Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

Teamname: Tryhards

Github Name: csc415-filesystem-Amandeep-Singh-24

CSC 415 Operating Systems

Name	Id	Github
Amandeep Singh	921287533	Amandeep-Singh-24
Akshat Sohal	917815046	sohal786
Conrad Choi	911679059	ChoiConrad
Rene Antoun	922654834	reneantoun

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal CSC 415 Operating Systems
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

File Systems Writeup Milestones 1-3

Description:

This assignment is focused on developing a fully-functional file system in C. This file system consists of three main phases, one to format the volume, implementing directory-based functions, and implementing the ability to handle file operations.

Approach/What we did:

M1:

- The purpose of Milestone 1, is to format the volume in order to set up a basic file system on a storage level. This is completed by implementing methods of initializing the volume control block, managing free space, and setting up the root directory.
- When forming our group we initially discussed what kind of freespace system we'd like to use. The most basic/simple approach to us was implementing a bitmap, as we all seemed to have a clear understanding through the lectures in class.
    - However upon reviewing further we were most interested in the FAT Table approach, this notion of using a table like format that uses a linked list.
        - After reviewing our approach and the type of freespace system we wanted to use, we began to plan our initial tasks. We began by appointing everyone to a certain task such as initializing the vcb, the freespace, and the root directory.
            - We discussed the type of structs which each file/function would need and continued to alter this based on the feedback given from the professor's initial assignments.
- After dividing up the tasks we began to implement the files, focusing on the vcb we followed the professor's rule of implementing a magic number, a way to show whether or not the vcb has already been initialized.
    - Within the vcb block we decided to initialize values of magicNumber, volumeName, fileSystemType, blockSize, the starting block, tableSize, the rootDirectory start block, along with other helpful fields.
- We took advice from the professor's input regarding our initial setup:
    - Our feedback was- "How do you know if the volume needs to be initialized or not? Missing signature. How do you know where your Root directory is located (missing from VCB) Where is the file located? - missing the starting block number for the file in the Dir_entry"
        - We made sure to keep these comments in mind while implementing the root directory and Volume control block. We spent considerable time dissecting the structure and functionalities, such as the initialization of the file system (initFileSystem). We discussed the volume_control_block, understanding its pivotal role in our file system structure. Our in-depth

analysis covered memory allocation, error handling, and even the interaction with physical blocks of storage through LBAread and

- LBAwrite functions. Recognizing the importance of proper memory management, we reviewed the memory allocation for the volume control block.

- fsInit
    - To begin with, since initFileSystem is the main driver of the assignment, it makes function calls to the different files, freespace and directory.
        - Before any logic implementation truly begins we wanted to format our files appropriately allowing for code to be much easier to find and read, thus we implemented .h files to each respective file we made in order to allow for clarity.
    - When approaching the fsInit.c file, we new the file system would have to recognize if memory has already been mallocd and would need to determine whether or not the FileSystem would need to be initialized every time, thus by reading the vcb and checking whether or not the magicNumber exists is the unique indication of telling whether the vcb has been initialized or not.
        - When it isn't initialized, thus meaning the magicNumber does not exist on the vcb, the vcb needs to be populated with its respective struct values. Thus the vcb includes the magicNumber, name of the volume, fileSystemType, the blockSize, startBlock, tableSize, rootdirStart, lastAllocatedBlock, freeBlockCcount, firstFreeBlock, all value which are crucial in understanding the format of our fileSystemType.
    - After the vcb has been initialized, the function then calls initFAT, as within our file system the FAT Table comes after the implementation of the vcb. Within the initFAT function, it creates a linked list of entries as seen in a typical FAT Table, once this is called and written to disk, the vcb is then also written to disk, saving the initialized vcb.
    - Thus after the FAT Table is initalized and written to disk as well, the last aspect left as seen in our file system are directory entries, wherein we initialize values for default entries to all contain which is then initialized and loaded to disk.
- Freespace
    - When approaching our implementation of the FAT Table for freespace we understood the task of allocating the necessary amount of blocks to represent our FAT Table and portray the ability to manipulate those blocks of data.
        - Thus when following the specified rules from the Milestone 1 guidelines, we approached the FAT Table like a bitmap system, which to note was completely incorrect as a lack of understanding of how a FAT Table truly worked.
            - To mention briefly this was fixed as we began to work/prep to work on Milestone 2, where we were implementing it as a bitmap system, however we needed to have indications of a linked list and a way to show the end of specific blocks and such which will be mentioned in our approach to M2.

- Once the freespace is initialized, we attempted to initialize the FAT Table as well, however we found difficulty on showing our results onto disk.
    - We had essentially forgot to implement a function which would update the disk for the fat table and freespace, which we fixed later on in office hours as we began to progress on our project.
- After having freespace initialized along with our FAT, we understood the need to allocate a proper amount of blocks would be needed from this file. When function calls are to be made for directories, it was crucial to make sure the blocks were available to write on to avoid segmentation faults and overriding valid block counts.
    - The same idea goes for allocating free space, we must initialize the free space in our system so the directories could find spots within our file system that are free and would be able to be written upon.
        - Thus by implementing allocateFreeSpace along with initializeFAT we attempted to create our FAT Table onto disk but again failed to see the appropriate values in the hexdump wherein it was simply showing all 0's without any attempts of writing to disk to be seen.
            - Again this is later fixed in milestone 2 which will be mentioned further in the approach.
- Thus with these functions implemented, it allowed for initFileSystem to call these functions and attempt to initialize our disk properly, wherein our fileSystem within milestone 1 acted more like a bitmap than it did like a FAT Table as a result of a lack of understanding of the concept.

- Directory
    - Using input from the professors feedback on our directory struct we decided the necessary elements for our directory entries would be fileName, fileSize, m/a/c time, isDirectory, and location.
        - File name is important when searching for the entry within the directories, such as needed for commands like ls, or even cd. File_size is a field which allows the understanding of how big the file would be, all time structs are helpful for users. Lastly isDirectory and location are crucial in determining whether or not a given Entry is a directory or not, and location allows us to find the exact struct of a specific path/entry which is necessary when writing these entries to disk.

