Team DALJ

Anthony Benjamin | 921119898

Nyan Ye Lin | 921572181

David Chen | 922894099

Joshua Hayes | 922379312

1. Description of File System

Our filesystem is based around a bitmap array contiguous allocation. We begin by formatting our volume if it's not been initialized yet. There are important values that our system must know for our entire file system. So there needs to be some level of data persistence at runtime and on the disk. We can achieve this by using a struct called the Volume Control Block. Our Volume Control Block, short for VCB, will hold all information necessary for the volume to be initialized. In other Operating Systems, this is commonly known as the superblock or the master file table. Our VCB contains information like our signature (magic number), the size of each individual block that's stored onto our disk, the amount of blocks that the filesystem contains, the amount of blocks that we have free to allocate. It also contains the starting block numbers of our free space manager, and our root directory. These are key, as the filesystem doesn't know anything other than blocks and their locations. Our filesystem can't just look for a directory/file on the disk by using its name, as that's not how it's implemented on the disk layer. So we must use block locations in order to understand where our directory entries are stored. We will discuss lookups soon.

In our Sample Volume, we allocated 10 MB of space. This amounts to about 19,531 blocks since each block size is 512 bytes. We would store these values within our VCB so that we can reference them later to help our file system work in it's intended way.

You may have noticed that I mentioned a signature value in our VCB, we will call this our magic number. What does this magic number do? It helps us indicate whether or not a volume needs to be formatted. We do this simply by running a check on the

signature integer value, if it's not stored in our VCB, or if it doesn't match the signature that's already in the VCB, then we need to run the initialization code for populating our VCB members. The signature is any arbitrary large integer value.

Now, we move onto storing where the starting block of our free space manager is. However, lets take a step back. What exactly is our free space manager? The free space manager is in charge of telling the file system whether a block is in use/not in use. This is implemented with a bitmap array with 1's and 0's signifying whether or not the block is in use. We must make an important distinction with our file system, we use contiguous allocations. This means that files are not associated with blocks all over the place. Each file is stored one block after the other, until we've reached the end block of the file.

The free space manager code that is associated with reading and writing to the bitmap array is written to our disk. In order to locate the manager, we store the starting block to our VCB. This variable is globally accessible to our filesystem.

Finally, we store the starting block number of our root directory. The way we store the code is exactly the same process as the free space manager code, it's just a different implementation. Before we explain how the code of initializing any directory, let alone the root, we must make an important distinction. Each directory and file in the filesystem is just a directory entry. Essentially, they share the same struct members (we will visit this later), but just have a macro associated with it identifying it with either a file or directory.

When initializing our root directory, we reserve space in memory with malloc for the directory entry. Let's widen our scope for a minute and understand exactly what a directory entry is. As mentioned before, a directory entry is what stores data related to a

file or directory. We have a struct set up to store the data pertaining in an entry. We have the ID which stores a unique ID that is associated with the directoryEntry, isFile which is a binary value that's associated whether the entry is a file or directory, the location which stores the starting block location of the entry. Also, we have the fileSize in the struct that contains the file size in bytes. We have different timestamp members in the struct that are associated with whenever the entry is created, last accessed, and last modified.

Finally, we have members that will contain the fileName and the author of the file.

When we initialized an entry, all of these values must be populated in order to have a valid entry.

```
typedef struct directoryEntry
{
    // size of struct is 136 bytes
    int id;               // unique ID that is associated with the directoryEntry
    int isFile;           // boolean value that's assocaited whether the entry is a file or directory.
    int location;         // stores block location of file
    int fileSize;         // the file size in bytes
    time_t createDate;    // timestamp associated with when the file was created by FS
    time_t lastAccessDate; // timestamp associated when the file was last accessed by FS
    time_t lastModifyDate; // timestamp associated when the last write of a file occurs
    char fileName[64];    // string with max 64 characters that contains the file name
    char author[32];      // string with max 32 characters that contains the author name of the file.

} directoryEntry;
```

Now that we know what exactly is stored within an entry, we can begin to discuss how exactly these entries are initialized. We have a helper routine set up called initDir, which helps us solve this problem. This function takes in the minimum number of entries that the filesystem

needs to be able to handle, and a pointer to the parent directory. What if the pointer directory doesn't exist, AKA the root directory? We'll simply set this pointer to a NULL pointer indicating that it doesn't exist. The function then calculates the amount of memory required to store the specified number of directory entries, then finds the required number of free blocks using the free space manager. Once we know these values, we allocate memory for the directory entries using the "malloc" function and initializes each entry with empty string values and appropriate attributes.

