fs_rmdir(const char *pathname)

The fs_rmdir function removes the directory specified by the given path from the filesystem, provided that it is empty. It involves removing the directory entry and deallocating any associated resources, such as directory blocks and entries.

int fs_isFile(char *filename)

The fs_isFile function checks if the given path corresponds to a file in the filesystem. It requires traversing the filesystem hierarchy and determining if the specified path points to a file or directory entry.

int fs_isDir(char *pathname)

The fs_isDir function checks if the given path corresponds to a directory in the filesystem. Similar to fs_isFile, it involves traversing the filesystem hierarchy and verifying if the specified path points to a directory entry.

int fs_delete(chat *filename)

The fs_delete function deletes a file specified by the given path from the filesystem. It requires removing the corresponding file entry from the filesystem and releasing any associated resources.

fdDir *fs_opendir(const char *pathname)

The fs_opendir function opens a directory specified by the given path and returns a handle to it for subsequent directory operations. It involves locating the directory entry corresponding to the specified path and returning a reference or handle to it.

struct fs_diriteminfo *fs_readdir(fsDir *dirp)

The fs_readdir function reads the contents of the directory specified by the directory handle and returns information about each entry. It requires iterating over the directory entries and retrieving information such as file names, sizes, and attributes.

int fs_closedir(fsDir *dirp)

The fs_closedir function closes the directory specified by the directory handle. It typically involves releasing any resources associated with the directory handle and performing cleanup operations.

int fs_move(char srcPath, char *destPath)

The move function takes a source path and a destination path. It calls parse path on both to get the relevant directory entry info. For the source it needs to know the source's name location and other metadata, and for the destination it needs to find if there is available space for a new directory entry. If the function determines there is space in the destination directory, it copies over the source directoryEntry meta data to the next free DE slot in the destination, the function then goes to the source's parent to clear the source information marking the DE is free. The function then writes the updated destination directory and source's Parent to disk.

int fs_stat(const char *path, struct fs_stat *buf)

The fs_stat function retrieves information about a file or directory specified by the given path, such as its size, type, permissions, and timestamps. It involves accessing the metadata associated with the specified file or directory entry.

b_io:

void b_init()

This function is responsible for initializing our IO operations. It marks our file control block array as NULL so that IO functions can use them.

b_io_fd b_getFCB()

This function is used to get a free file control block (FCB) from the file control block array. It does this by iterating through the array until it finds a FCB that is marked as NULL i.e. unused.

b_io_fd b_open(char *filename, int flags)

This b_open function opens a buffer file. It starts the file if needed, gets the file information and allocates memory for the buffer. It receives a free file control block (FCB) and puts it as file information,buffer,and index. It then returns it as a descriptor of the open file. We did not implement flag handling.

int b_seek(b_id_fd fd, off_t offset, int whence)

We did not implement this :(

int b_write(b_io_fd fd, char *buffer, int count)

This function is the opposite of b_read where it takes a callers buffer, transfers the data into our own buffer and then writes it to disk.

int b_read(b_io_fd fd, char *buffer, int count)

This function is pretty much unchanged from assignment 5, it takes data from disk and writes it into the caller's buffer.