    - Within the directory file, we understood the purpose of having to find the starting block of the root directory and initializing the values of all other directories. By calculating the amount of bytesNeeded we were able to find the size of a file and in order to find the startBlock we would call allocateFreespace and find it through that function call.
        - Note we alter this approach later on as the values passed into startBlock were unsatisfactory, however in terms of writing to the hexdump it was fully implemented.
    - By initializing the current directory along with the parent directory, we had the initial root directory fully implemented, writing to disk without error.

        - Further things to note in our approach of directory, rewatching lectures on zoom of the professor's implementation of initializing the root directory was a big help, it allowed us to understand the purpose and importance of initializing a current parent directory.
            - By implementing current and parent directory, all other directories are initialized with a default value which is called in fsInit.c showing our ability to modify the directory part of the hexdump properly.
- Overall our implementation of milestone 1 and approach to it was structured well to an extent but our lack of understanding of how a FAT Table/System works proved it to be difficult to fully complete this task, however our basic implementation of the vcb and root directory were practical enough to complete this task to an extent, however when creating office hours we understood there was lots of room for improvement which leads into our approach for milestone 2.

M2:

- Milestone 2 involved integrating the fsshell into our file system. Shell code with built in commands was provided to us. All the commands in the shell were switched off so that we are able to build our files without getting errors and working on one function at a time. Our plan was to start with mk_dir first as that involved creating a directory. We first reviewed the lectures and took notes on how professor approached mk_dir.
    - While watching the lecture video, we learned that mk_Dir is going to involve a lot of helper functions parse-path being the most important one. First thing, we need to check if the path given in terminal prompt is relative or absolute path. We then need to tokenize the path and get the last element. But while tokenizing we need check if each of the directories in path are actually directories and they actually exist. For instance: if we do mk_dir ("/home/student/documents/foo"), for mk_Dir to actually work, we need to make sure in this path each of these directories before foo exist. So first we need to have a helper function that locates them. If it exists, we need to load them and search each directory entry to find the one given. Also, we need to make sure "foo" does not exist as it needs to be created.
- We followed the structure code provided in the lecture and tried implementing it according our file system. In order for mk_dir to work we needed access to a lot of variables like the pointer to parent directory, index where directory is found and the last element. As in C language, we cannot return a tuple, we had to create a struct and assign it the above mention input fields. We called this ppiinfo struct.
    - After implementing pseudocode for parse path, we stopped coding and planned how the helper functions that we just named will work.
- We could not figure out the working of helper functions so we went to Office hours. We learned that we need to start with one function at a time. Professor helped us with the load directory function. The professor also recommended to actually return the pointer to entry rather than startblock (location) from our Init Directory functions in Directory.c file which was part of milestone 1. This way it was easier to have access to the directory and all the entries. The startblock could be accesses by getting the location of 0th entry. We have attached the notes of

pseudocode in this approach , we followed the pseudo code along to write the parse path function. After this was done, we went to office hours again and learned that for mk_dir, if we want to make a new directory, we are actually editing its parent directory so we need to access it through disk by reading it (here we implement our readFromDisk helper function). Then after editing the directory, we also need to write it back to the disk (here we implement writetoDisk) function. Then we also implemented the redundant helper functions like IsaDirectory , findEmptyEntry and FindEntryinDir. For FindEmpty Entry entry we just compared the name with empty name and if it was empty, we just used that index to make a new entry. This worked as we made a error condition which not accept an empty name while a directory is being created.

- Once we were done with all these functions, our mk_Dir worked and created a new entry, but we found out our information about directory would disappear from hexdump once we exited. After talking to the professor we found out that we were never loading the root directory to disk. Therefore we created a new function in directory.c which wrote the root directory to disk.
- There were issues with our write and read to disk functions but we talk more about these issues in our issues and resolution part of this report. Now, it was time to make another function work. We chose to do rm_Dir. We first call parsePath to get the parent, index and the last element that we need to delete. Here we make sure that the last element actually exists to be deleted. We first load the directory and then call memset function to remove the directory from its parent. Then call releaseMutipleBlocks function from Freespace to free the blocks. This function also calls FATupdate function to make changes in the FAT.
- After rm_Dir, we also implemented fs_delete which was similar except that function deletes a file rather than a directory. Other than that, majority of code was same for both of these functions. Our next target was to make cd work. For CD command, we first reviewed the shell and found out that we need to implement fs_set as thats the function that sets the current directory and is being called in cd. We already had made cwd and rootDir as global variables according to the lectures.
- In implementing the fs_setcwd function, we first call the ParsePath function to dissect the given path into manageable components. This step is crucial for understanding the structure of the path within our file system. After parsing, we immediately verify whether the parent part of the parsed path represents a directory, using the IsADirectory function. This check ensured that the new path is indeed a directory, as a file system's current working directory can only be set to a directory. If the path is not a directory or if the parsing fails (indicated by a NULL parent or an index of -1), the function returns -1 to signal an error.
- Once the path is confirmed as a valid directory, we proceed to update the currentPath. If path is an absolute path (starting with /), it replaces the currentPath entirely. For relative paths, we first ensured currentPath ends with a slash before appending the new path, respecting the maximum path length limit. After updating currentPath, we load the directory contents into currentDir to reflect the change in the working directory.
- Although this function made cd work, we still had a issue with the absolute and relative path. We were always getting the relative path after this function. We went to office hours regarding this issue and found out we need to make a global current path variable which stores the absolute path. However, we were not able to implement a function which would clean our path. The

approach was talked about in the lecture video but due to fixing other issues and lack of team resources, we were not able to implement it.