Finally, we put the starting block of our root block inside our VCB to be able to reference later. Once all our VCB values have been initialized, we write the VCB to disk to ensure data persistence between compilations.

Now that we've covered, let's take a deeper look into our filesystem functions. We needed to mimic common filesystem operations. That means we implemented: fs_setcwd (set current working directory), fs_getcwd (get current working directory), fs_isFile(check if directory entry is a file), fs_isDir (check if directory entry is a directory), fs_mkDir (makes a directory in our filesystem), fs_openDir (opens up a directory and gives it the first free file descriptor), fs_readdir(reads a directory by a specific amount of bytes specified by the user, then returns the bytesRead to the user), fs_closedir (free all resources associated with the file, including the file descriptor), and fs_stat (fill in the fs_stat struct to show all the information of the file).

Most of our file system functions share a common utility, and this is the ability to traverse of a given path by our system, then getting the entry's parent directory and it's index in the parent directory entry. We call this function parsePath.

This converts a path name, denoted by a string, into an accessible directory entry structure that the file system can understand and navigate. Paths can be either directories, or files, and they can be relative or absolute. parsePath takes care of all of this functionality for us

to guarantee something accessible for the filesystem. Without a function like parsePath(), file system functions would need to include their own parsing logic, which could be prone to errors and difficult to maintain as a developer. By using parsing logic that's shared throughout the file system, we can guarantee consistent and intended results.

One of the first functions we had implemented was fs_mkdir(char *pathName, mode_t mode). In order to check if the directory entry already exists, we call parsePath and check if the parent directory of the path exists, but the pathName parameter doesn't exist within the parent directory, then we can create a directory. We look for the first open index slot in the parent, and then proceed to create the directory in that index.

```
int fs_mkdir(const char *pathname, mode_t mode)
{
    parsedPath parsed = parsePath(pathname);
    if (parsed.parent != NULL && parsed.index == -1)
    {
        // Searches parent directory for available slot
        int entryIndex = findOpenEntrySlot(parsed.parent);

        if (entryIndex > 0)
        {
            // Create directory
            directoryEntry *createDir = initDir(initDirAmt, parsed.parent);
            // copy directory info into parent
            strcpy(parsed.parent[entryIndex].fileName, parsed.dirName);
            parsed.parent[entryIndex].location = createDir->location;
            parsed.parent[entryIndex].lastModifyDate = createDir->lastModifyDate;

            // writes back to disk
            int bytesNeed = initDirAmt * sizeof(directoryEntry);
            int blkCount = (bytesNeed + vcb->blockSize - 1) / vcb->blockSize;
            LBAwrite(parsed.parent, blkCount, parsed.parent->location);

            // free memory allocations
            free(parsed.parent);
            parsed.parent = NULL;
            return 0;
        }
    }
    // Unable to reach directory or
    printf("Unable to create directory %s\n", pathname);
    return -1;
}
```

Now that we know how to make directories, how can we remove them? That's where the function fs_rmdir() comes in. Before we continue, we must touch on one key point. Our file system won't allow the user to delete a directory if there's files contained in it. If there's no files