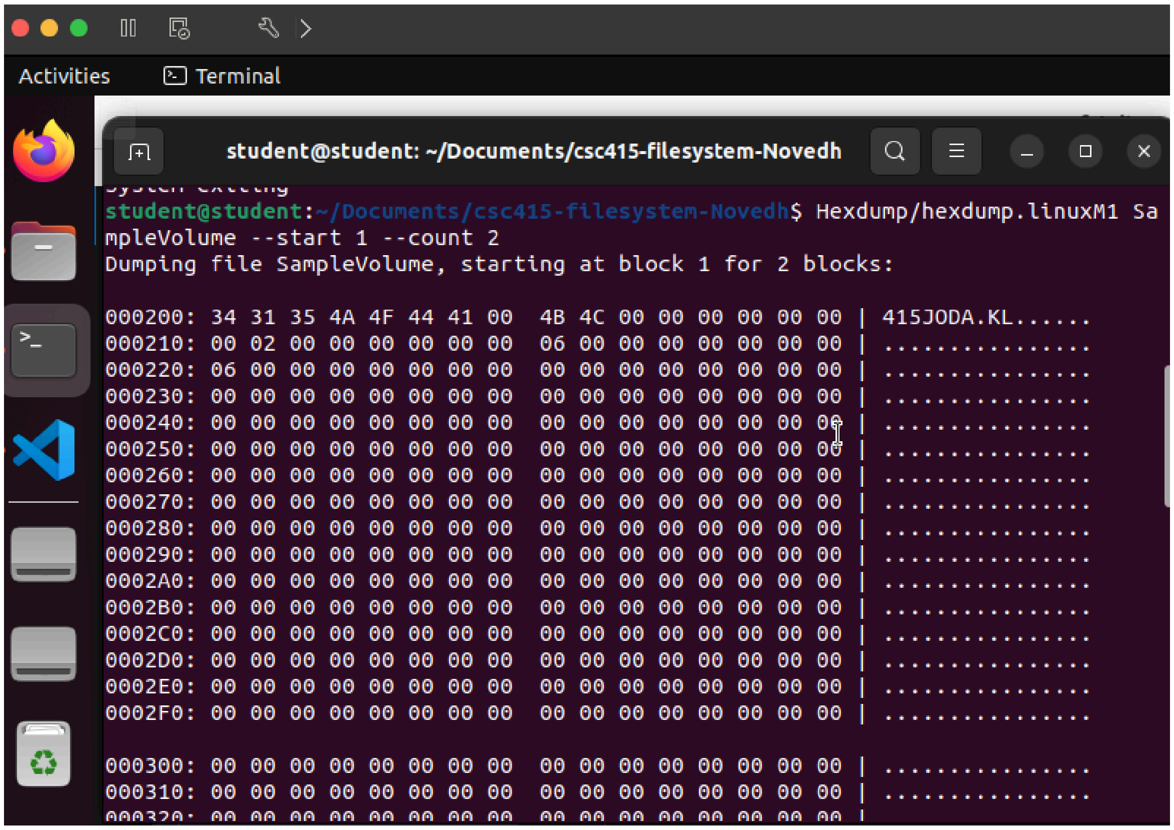
int b_close(b_io_fd fd)

This function writes the updated file to disk, and should ideally free all the file control blocks that are no longer in use as well as any malloced memory, and write the last of the dirty buffer.

Screenshots:

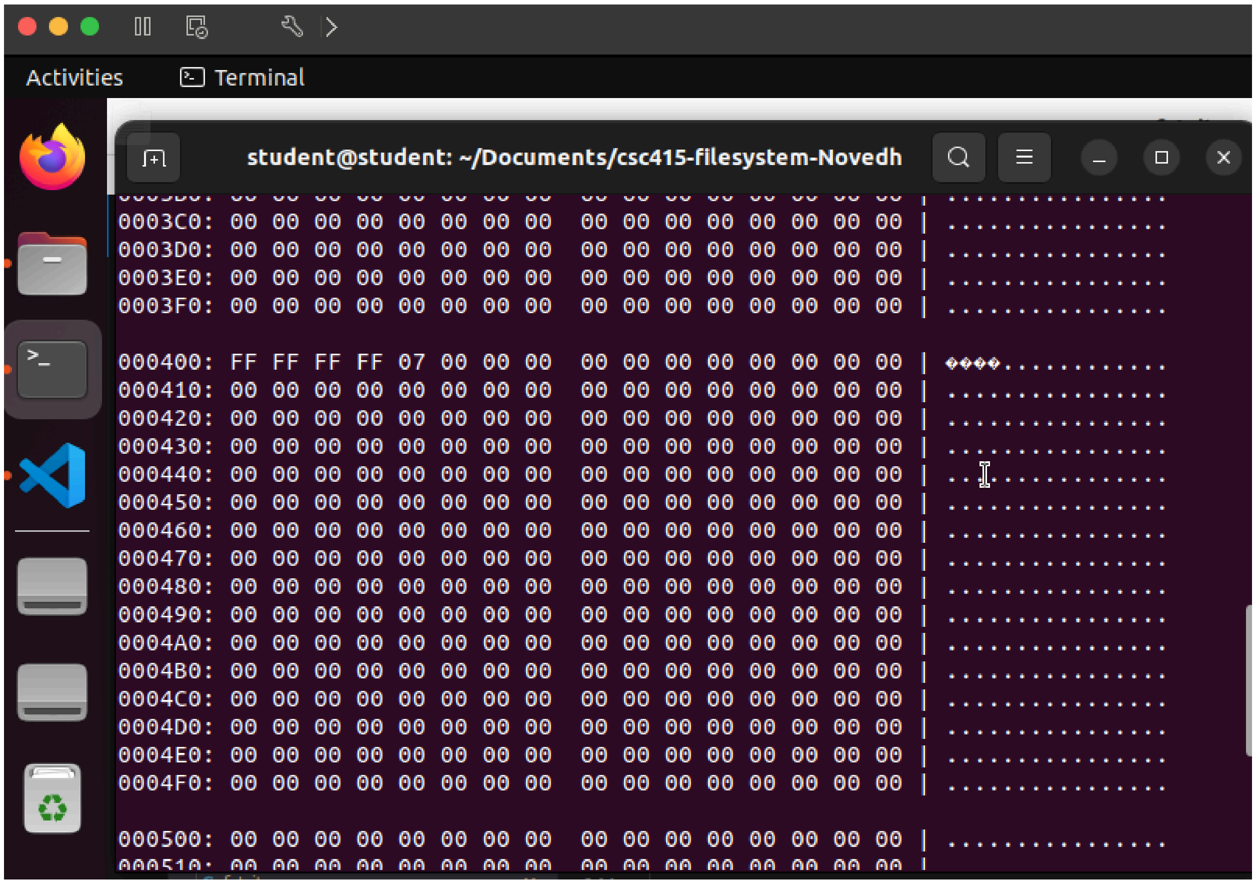
Dump

//show a dump (using provided HexDump utility) that show VCB, Free Space, and complete root directory



```
student@student:~/Documents/csc415-filesystem-Novedh$ Hexdump/hexdump.linuxM1 Sa
mpleVolume --start 1 --count 2
Dumping file SampleVolume, starting at block 1 for 2 blocks:

000200: 34 31 35 4A 4F 44 41 00  4B 4C 00 00 00 00 00 00 | 415JODA.KL......
000210: 00 02 00 00 00 00 00 00  06 00 00 00 00 00 00 00 | ................
000220: 06 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000230: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000270: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000310: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000320: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |
```

This Dump shows Block 1 which represents our VCB, 34 31 35 4A 4F 44 41 is our volume signature, which is 415JODA



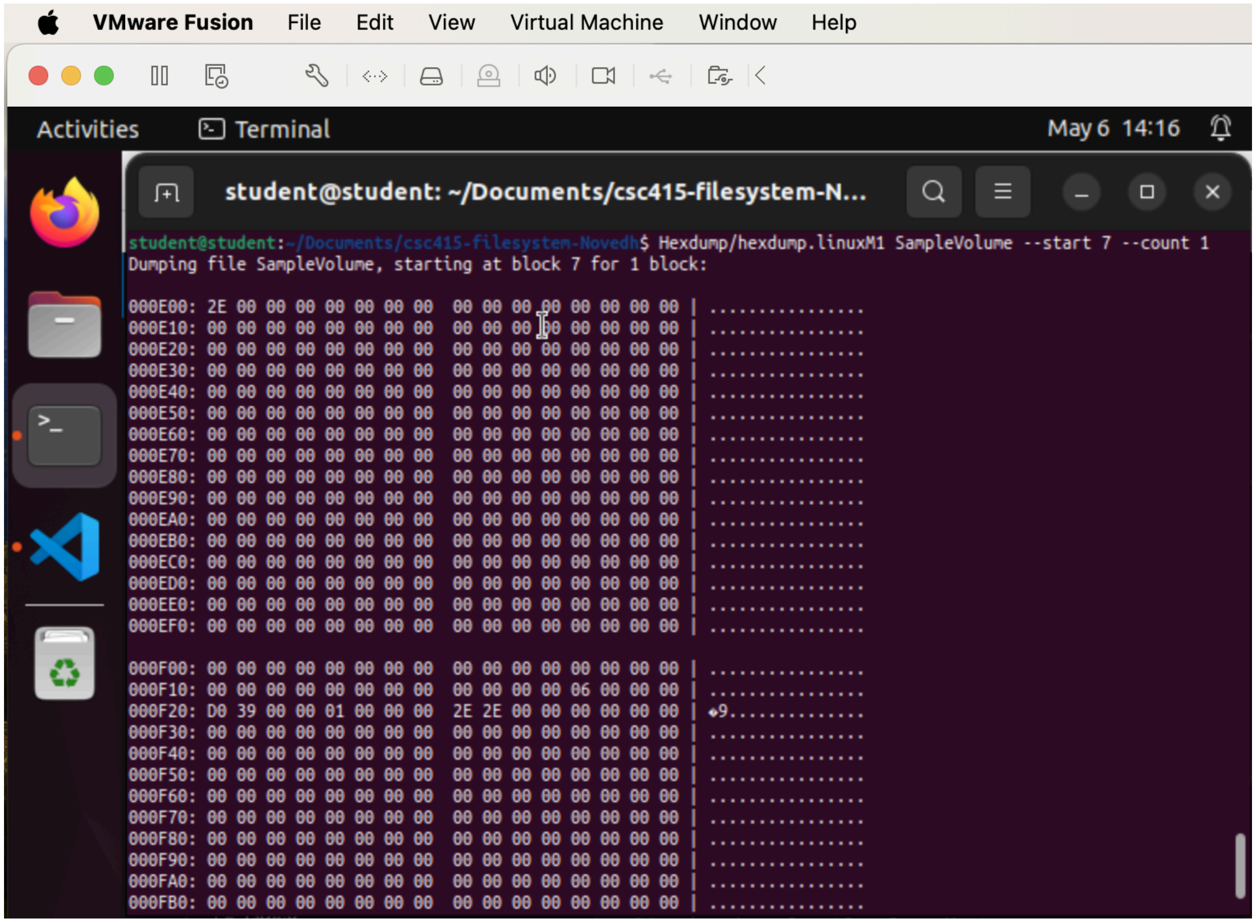
```
0003C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000400: FF FF FF FF 07 00 00 00  00 00 00 00 00 00 00 00 | ����............
000410: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000420: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000430: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000440: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000450: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000460: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000470: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000480: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000490: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0004F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000500: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000510: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |
```