- The easiest function to implement was fs_getcwd where just did a error check for path being null and copied the current path to buffer. After this , even though we were constantly fixing our earlier functions because of errors in our Milestone 1 freespace code, we kept on moving on further. Next step was to work on the ls command. This involved implementing fs_stat, fs_opendir and isFile function. We already implemented isFile function before as it was similar to isDir function. The implementation for these two was straightforward as it just involved passing the directory entry structure as argument and accessing its isDirectory input field.
- In the fs_stat function, we began by ensuring that both the path and the buf (the buffer for storing file statistics) are not NULL, immediately returning -1 if either is, to prevent any null pointer errors. Next, we called ParsePath function.. If the path parsing fails, indicated by a NULL parent or an index of -1, the function returns -1, signaling an error due to either a parsing issue or a non-existent path.
- Upon successful parsing, we retrieve the DirectoryEntry corresponding to the path from ppi.parent[ppi.index]. This entry contains the metadata for the file or directory at the given path. We then proceed to populate the fs_stat structure pointed to by buf with this metadata. We set the file size (st_size) directly from the entry's file_size. We also retrieved and set the access, modification, and creation times (represented by st_accesstime, st_modtime, and st_createtime, respectively) from the corresponding fields of the DirectoryEntry. This function was easier to implement as we had implement mk_Dir before. We also had help from the professor in office hours to see if we were on the right track.
- Next, we implemented the open function. In implementing the fs_opendir function, our objective was to open a directory specified by pathname and create a stream to read its contents.

First we looked at the mfs.h file as we had to uncomment some stuff there. We also filled the struct according to our file system

```
 {
   /*****TO DO:  Fill in this structure with what your open/read directory needs  *****/
   unsigned short  d_reclen;       /* length of this record */
   unsigned short  dirEntryPosition;   /* which directory entry position, like file pos */
   DirectoryEntry *  directory;          /* Pointer to the loaded directory you want to iterate */
   struct fs_diriteminfo * di;     /* Pointer to the structure you return from read */
   } fdDir;
```

- We then checked if the given pathname was indeed a directory using the fs_isDir function.Then call parse path.
- We allocated memory for a fdDir structure using malloc. This structure was intended to hold the directory stream, encompassing all necessary information to read through the directory's contents. In case of a memory allocation failure, we logged an error about the allocation issue and returned NULL, ensuring robust handling of resource limitations.

With the fdDir structure in place, we loaded the directory data into its directory field using loadDirectory(ppi.parent[ppi.index]). This loaded the actual directory contents into the structure, based on the parsed path.We initialized dirEntryPosition to 0, setting the starting point for reading directory entries. The d_reclen was set to MAXDIRENTRIES. Finally, we returned the directory stream object (dirStream), ready for use in reading the directory's contents.

- After implementing this function, our ls was working. But while creating a directory for instance, say in foo (the directory that we tried to make was bar), we got both directories printed out as foo bar . After discussing with the professor agreed that this was because our cd (fs_set) was not doing the job to actually cd into foo. However, after doing print statements we realized that even cd was implemented okay. The problem actually stemmed from where we were writing on the disk. This all went back to allocateBlock function which was being called from the freespace.c. We were unfortunately not able to figure out the root cause of this issue. We have talked more about this in issues and resolution.

M3:

B_open function:

- The purpose of this function is to open the specified file. When given the file name, and the flags.
    - Process: it will initialize a b_io_fd variable called returnfd, Then it will start the first bound check.
    - The first Bound Check: the first bound check will verify if a start up variable is 0 then it will run the b_init function. Should it fail it won't run the init function.
- We then assign the returnfd variable with b_getFCB which will just get a free element number however, if it is -1 that would mean that the FCB is full.
    - Second Bound Check: the second bound check will make sure that -1 is not returned indicating everything is in use.
    - Third Bound Check: This bound check makes sure that there is no invalid symbols being used for a file name.
- We check the flags and make sure not to give the provided flags more permissions than it is allowed.
- Only when certain flags and a appropriate file size is given then it will create a file calling the mk_file function .
    - Bound Check: we make sure the file was created properly if not then
- If passed all the checks we then allocate and initialize the buffer.
    - Bound Check: this bound check makes sure that the buffer allocation didn't fail.
- Then we initialize fcbArray.
    - We then bound check again if the the strdup failed. And if it does we free the buffer.
- Finally we retrieve the file statistics
    - Bound Check: if the file statistic doesn't return 0 then we clear and return an error.
- Assign the file size then return returnFd.
- As much as the bound checks this could use mutex functions so it can handle multithreading.

Mk_file Function:

    - Purpose: the purpose of this function is to make a file.
- This function takes in a file name

    - First bound check: this bound check makes sure that everything isn't null.
    - Second Bound Check: this makes sure that the current working directory isn't set to null or else it will notify the user and it will set current working directory as the root directory.
        - Sub Bound Check: This makes sure that the root directory is not null, or else it will pass an error message.
- It will then parse the path and assign it to the parsed path structure we call ppi.
    - Bound Check: this bound check makes sure that there was no error when parsing the path.
- Then we need to check if the file exists, if it exists there is no point to create another one.
- Next, we need to check if the parent directory is not null.
- We then create a new file entry and initialize the details for the file.
- We find an empty entry in the parent directory while making sure there is space.
- We then update the parent directory then write it onto disk.
    - Bound Check: this check is to make sure that there the update to the disk didn't fail or else we send an error code.
- Return pass and inform the user that the file was created successfully.
- Something I would do is test how it can handle nested directories. I would add more security tests to ensure maximum security.

B_close

- Purpose: the purpose of this function is to close a file that was opened.
    - The function starts off by checking that it took in something valid we called fd which will make sure that fd is valid making sure that it is not less than zero or greater than the max.
    - Second Bound Check: it will make sure that fd has been used. We do this because we want to make sure not to release NULL.
    - Third Bound Check: this bound check makes sure that the file name is freed. Again, we check that it isn't NULL. If it wasn't NULL we free the file name and reset to NULL.
- We reset the elements of the File control block when it reaches the end.
- Return zero if everything has been done.
- To change this I would add Mutex for multithreading and security because if two different threads run the close it might close multiple times, and security wise so no other outside access.

b_getFCB:

- Process: this is to return the FCB.
- I would change this function by adding mutex locking, and make sure that no one else can gain access to the function.
- The get function is not thread safe because if multiple threads call the function at once it could return the same element. Mutexs would fix this.

B_seek:

Purpose: the function is to change the file position.

- Some changes for the seek function is to check everything the function takes in, this is to end the function before any initialization even happens.