contained in the directory, we can continue with the removal of the directory. We then calculate the size we would need to malloc for the entry. We begin this by referencing the VCB, and our initDirAmount (variable is 52 which means we can have 52 total directory entries), then calculated the bytes that we need, the blocks that we need to load the directory entry, and the bytes in use by the directory entry. We then this function by taking in a path name, and then using this to get the parent directory and the index of the child in the parent from parse path. Afterwards, using parsePath we can check if the child exists in the parent and if the child is a directory. If these are logically true, then we know the path is reachable and we can continue with the removal process. After this, using the values we calculated in the beginning, we allocate memory towards our directory entry, and then read our directory entry from disk into memory. If the directory entry is empty, meaning there's no other entries within it, then we can go through with our final step in the removal process. The function will then remove the directory by setting its location as free, setting it's fileSize to 0, and clearing its name. Then we write back to our disk the updated parent directory. We finally have to update the free space manager to represent the newly freed blocks. We free up any allocated memory, then return 0 to indicate a successful removal of our directory.

```c
#include <stdio.h>
#include "mfs.h"
#include <stdlib.h>
#include <unistd.h>
#include "directoryEntry.h"
#include "fsLow.h"
#include "vcb.h"
#include "parsePath.h"
#include "freeSpaceManager.h"

int fs_rmdir(const char *pathname)
{
    //Calculate size needed to malloc for directory
    int bytesNeed = initDirAmt * sizeof(directoryEntry);
    int blkCount = (bytesNeed + vcb->blockSize - 1) / vcb->blockSize;
    int byteUsed = blkCount * vcb->blockSize;

    parsedPath parsed = parsePath(pathname);  //get parent directory and index of child
    if (parsed.index > 0 && parsed.parent[parsed.index].isFile == DIRECTORY)
    { // If path is reachable
        // load dir into memory
        directoryEntry *checkDir = malloc(byteUsed);
        LBAread(checkDir, blkCount, parsed.parent[parsed.index].location);

        // checks the directory for any files/directories before attempting to delete
        int directoryEmpty = directoryIsEmpty(checkDir);
        if (directoryEmpty == 0)//directory was found to be empty-> delete the directory
        {
            parsed.parent[parsed.index].location = 0;//make location as free
            parsed.parent[parsed.index].fileSize = 0;//set file size to 0;
            strcpy(parsed.parent[parsed.index].fileName, "\0");// clear name
```

```
46              strcpy(parsed.parent[parsed.index].fileName, "\0");// clear name
47
48              LBAwrite(parsed.parent, blkCount, parsed.parent[0].location);
49
50              unsigned char *freeSpaceManager =
51                  malloc(vcb->sizeOfFreeSpaceManager * vcb->blockSize * sizeof(char));
52              freeBlocks(freeSpaceManager,
53                  vcb->freeSpaceManagerBlock,vcb->sizeOfFreeSpaceManager);
54                  free(freeSpaceManager);
55                  freeSpaceManager = NULL;
56          }
57          // free memory allocations
58          free(checkDir);
59          free(parsed.parent);
60          free(parsed.path);
61
62          parsed.parent = NULL;
63          parsed.path = NULL;
64          checkDir = NULL;
65          return 0;
66      }
67
68      return -1; //Unsuccessful removal of directory
69  }
```

We created a function - fs_opendir(), in order for the filesystem to pass in a pathname into this function so that we can load the directory entry into memory to be used by other functions. We do this by reserving space in memory for the directory entry struct that we will eventually fill in with values associated with the struct. Then we call parsePath on the path name in order to access the directory on disk if it exists. Once we've loaded the directory from disk, we put the directory entry metadata information into memory so it's easily accessible.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "directoryEntry.h"
#include "mfs.h"
#include "parsePath.h"
fdDir *fs_opendir(const char *pathname)
{
    fdDir *dir = malloc(sizeof(fdDir));
    parsedPath parsed = parsePath(pathname);
    if (parsed.index == -1)
    {
        printf("open error: can't found such directory\n");
        return NULL;
    }
    if (parsed.parent[parsed.index].isFile != DIRECTORY)
    {
        printf("open error: this is not a directory\n");
        fs_closedir(dir);
        return NULL;
    }
    dir->lastAccessDate = time(NULL);
    dir->directoryStartLocation = parsed.parent[parsed.index].location;
    dir->dirEntryPosition = 0;
    dir->DE = malloc(parsed.parent[parsed.index].fileSize);
    dir->DE->fileSize = parsed.parent[parsed.index].fileSize;
    strcpy(dir->dirinfo->d_name, parsed.parent[parsed.index].fileName);
    dir->d_reclen = sizeof(fdDir);
    return dir;
}
```

Now, let's get into how fs_readdir() works. We've loaded the metadata associated with our directory entry into memory. We want to read the contents of a directory and return information associated with the directory entry. The function will take in a pointer to our directory entry in memory. This pointer would be initialized from the open directory function, otherwise, this function will not be able to read the parameter and return a NULL pointer. If it's valid, we will calculate the number of blocks needed to read the directory entry based on the directory's file size and block size. Once we know this number, we can use LBAread to read the amount of

blocks from disk into memory. Using this, we will begin to populate the values of the fs_diriteminfo struct and return a pointer to this struct afterwards.