This dump shows the bitmap when we convert the hex FF FF FF FF 07 to binary we get

11111111 11111111 11111111 11111111 00000111

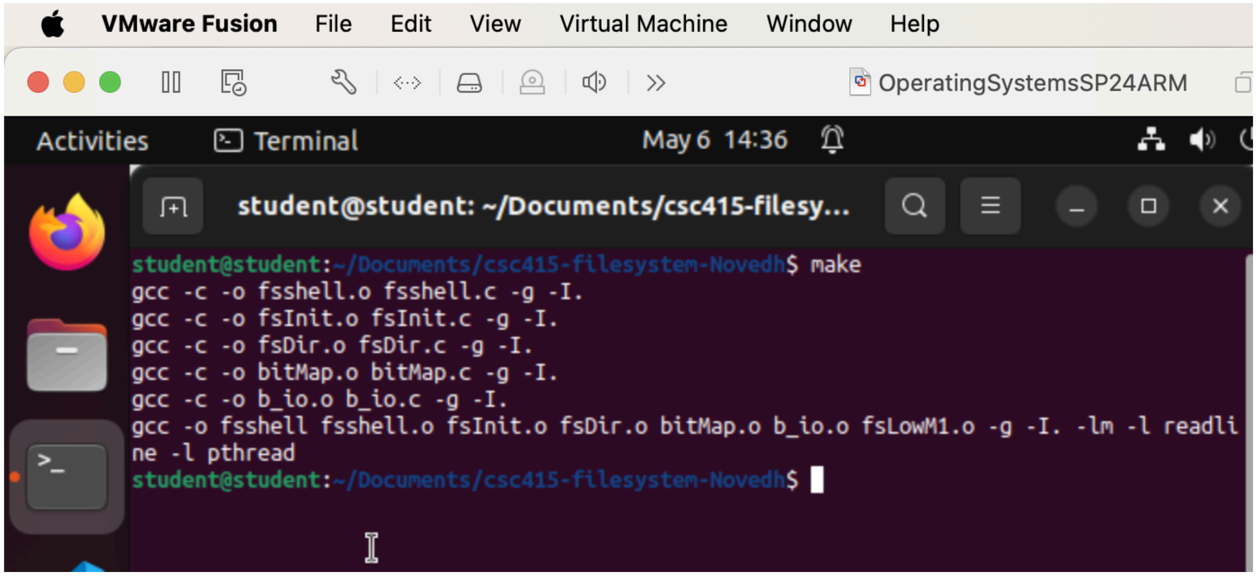
Which shows the 35 blocks that are used
1 for the VCB
5 for the free space map
29 for the root directory
Our root directory is 29 blocks because our current DE size is 296 bytes and we chose to have 50 entries in our root directory,
296 * 50 = 14800
Then we see how many blocks we will need
(14800+511)/512 = 29 blocks

This dump shows the root directory starting on block 6 (block 7 for the hexdump)