B_write:

- Purpose: This is writes the data into a io system

// pass NULL if root

DE * parent

⇒ Initialize Root Directory :-

int init Root Directory (int default entries,))

int bytes_Needed = defaultEntries * size of (DE)
int blocks_Needed = (bytesNeeded + (blocksize - 1)) / blocksize

bytes Needed = blocks Needed * block size.
int actual DirEntries = bytes Needed / sizeof (DE)
DE * dir = malloc (bytes Needed)
int StartBlock = alloc Blocks (blocks Needed)

for (int i=0 ; i < actual Dir Entries ; i++)
{
dir[i].name[0] = '\0' ; set name to Null
dir[i].used = 0;
}

strcpy (dir[0].name, ".") ;
dir[0].size = actualDirectoryEntries * sizeof(DE)
dir[0].loc = StartBlock ;
dir[0].extent.start = e.start
dir[0] ;
dir[0].isDir = 1
time_t t;
dir[0].create = t
dir[0].modified = t
dir[0].accessed = t

use if parent != null
DE * p = & dir[0];
strcpy (dir[1].name, "..") ;
dir[1].size = p → size
dir[1].loc = p → loc
dir[1].isDir = * → isDir
dir[1].create = p → create

dir[i].modified = p -> modified
dir[i].accessed = p -> accessed.

└ DA write (dir, blocksNeeded, start block)
free (dir)
return (start block);

How to implement FAT table for file system.

* chain of blocks
* init directory calls functions

how we set unused, isDir

dir[i].name[0] = '\0'; // Null term name

// Make sure we did isDir
dir[0].isDir = 1;

if (parent == NULL)

else DE *p = parent; the we p -> size preference

make Dir -> has to know parent and position

=> the structures in mfs.h only are for fsopen, fsread

=> fs_getcwd(char *pathname, size_t size)

Use strncpy
(make sure path is not null)

=> fs_delete(~~delete~~ free blocks and mark it unused)
↳ Make sure its a dir/file

MKdir (pathname)

mkdir ("foo") Relative path to cwd

("/foo") Absolute to root

mkdir ("/home/student/documents/foo") → must not exist

→ must exist and must be dir

→ Load Root directory
→ Search each directory entry enroute
for one named "home"
if found - is it a directory?
→ load home
similarly load student then documents
if foo found { no }

Same function for delete, remove dir etc.

mkdir (path) {

parse path (path, DE * * parent,
int DE index)
char * last elem - 1 index

if invalid get out
if DE index != -1 get out

for mkdir

DE index = Find DE (parent)

Block # = init Dir (Default # entries, parent)

Load Dir (block)

Strcpy (parent [DE index]. name,
last element)

DE * New Dir

parent [DE index]. size = new dir [0].
size

Rather than Returning just block #,
return DE pointer.
to get block # look at 0 dE
and the location will give start

Parse path

struct
- → # parent
- → index into the parent of the last element
- → name last element
- → is path valid

Actual return values

* For rest, use struct

```
int parsepath(char * parsepath, pp
                    Imfo * ppi)
{
typedef struct ppinfo
    {
        DE * parent;
        char * last element;
        int index; }

Parse Path (char * path, pp info * ppi)
        {
            if (path == = Null)
                return -1;
globals
rootdir         if (ppi == = Null)
cwd                 return -1;
            DE * start dir;
            if (path [0] = = '/')
                Stardir = root dir
            else
                Stard dir = cwd
```

```
parent = start dirg
token1 = strtok_r (path, "/",
                   & save ptr)

if (token1 == = NULL)
    {
if ( strcmp ( path, "/") == = 0 )
    {
ppi → parent = parent
ppi → index = -1
ppi → last element = NULL
  return 0; }
Return (-1)
        }
while ( token1 != = NULL)
        {
        index = Find Enty in Dir (parent,
                                  token1)
token2 = strtok - r (Null, "/", save ptr);
    if  token2 = = = NULL
        {
        ppi → parent = parent
        ppi → index = index
        ppi → last element = Str dup (token1)
        return 0;
        }
if (index = = -1)
        return (-2);

if (! is A Directory (& parent
                      [index]))
```

find → 0 to $\frac{\text{size of '.' dirents}}{\text{\# of directories}}$

return (-2);

DE * temp = Load Dir (& (parent[index]))

if temp == NULL

return -3

if (parent != start dir

free (parent);

parent = temp

toRem1 = toRem2

fs.Delete

if(index == -1) exit

if(GetType(&p[Index]) != File
exit

Release Blocks From DE(&p[index])

Is Dir

if (index == -1) exit

Return get type(&p[index]

rmdir

if(index == -1) exit

if(GetType(&p[index]) != Dir
exit

is File

Return (GetType() == file)

Get type
type = p[index]
De * de
return(de => type)

- I would add mutexs in writing for safety and to be thread safe incase of a multithreaded environment.
- next, would be ensuring the location of writing would be written into a null segment.
- I would change this by checking if a file is available for writing.

Issues and Resolutions:

- **free space : didnt fully understand FAT. Created a bitmap , then went to OH and learned its actually a linked list. Created a struct to track free blocks.**
    - The resolution to this was seen through office hours, we were explained that a FAT Table isn't supposed to act like a bitmap, hence defeating the purpose of a FAT Table. Showing an initialized hexdump with the ability to right to the blocks shows a bitmap system, however a FAT Table should show a linked list of values, one pointing to the next for the entirety of the FAT Table size. Thus we needed to review and alter our approach to freespace, we then implemented a version of free space which held hex values all the way to our table size of 306, containing a linked list. Each element pointing to the next, our member Conrad had this task to implement and complete, wherein our hex dump visually atleast when being initialized showed the right values and the proper implementation of a hex dump.

-Loading Root Directory:

- When we created the mk_Dir function and implemented all the helper functions which were needed to create a directory. We were able to create a new directory but when we would exit the shell, all our entry data from hexdump will disappear. We went to office hours for this issue. After talking to the professor we found out that we were never loading the root directory. So therefore we created a new function in directory.c called loadRootDirectotry and we called that function while initialzing our file system fsinit.c.