```
struct fs_diriteminfo *fs_readdir(fdDir *dirp)
{
    if (dirp == NULL)
    {
        printf("read error: can't found directory\n");
        return NULL;
    }
    int blocks = (dirp->DE->fileSize + vcb->blockSize - 1) / vcb->blockSize;
    LBAread(dirp->DE, blocks, dirp->directoryStartLocation);
    dirp->dirEntryPosition++;
    if (dirp->DE[dirp->dirEntryPosition].isFile == DIRECTORY)
    {
        dirp->dirinfo->fileType = FT_DIRECTORY;
    }
    else
    {
        dirp->dirinfo->fileType = FT_REGFILE;
    }
    dirp->dirinfo->d_reclen = sizeof(dirp->dirinfo);
    strcpy(dirp->dirinfo->d_name, dirp->DE[dirp->dirEntryPosition].fileName);
    return dirp->dirinfo;
}
```

The next function we had worked on is fs_closedir(). This function would take in a directory entry parameter from the filesystem and “close” the open directory. “Close” in this context means that we would set directory entry values to NULL/0 and then free up the memory that’s associated with the directory entry struct. If the directory successfully closed, we will return 0 in the function. Otherwise, we return a -1, indicating that we weren’t able to find the open directory to close.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "directoryEntry.h"
#include "mfs.h"
#include "parsePath.h"
#include "vcb.h"
#include "fsLow.h"
int fs_closedir(fdDir *dirp)
{
    if (dirp == NULL)
    {
        printf("close error: no such directory\n");
        return -1;
    }
    dirp->lastAccessDate = time(NULL);
    dirp->dirEntryPosition = 0;
    dirp->dirinfo = NULL;
    dirp->directoryStartLocation = 0;
    free(dirp->DE);
    free(dirp->dirinfo);
    free(dirp);
    return 0;
}
```

Let’s discuss our next function in detail - fs_delete(). This functions purpose is to take in a file name and simply delete it from within a directory. We begin by checking if the path exists by calling parsePath. If it exists and the file in question, is a file, not a directory, then we know we can delete the file. The function works similarly to fs_rmdir() in that we set the directory entry’s (file) location to 0, and the fileSize to 0. We then strcopy a null-terminator to the full name. These are the 3 indicators that the file doesn’t exist. Once we’ve done that, we write it back to our disk, free up our memory allocations that we made throughout the runtime of this function, and set the parent and it’s path to NULL.

```c
#include <stdio.h>
#include "mfs.h"
#include <stdlib.h>
#include <unistd.h>
#include "directoryEntry.h"
#include "fsLow.h"
#include "vcb.h"
#include "parsePath.h"
#include "freeSpaceManager.h"

int fs_delete(char *filename)
{
    parsedPath parsed = parsePath(filename);
    int blkCount = //Calculate filesize in blocks
        (parsed.parent[parsed.index].fileSize + vcb->blockSize - 1)/vcb->blockSize;

    if (parsed.index > 0 && parsed.parent[parsed.index].isFile == FILEMACRO)
    {
        // If path is reachable-> mark entry as avail on disk
        parsed.parent[parsed.index].location = 0;
        parsed.parent[parsed.index].fileSize = 0;
        strcpy(parsed.parent[parsed.index].fileName, "\0");

        LBAwrite(parsed.parent, blkCount, parsed.parent[0].location);
        // TODO set freespace blocks to available

        // free memory allocations
        free(parsed.parent);
        free(parsed.path);
        parsed.parent = NULL;
        parsed.path = NULL;
        return 0;
    }

    return -1; //Unsuccessful file delete
}
```