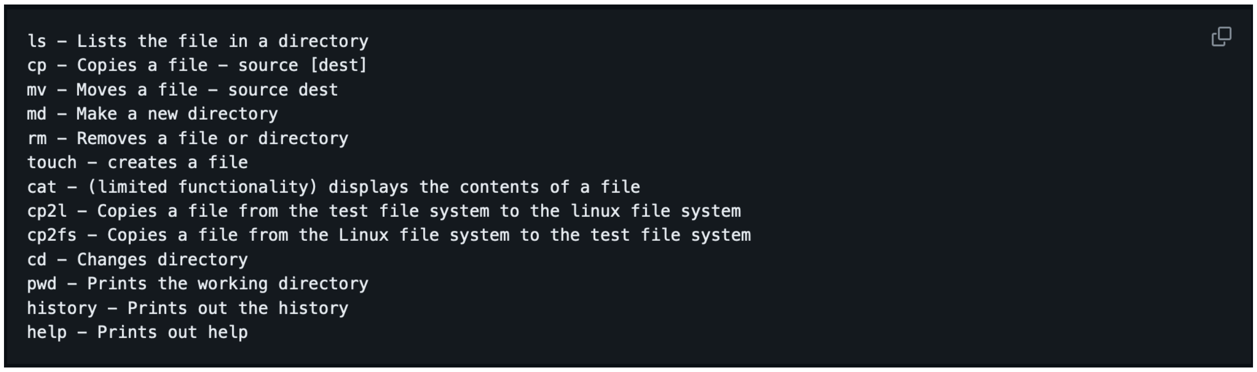
```
student@student:~/Documents/csc415-filesystem-Novedh$ Hexdump/hexdump.linuxM1 SampleVolume --start 7 --count 1
Dumping file SampleVolume, starting at block 7 for 1 block:

000E00: 2E 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000E90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000ED0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000EF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000F00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F10: 00 00 00 00 00 00 00 00  00 00 00 00 06 00 00 00 | ................
000F20: D0 39 00 00 01 00 00 00  2E 2E 00 00 00 00 00 00 | ◆9..............
000F30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F40: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000F90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000FB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Running make:



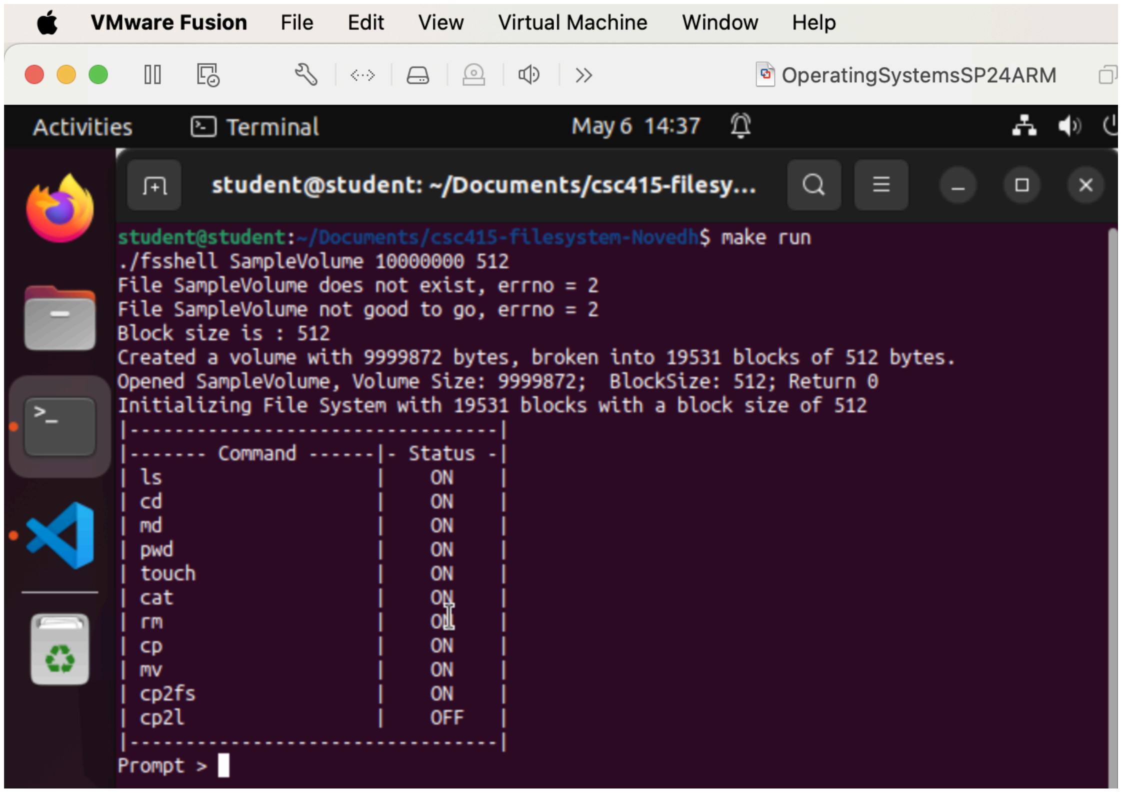
```
student@student:~/Documents/csc415-filesystem-Novedh$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o fsDir.o fsDir.c -g -I.
gcc -c -o bitMap.o bitMap.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsDir.o bitMap.o b_io.o fsLowM1.o -g -I. -lm -l readli
ne -l pthread
student@student:~/Documents/csc415-filesystem-Novedh$ 