- **Blocks Needed calculation wrong:**
    - When calculating the amount of blocksNeeded we had error in our logic which didn't include for rounding errors along with having the parentheses in the wrong spot. Thus when we would calculate the amount of blocksNeeded this led to a value which wasn't expected and thus leading to errors within our program when it came to writing to disk and so on. Thus by going to office hours the professor explained the importance of each element, the size of the directory + blockSize -1, must be calculated first, thus when dividing it all by blovkSize it yielded the appropriate value.
        - **int blocksToWrite = (dirSize + blockSize - 1) / blockSize;**

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal CSC 415 Operating Systems
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

- **paramters issues and conflicting types:**

We implemented bunch of print statements in mk_Dir during office hours with the professor and going through those we found out that the block number where the data was being written were just out of bounds. Then the professor helped us run Val Grind to see where the issue was and we found out that the issue stemmed from out writeToDisk helper function. On checking that function in mfs.c file, the professor pointed out we were using unnecessary parameters and actually callin the function with wrong parameters which was leading to out of bound number for start block.
Below is the code snippet where the professor pointed out the issue.

int writeDirectoryToDisk(DirectoryEntry *dir/*, uint32_t startBlock, int numEntries*/)

We had another issue with conflicting data types of parameters as we were getting confused between pointers and locations (where to use & and where to use just the pointer name). This was fixed by reviewing the code. Even after this our block numbers were out of bound , but that issue was in freespace function where in for loop the value of i was set to 2 but it should have been 402.
Following is the code snippet where the change was made:

```
uint32_t findNextFreeBlock() {
   if (fatTable == NULL) {
      fprintf(stderr, "findNextFreeBlock: fatTable is not initialized.\n");
      return END_OF_FILE;
   }
   for (uint32_t i = 402; i < 19531; i++) {
      if (fatTable[i].status == FREEBLOCK) {
         return i; // Found a free block.
      }
   }
   return END_OF_FILE; // No free blo
```

- **Compilation Error due to Header Issues:**
    - When attempting to implement M2, we ran into an error with compilation, although on vscode there were no apparent errors it had something to do with declarations and how it was affecting our compilation. In order to fix this issue, I first had to reformat the error in which our functions were created in mfs.c, by organizing them based off function calls, thus if a specific function call parse, parse must be implemented before it and so on. Furthermore when it came to the header issue, the error stemmed from multiple .h files containing the same include statements to other files, upon altering it and cleaning up the proper include values we were final able to make it run printing the shell properly.

- **Directory Entry location issue:**

  After fixing the freespace and the parameters for writeToDisk, we still were not getting the right numbers for the blocks where the directory from mk_Dir should be written. We found out in office hours that there was an issue with our Directory.c file. In the function Init Directory, while initiating the directory entries other than root and parent , we set the location first to 0 but then one of our team members added a for loop that looped over all the Directory Entries and set their location to startblock. We commented out that code block and finally were able to see the right number for startblock and blocks needed for the directory entry.

-Accessing memory blocks which didnt make sense: used val grind in OH , to find out the issue , we were not using write parameters for writetoDisk and readtoDisk function. After Oh also fixed free space and set it up to find free blocks after block 402 as those blocks were occupied by vcb and FAT

- When going to OH and showing the professor the shell commands such as mkdir, wherein we were having issues as to where it was being written to and so on. Through implementing debugging statements we noticed that the startBlock was a ridiculously high number, a number out of scope of the file system as it only contains 19531 blocks. After taking the professors advice we noticed that after initFileSystem is called the startBlock number was affected heavily, thus the issue was within fsInit.c or maybe in its call to other functions. However we noticed that we were always freeing the vcb regardless of if user was exiting the system or not, it was counterintuitive, we were initalizing the vcb just to free all of the data. Thus the startBlock numbers were very far off and allowed us to get back on track to an extent. Furthermore we then had issues with our parameters for writetoDisk and readToDisk functions, by using val grind the professor showed us how to track down potential major issues to which one was incorrect. Thus as a result of fixing these issues through vrun we were finally able to see the appropriate directories finally being printed to disk.

-mk_dir: directories kept being made at wrong addresses in memory

- **Fat update:**
  - When it came to updating the values based on the FAT Table we ran into many issues, we were not updating the FAT Table properly as the parameters being passed in were on improper values. Within our initFat function the professor helped us implement the proper blockCounts and helped us find errors in our startBlocks. By using debugging print statements we were able to see that our startBlock was not being calculated properly, and that the size of our Fat Table wasn't initially correct. Through the professors debugging we were able to fix the FatUpdate() function to write to disk at the appropriate blocks and for the appropriate lengths.

B_open issues and resolutions:

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

An issue I had was handling the Flags. I (Conrad Choi) had to research how to handle the flags. I had to relearn how the bit wise comparison works. The actual managing of the flags and setting up the conditional logic. For security reasons it is important to know the different conditional logic and how it works. I had to learn the importance for bitwise comparison for efficient storage and it allows checking for multiple conditions in one operator.

Mk_file issues and resolution:

A large issue was what happens when mk_file hits a null parent directory. I assumed that it was an error however, as everything was working, I learned to properly handle the parentless directory.

General Milestone 3 issues

I learned the importance of defensive programming not only was it important for maintaining functionality there is an importance for safety reasons and risks to data leaks.

Compilation (Make/Make Run):

```
parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/csc415-filesystem-Amandeep-Sin
gh-24$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o freespace.o freespace.c -g -I.
gcc -c -o directory.o directory.c -g -I.
directory.c: In function ‘loadRootDirectory’:
directory.c:56:9: warning: assignment to ‘DirectoryEntry *’ from incompatible po
inter type ‘char *’ [-Wincompatible-pointer-types]
   56 |     cwd = strdup("/");  // Ensure cwd is dynamically allocated
      |         ^
gcc -c -o mfs.o mfs.c -g -I.
gcc -o fsshell fsshell.o fsInit.o freespace.o directory.o mfs.o fsLowM1.o -g -I.
 -lm -l readline -l pthread
parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/csc415-filesystem-Amandeep-Sin
```