The next function we created was fs_getcwd(). This function's responsibility is to grab the current working directory, and return it back to the user. There were many ways of going about this but we decided to make the current working directory a global variable. This made the implementation of this function extremely easy. The function took two parameters, the pathname, and the size of the user's buffer. We check if the currentWorkingDir can fit in their buffer, and if it can't, we return back a NULL pointer. Otherwise, we return the currentWorkingDir variable to the user.

```c
#include <stdio.h>
#include "mfs.h"
#include <stdlib.h>
#include <unistd.h>
#include "directoryEntry.h"
#include "fsLow.h"
#include "vcb.h"
#include "parsePath.h"

char *fs_getcwd(char *pathname, size_t size)
{
    if (strlen(currentWorkDir) > size)
    {//User given buffer is not large enough to store cwd
        return NULL;
    }
    //Return the cwd to user.
    strcpy(pathname, currentWorkDir);
    return currentWorkDir;
}
```

Now that we have set up our getter function appropriately, our next step was to implement a setter function so that we can change the global currentWorkingDir variable throughout runtime. The function fs_setcwd() will handle this. It will take in a pathname as the parameter, and we will call parsePath to check if it exists. If the path exists we'll simply set the path to the currentWorkingDir global variable and return 0 to signify successful operation. Otherwise, we will return a -1, which will indicate failure.

```c
#include <stdio.h>
#include "mfs.h"
#include <stdlib.h>
#include <unistd.h>
#include "directoryEntry.h"
#include "fsLow.h"
#include "vcb.h"
#include "parsePath.h"

int fs_setcwd(char *pathname)
{
    parsedPath parsed;
    parsed = parsePath(pathname); //Checks for valid path
    if (parsed.index > -1)
    {
        strcpy(currentWorkDir, parsed.path); //Update cwd

        // free memory allocations
        free(parsed.parent);
        free(parsed.path);
        parsed.parent = NULL;
        parsed.path = NULL;

        return 0;
    }
    return -1; //Unsuccessful in setting cwd
}
```

Throughout our file system, we kept on having to check whether or not a directory entry was a file or a directory. Let's get into our two functions that handle this - fs_isDir() and fs_isFile(). They both work in the same way, just with one minor difference. We begin by taking in a filename as a parameter, and using the helper routine parsePath to check if the path exists. If the path exists, we will reference the parent that was returned from the parsePath function. The parent will have a member called "isFile", and this will be either 1 indicating a directory, or 0 indiciating a file. We will then return the value depending on if it's a file or directory.

```c
 **************************************************************/
#include "mfs.h"
#include "directoryEntry.h"
#include "parsePath.h"

int fs_isFile(char *filename)
{

    parsedPath path = parsePath(filename);
    if (path.index > 0)
    {
        directoryEntry *dir = path.parent;
        if (dir->isFile == FILEMACRO)
        {
            return FILEMACRO;
        }
    }

    return DIRECTORY;
}
```

```
 **************************************************************/
#include "mfs.h"
#include "directoryEntry.h"
#include "parsePath.h"

int fs_isDir(char *filename)
{

    parsedPath path = parsePath(filename);
    if (path.index > 0)
    {
        directoryEntry *dir = path.parent;

        if (dir->isFile == DIRECTORY)
        {
            return DIRECTORY;
        }
    }

    return FILEMACRO;
}
```