```

Screenshots of Commands in ReadMe



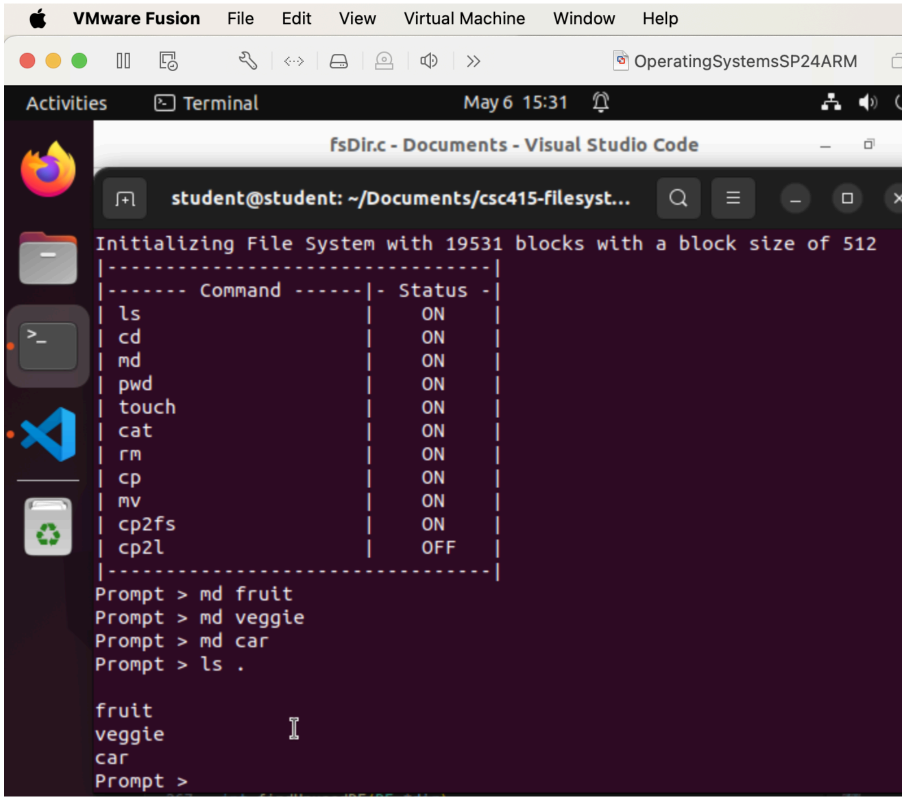
```
ls - Lists the file in a directory
cp - Copies a file - source [dest]
mv - Moves a file - source dest
md - Make a new directory
rm - Removes a file or directory
touch - creates a file
cat - (limited functionality) displays the contents of a file
cp2l - Copies a file from the test file system to the linux file system
cp2fs - Copies a file from the Linux file system to the test file system
cd - Changes directory
pwd - Prints the working directory
history - Prints out the history
help - Prints out help
```

This shows the Initialization of our file system :



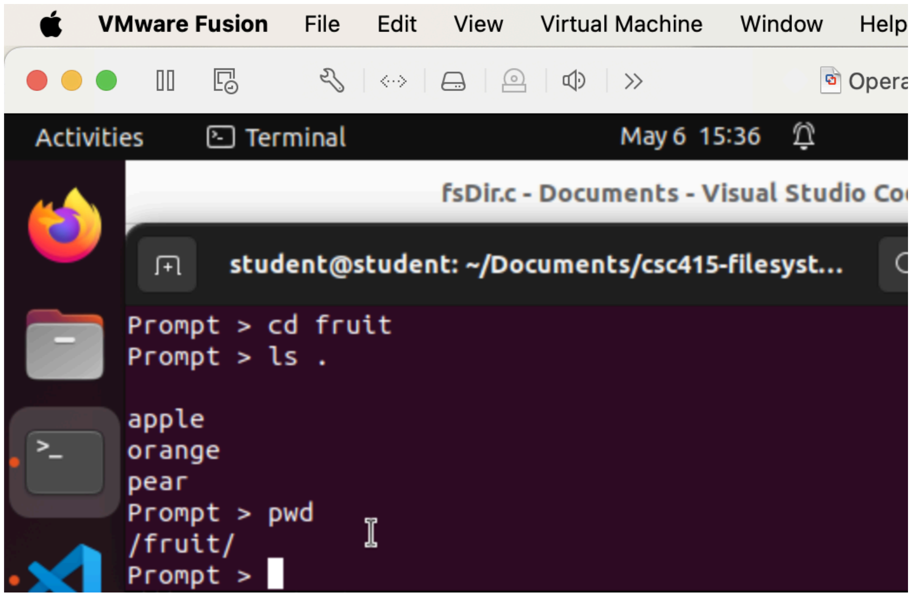
```
student@student:~/Documents/csc415-filesystem-Novedh$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
|-------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|-------------------------------|
Prompt >
```

This shows the md and ls command on the root directory:



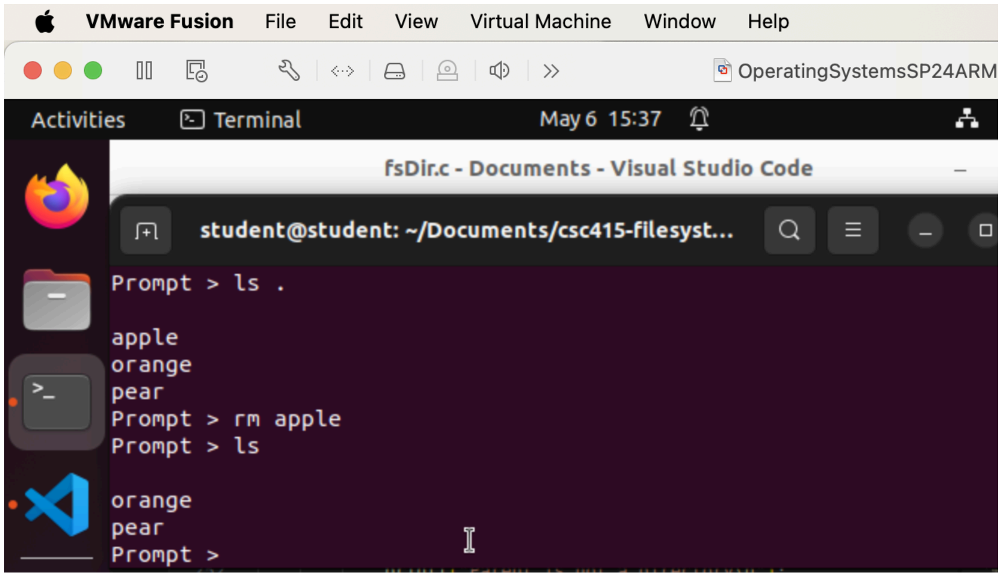
```
Initializing File System with 19531 blocks with a block size of 512
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|--------------------------------|
Prompt > md fruit
Prompt > md veggie
Prompt > md car
Prompt > ls .