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786


```
parallels@ubuntu-linux-22-04-02-desktop:~/Desktop/csc415-filesystem-Amandeep-Sin
gh-24$ 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
initFat: Blocksize = 512 and startblock is 1
initFAT: Memory allocation for FAT table successful
initFAT: Completed successfully
initFAT: Initialization complete. Number of FAT blocks: 306
FATupdate: Blocksize = 512 and startblock is 1
FAT successfully updated on disk. Written 306 blocks.
```

Analysis:

Hexdump:

VCB Dump:

```
000200: 55 AA 00 00 00 00 00 00  4D 79 56 6F 6C 75 6D 65 | U♦......MyVolume
000210: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000220: 00 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  46 41 54 33 32 00 00 00 | ........FAT32...
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 02 00 00 00 00 00 00 | ................
000330: 01 00 00 00 00 00 00 00  32 01 00 00 00 00 00 00 | ........2.......
000340: 92 01 00 00 00 00 00 00  91 01 00 00 00 00 00 00 | ♦.......♦.......
000350: AF 4A 00 00 00 00 00 00  9D 01 00 00 00 00 00 00 | ♦J......♦.......
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 | ................
```

- 55 AA 00 00 00 00 00 00
    - When initializing the VCB we knew we needed a specific signature, a “magic number”, and since our file system is based on a FAT system, we used what we found to be the industry standard of 0xAA55. Note, the declaration of magicNumber is of size uint64_t, which is 8 bytes, thus the address is the whole line. Thus within the VCB we add the unique signature to signify the start of it, thus in little endian notation we see 55AA, which equates to our unique signature.
- 4D 79 56 6F 6C 75 6D 65
    - Next is the volume_name, which is an array of characters which can be up to 256 bytes long. Thus using strcpy, we add volume_name to the VCB and note this is of type char and an array of 256 bytes. Translating the hex values to text, we can see the output in ASCII as “MyVolume” followed by a large buffer of 0’s. This is due to the size of the character array, thus theres a gap of 256 bytes between volume name and fileSystemType.
- 46 41 54 33 32 00 00 00
    - Within the struct after volume_name is the fileSystemType, which is also an array of characters, however, it is limited to 32 bytes. Thus when we strcpy the value “FAT32” to the VCB, we can see the hex be translated to ASCII portraying FAT32, and has a buffer of 0’s after it due to the size of the array, 32 bytes.
- 00 02 00 00 00 00 00 00
    - The next field is block_size, blockSize is already given from the function parameter and is 512 bytes. Thus when adding this to the VCB, we can see the value of 02000 and again the size is 8 bytes, and moreover is represented in little-endian format. Thus these 8 bytes can be adjusted as being 0x0200, which translates to the block size, which again is 512 bytes.
- 01 00 00 00 00 00 00 00
    - Next is the starting block, which simply holds a value of 1. Assuming little-endian format again the hex value of 1 still translates to 1 signifying the beginning of the block.
- 32 01 00 00 00 00 00 00
    - Now table size is evaluated by calculating the number of clusters and since each entry in the fat table is 4 bytes, we multiply it by 4, eventually leading to the value of 306, thus the FAT TABLE has a decimal size of 20. When translating this value into decimal taking it from little-endian this equates to 20, showing that the VCB holds the proper value.
- 92 01 00 00 00 00 00 00
    - The next value is the first_free_block which is essentially an offset of the root_directory_start_block by 32 bytes. This evaluates to 402, the start of our rootdir.
- 91 01 00 00 00 00 00 0
    - Last allocated block, one before rootdir, thus value of 401
- AF 4A 00 00 00 00 00
    - This is the value of 19119, the amount of free blocks left 19531 - 401.
    - 9D 01 00 00 00 00 00 00
    - Lastly is next free block after root dir which is 413, thus as rootDir is 402, 10 allocation spaces, thus yielding 413.

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

FAT TABLE:

```
000400: 01 00 00 00 FF FF FF FF  00 00 00 00 02 00 00 00 | ....ÿÿÿÿ........
000410: 00 00 00 00 03 00 00 00  00 00 00 00 04 00 00 00 | ................
000420: 00 00 00 00 05 00 00 00  00 00 00 00 06 00 00 00 | ................
000430: 00 00 00 00 07 00 00 00  00 00 00 00 08 00 00 00 | ................
000440: 00 00 00 00 09 00 00 00  00 00 00 00 0A 00 00 00 | ................
000450: 00 00 00 00 0B 00 00 00  00 00 00 00 0C 00 00 00 | ................
000460: 00 00 00 00 0D 00 00 00  00 00 00 00 0E 00 00 00 | ................
000470: 00 00 00 00 0F 00 00 00  00 00 00 00 10 00 00 00 | ................
000480: 00 00 00 00 11 00 00 00  00 00 00 00 12 00 00 00 | ................
000490: 00 00 00 00 13 00 00 00  00 00 00 00 14 00 00 00 | ................
0004A0: 00 00 00 00 15 00 00 00  00 00 00 00 16 00 00 00 | ................
0004B0: 00 00 00 00 17 00 00 00  00 00 00 00 18 00 00 00 | ................
0004C0: 00 00 00 00 19 00 00 00  00 00 00 00 1A 00 00 00 | ................
0004D0: 00 00 00 00 1B 00 00 00  00 00 00 00 1C 00 00 00 | ................
0004E0: 00 00 00 00 1D 00 00 00  00 00 00 00 1E 00 00 00 | ................
0004F0: 00 00 00 00 1F 00 00 00  00 00 00 00 20 00 00 00 | ........... ...

000500: 00 00 00 00 21 00 00 00  00 00 00 00 22 00 00 00 | ....!.......".....
000510: 00 00 00 00 23 00 00 00  00 00 00 00 24 00 00 00 | ....#.......$...
000520: 00 00 00 00 25 00 00 00  00 00 00 00 26 00 00 00 | ....%.......&...
000530: 00 00 00 00 27 00 00 00  00 00 00 00 28 00 00 00 | ....'.......(...
000540: 00 00 00 00 29 00 00 00  00 00 00 00 2A 00 00 00 | ....).......*...
000550: 00 00 00 00 2B 00 00 00  00 00 00 00 2C 00 00 00 | ....+.......,...
000560: 00 00 00 00 2D 00 00 00  00 00 00 00 2E 00 00 00 | ....-...........
000570: 00 00 00 00 2F 00 00 00  00 00 00 00 30 00 00 00 | ..../.......0...
000580: 00 00 00 00 31 00 00 00  00 00 00 00 32 00 00 00 | ....1.......2...
000590: 00 00 00 00 33 00 00 00  00 00 00 00 34 00 00 00 | ....3.......4...
0005A0: 00 00 00 00 35 00 00 00  00 00 00 00 36 00 00 00 | ....5.......6...
0005B0: 00 00 00 00 37 00 00 00  00 00 00 00 38 00 00 00 | ....7.......8...
0005C0: 00 00 00 00 39 00 00 00  00 00 00 00 3A 00 00 00 | ....9.......:...
0005D0: 00 00 00 00 3B 00 00 00  00 00 00 00 3C 00 00 00 | ....;.......<...
0005E0: 00 00 00 00 3D 00 00 00  00 00 00 00 3E 00 00 00 | ....=.......>...
0005F0: 00 00 00 00 3F 00 00 00  00 00 00 00 40 00 00 00 | ....?.......@...
```

- Our hexdump for the FAT Table shows proper allocation and initalization of the FAT Table to act like a proper linked list.

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

RootDirectory:

- Within our root directory we see the presence of the current and parent directory through the 2E characters, 2E = (“.”), thus the presence of 2E2E = (“..”) showing a proper initialization of our root directory.
- 6F 69 65, the repetition of this 3 times is the values for atime, ctime, and mtime, which all have the same value.
- 01 represents that it is a directory.
- The value of 92 01, represents the location which this is located at which again is block number 402.

Amandeep Singh, Conrad Choi, Rene Antoun, Akshat Sohal
ID:921287533, 911679059, 922654834, 917815046
Github: Amandeep-Singh-24, ChoiConrad, reneantoun, sohal786

Shell Run:

```
Prompt > touch test.txt
line 260
[ParsePath] Called with path: 'test.txt'
im in parse
Pathcopy: 0x55bbae5c3650
Im in the else statement
Before ppi stuff
[FindEntryInDir] Searching for token: 'test.txt'
[FindEntryInDir] parent or token is NULL
[ParsePath] Path is root ('/'), setting ppi->parent: (nil), ppi->index: -1, ppi->lastElement: 0x55bbae5c3670
[IS FILE] test.txt does not exist or is not a valid entry
mk_file: Current Working Directory is not set. Defaulting to root directory.
[ParsePath] Called with path: 'test.txt'
im in parse
Pathcopy: 0x55bbae5c3650
Im in the else statement
Before ppi stuff
[FindEntryInDir] Searching for token: 'test.txt'
[FindEntryInDir] Token not found
[ParsePath] Path is root ('/'), setting ppi->parent: 0x55bbae5a0930, ppi->index: -1, ppi->lastElement: 0x55bbae5c3690
[findEmptyEntry] Called
[findEmptyEntry] Found empty entry at index: 3
[writeDirectoryToDisk] Called with startBlock: 402, numEntries: 16
[writeDirectoryToDisk] BlockSize: 512, dirSize: 4864, blocksToWrite: 10
mk_file: File 'test.txt' created successfully
[ParsePath] Called with path: 'test.txt'
im in parse
Pathcopy: 0x55bbae5c36b0
Im in the else statement
Before ppi stuff
[FindEntryInDir] Searching for token: 'test.txt'
[FindEntryInDir] Found token at index: 3
[ParsePath] Path is root ('/'), setting ppi->parent: 0x55bbae5a0930, ppi->index: 3, ppi->lastElement: 0x55bbae5c36d0
line 262
success
Prompt >
```

```
| pwd                 |    ON    |
| touch               |    OFF   |
| cat                 |    OFF   |
| rm                  |    ON    |
| cp                  |    OFF   |
| mv                  |    OFF   |
| cp2fs               |    OFF   |
| cp2l                |    OFF   |
|---------------------------------|
Prompt > md /foo
[MKDIR] Called with pathname: '/foo'
[ParsePath] Called with path: '/foo'
im in parse
Pathcopy: 0xaaaaccf2adc0
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'foo'
[FindEntryInDir] Token not found
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: -1, ppi->lastElement: 0xaaaaccf2ade0
[MKDIR] ParsePath result: 0
Count: 1

Before initDirfindNextFreeBlock: Checking FAT entries... (vcb->table_size: 306)
FAT Entry 0: status = 1, nextBlock = 4294967295
FAT Entry 1: status = 0, nextBlock = 2
FAT Entry 2: status = 0, nextBlock = 3
FAT Entry 3: status = 0, nextBlock = 4
FAT Entry 4: status = 0, nextBlock = 5
FAT Entry 5: status = 0, nextBlock = 6
FAT Entry 6: status = 0, nextBlock = 7
FAT Entry 7: status = 0, nextBlock = 8
FAT Entry 8: status = 0, nextBlock = 9
FAT Entry 9: status = 0, nextBlock = 10
findNextFreeBlock: Checking FAT entries... (vcb->table_size: 306)
FAT Entry 0: status = 1, nextBlock = 4294967295
```

```
FAT Entry 3: status = 0, nextBlock = 4
FAT Entry 4: status = 0, nextBlock = 5
FAT Entry 5: status = 0, nextBlock = 6
FAT Entry 6: status = 0, nextBlock = 7
FAT Entry 7: status = 0, nextBlock = 8
FAT Entry 8: status = 0, nextBlock = 9
FAT Entry 9: status = 0, nextBlock = 10
FATupdate: Blocksize = 512 and startblock is 1
FAT successfully updated on disk. Written 306 blocks.
Startblock: 412

After initDir[MKDIR] Updated new directory name to: 'bar'
[findEmptyEntry] Called
[findEmptyEntry] Found empty entry at index: 3
[MKDIR] New index in parent directory: 3
[MKDIR] Writing updated parent directory to disk
[writeDirectoryToDisk] Called with startBlock: 402, numEntries: 16
[writeDirectoryToDisk] BlockSize: 512, dirSize: 4864, blocksToWrite: 10
[MKDIR] Directory 'bar' created successfully.
Prompt > cd /bar
[ParsePath] Called with path: '/bar'
im in parse
Pathcopy: 0xaaaaccf2aef0
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'bar'
[FindEntryInDir] Found token at index: 3
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: 3, ppi->lastElement: 0xaaaaccf2af10
[IsADirectory] Called
[IsADirectory] isDirectory value: 1
INSIDE CWD
Success
Prompt > pwd
bar
Prompt > cd /../
```

```
FATupdate: Blocksize = 512 and startblock is 1
FAT successfully updated on disk. Written 306 blocks.
Startblock: 407

After initDir[MKDIR] Updated new directory name to: 'foo'
[findEmptyEntry] Called
[findEmptyEntry] Found empty entry at index: 2
[MKDIR] New index in parent directory: 2
[MKDIR] Writing updated parent directory to disk
[writeDirectoryToDisk] Called with startBlock: 402, numEntries: 16
[writeDirectoryToDisk] BlockSize: 512, dirSize: 4864, blocksToWrite: 10