fruit
veggie
car
Prompt >
```

This image shows the cd command and the pwd command:



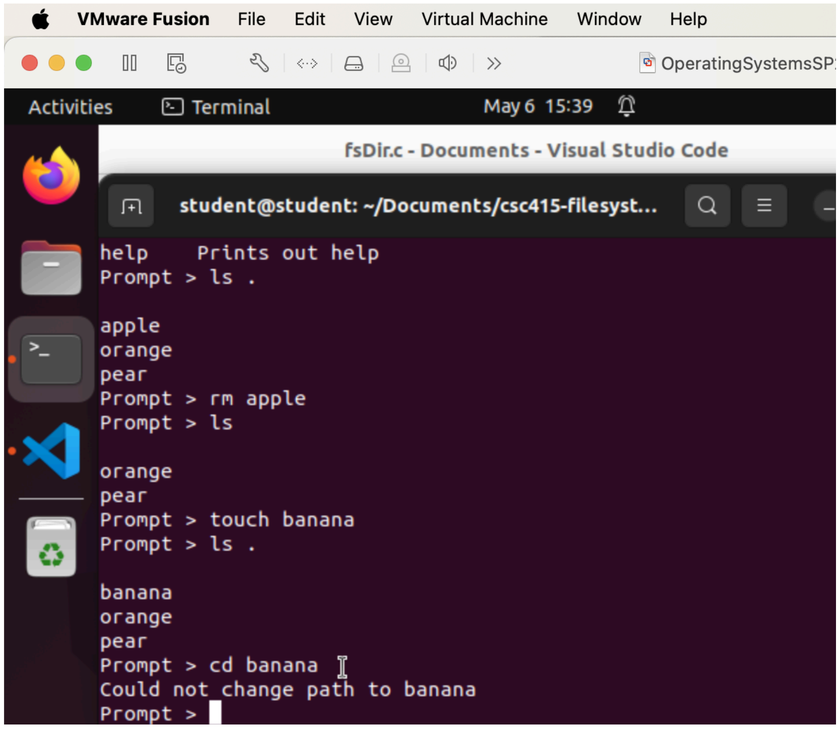
```
Prompt > cd fruit
Prompt > ls .