[MKDIR] Directory 'foo' created successfully.
Prompt > md /bar
[MKDIR] Called with pathname: '/bar'
[ParsePath] Called with path: '/bar'
im in parse
Pathcopy: 0xaaaaccf2ae70
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'bar'
[FindEntryInDir] Token not found
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: -1, ppi->lastElement: 0xaaaaccf2ae90
[MKDIR] ParsePath result: 0
Count: 1
```

```
..
Prompt > cd /foo
[ParsePath] Called with path: '/foo'
im in parse
Pathcopy: 0xaaaaccf4a8e0
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'foo'
[FindEntryInDir] Found token at index: 2
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: 2, ppi->lastElement: 0xaaaaccf4a900
[IsADirectory] Called
[IsADirectory] isDirectory value: 1
INSIDE CWD
Success
Prompt > pwd
foo
Prompt > md /bar2
[MKDIR] Called with pathname: '/bar2'
[ParsePath] Called with path: '/bar2'
im in parse
Pathcopy: 0xaaaaccf4a9c0
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'bar2'
[FindEntryInDir] Token not found
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: -1, ppi->lastElement: 0xaaaaccf4a9e0
[MKDIR] ParsePath result: 0
Count: 1
```

```
FATupdate: Blocksize = 512 and startblock is 1
FAT successfully updated on disk. Written 306 blocks.
Startblock: 417

After initDir[MKDIR] Updated new directory name to: 'bar2'
[findEmptyEntry] Called
[findEmptyEntry] Found empty entry at index: 4
[MKDIR] New index in parent directory: 4
[MKDIR] Writing updated parent directory to disk
[writeDirectoryToDisk] Called with startBlock: 402, numEntries: 16
[writeDirectoryToDisk] BlockSize: 512, dirSize: 4864, blocksToWrite: 10
[MKDIR] Directory 'bar2' created successfully.
Prompt > cd /bar2
[ParsePath] Called with path: '/bar2'
im in parse
Pathcopy: 0xaaaaccf4bdb0
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'bar2'
[FindEntryInDir] Found token at index: 4
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: 4, ppi->lastElement: 0xaaaaccf4bdd0
[IsADirectory] Called
[IsADirectory] isDirectory value: 1
INSIDE CWD
Success
Prompt > pwd
bar2
Prompt > cd /../
```

```
[FindEntryInDir] Searching for token: 'foo'
[FindEntryInDir] Found token at index: 2
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: 2, ppi->lastElement: 0xaaaaccf4c0b0
[ParsePath] Called with path: '/foo'
im in parse
Pathcopy: 0xaaaaccf4c090
StartDir: 0xaaaaccf11c80
Before ppi stuff
[FindEntryInDir] Searching for token: 'foo'
[FindEntryInDir] Found token at index: 2
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11c80, ppi->index: 2, ppi->lastElement: 0xaaaaccf4c0d0
[IsADirectory] Called
[IsADirectory] isDirectory value: 1
The path /foo is neither a file not a directory
Prompt > rm foo
[ParsePath] Called with path: 'foo'
im in parse
Pathcopy: 0xaaaaccf4c130
Im in the else statement
Before ppi stuff
[FindEntryInDir] Searching for token: 'foo'
[FindEntryInDir] Found token at index: 0
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11ee0, ppi->index: 0, ppi->lastElement: 0xaaaaccf4c150
[ParsePath] Called with path: 'foo'
im in parse
Pathcopy: 0xaaaaccf4c130
Im in the else statement
Before ppi stuff
[FindEntryInDir] Searching for token: 'foo'
[FindEntryInDir] Found token at index: 0
[ParsePath] Path is root ('/'), setting ppi->parent: 0xaaaaccf11ee0, ppi->index: 0, ppi->lastElement: 0xaaaaccf4c170
[IsADirectory] Called
[IsADirectory] isDirectory value: 1
The path foo is neither a file not a directory
Prompt >
```

- Overview of shell running in M2:
    - Through the output and implementation of our M2, we see the ability to properly make a directory, through md /foo and md /bar as they are successfully created and assigned a specific block number.
        - Furthermore since these directories are created in separate blocks and are not dependent on each other we see the use of ‘pwd’ printing the proper directory/file you are in.
    - Issues start to arise as you cd into a directory, when cd’ing and creating another directory it shows a success.
        - However when cd’ing into the child, and doing pwd, it no longer shows the entire proper path, and simply the specific directory you are within.
- When it comes to rm and ls, we can see that these implementations aren’t fully functional, there are various edge cases which we haven’t covered.
    - Ls and rm were working at a certain point in our file system, however when trying to fully fix mkdir along with cwd and so on, this affected our other file applications and thus lead to the results above.

Note:

- We’d like to note that within office hours we did reach a point where ls was working and certain basic implementations were working to an extent.
    - Ls was working properly listing all directories and such, however as we continued to fix our other issues the command no longer started to work.

- We had also fixed an issue with indexing at some point, wherein the blocks were being written to the appropriate block number when it came to our directory entries, however it no longer wrote location to the hex dump and simply the file name only.
    - Furthermore we implemented many debugging print statements to allow grader/user to follow the process of such commands to show our attempts in trying to understand the issues of our file system.