apple
orange
pear
Prompt > pwd
/fruit/
Prompt >
```

This image shows the rm command on a directory :



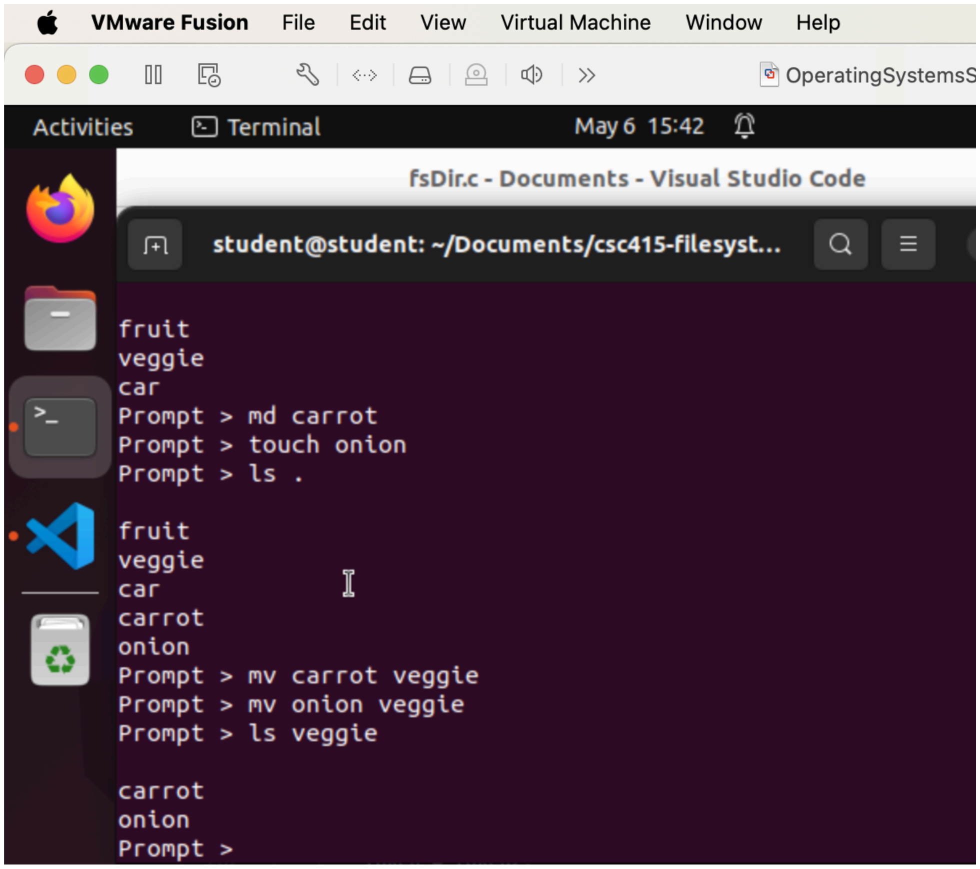
```
Prompt > ls .

apple
orange
pear
Prompt > rm apple
Prompt > ls

orange
pear
Prompt >
```

This image shows the touch command creating a file, and we prove that it's not a directory by trying to cd to it.



```
help     Prints out help
Prompt > ls .

apple
orange
pear
Prompt > rm apple
Prompt > ls

orange
pear
Prompt > touch banana
Prompt > ls .

banana
orange
pear
Prompt > cd banana
Could not change path to banana
Prompt >
```

This shows image shows the mv command on file onion and directory carrot:



```
fruit
veggie
car
Prompt > md carrot
Prompt > touch onion
Prompt > ls .

fruit
veggie
car
carrot
onion
Prompt > mv carrot veggie
Prompt > mv onion veggie
Prompt > ls veggie

carrot
onion
Prompt >
```