Finally, let’s discuss our last file system function - fs_stat().

```c
int fs_stat(const char * path, struct fs_stat *buf)
{
    parsedPath parsed = parsePath(path);

    int index = parsed.index;

    printf("Index From Parsed Path: %d\n", index);

    buf = malloc(sizeof(fs_stat) * 100);

    if (index > -1)
    {
        buf->st_size = parsed.parent[index].fileSize;
        printf("Size : %ld\n", buf->st_size);

        buf->st_blksize = vcb->blockSize;
        printf("Block Size : %ld\n", buf->st_blksize);

        buf->st_blocks = (parsed.parent[index].fileSize / vcb->blockSize) + 1;
        printf("Number of Blocks: %ld\n", buf->st_blocks);

        buf->st_accesstime = parsed.parent[index].lastAccessDate;
        printf("Access Time %ld\n", parsed.parent[index].lastAccessDate);

        buf->st_modtime = parsed.parent[index].lastModifyDate;
        printf("Modified Time : %ld\n",buf->st_modtime);

        buf->st_createtime = parsed.parent[index].createDate;
        printf("Create Time : %ld\n", buf->st_createtime);

        return 0;

    }


    return -1;
}
```

For our fs_stat() function, we used the index that is returned from parsedPath which take the path from our fs_stat arugment and we malloced the fs_stat pointer with the memory to use. We checked if the index is valid or not. If it is valid, we used that index to search the directory entry that we need in the parent directory to get all the information that is needed and fill in the information from the directory entry to our fs_stat struct pointer from the argument.

2. A discussion of what issues you faced and how your team resolved them.

Our team faced multiple issues, mainly around the conceptual understanding of how the initialization of our file system would work. One of our problems was not understanding how the magic number works. We were wondering what the actual number should be. After speaking to some classmates, I found out that the magic number is arbitrary. So long that it stays the same between compilation, we will maintain data persistence.

Another issue that we had run into was not understanding how the free space manager works. We had gone back to the lecture video and the professors video helped us out a ton. We realized that we just needed to allocate a array with 1's and 0's into memory, and these would indicate whether a specific block is free or occupied. We initialized the space that the free space manager took up, with 1's in our bitmap array, then we free'd up the rest of the bitmap array.

Within the parsepath implementation, there is a char * that is named "path". One of the recurring issues for updating this variable was that it would sometimes not be empty upon initialization and would return the concatenated pathname but preceded with illegible characters. The issue didn't always happen when entering the same order of commands which made it difficult to debug. However, while implementing the resolvePath function, which included moving some of the functionality from parsePath, we were able to not only write a function that handled resolving a pathname that is layered with ".."s and "."s, but also it helped spot light where the issue was beginning at. Right after the malloc, the variable in resolvePath was not already null. So to fix this, we strcpy'ed the null terminator the string after memory allocation.

There are some issues when I was implementing open, read, and close directory. When I was looking at the fddir structure, some of the elements I have no idea what it's doing for, and I don't know what's needed to add into the structure. I thought the open function would be like the b_open, but there are no such things like file control block arrays. However, the professor had talked about it in the class, so I could know what should be in the structure for implementing, and I also realized it when I was implementing the read directory function.

When we were doing b_write, we had tried to figure out how to handle the blocks of growing files, but we've realized our system is managing the blocks contiguous, since the file blocks are contiguous, the blocks of file are fixed, so we can't handle the growing file blocks.

3. How our driver program works?

We begin our driver program in the fsInit.c file. We define our magic number here for our VCB to some arbitrarily large value, as mentioned previously. This will be used later to check if our volume needs to be formatted. We will also reference our global VCB in this file, for use later on in the driver. Our function initFileSystem(), does as the name suggests, initalizes our file system. We allocate memory for our VCB so that we can directly reference it's values throughout the initialization phase. Then, we read block 0 from disk into our VCB, once that's been successfully read we move onto the next step. Now that we have our VCB in memory, we can check if our magic numbers match from our define block, compared to the magic number stored in the VCB. If it's not the same value, we must format our volume. By definition, if our volume needs to be formatted, then we must initalize all of the values of the VCB. The values that are contained within the VCB are the signature, each block size, the total amount of blocks we have, and the number of free blocks. We set these to the values through the parameters of the function, or global variables that we access. Now, we need to set the starting block numbers

of our free space manager, and our root dir. These have helper routines associated with them in order to find the values of these. Let's dive deeper into the first one.

We have our first helper routine that will give us the starting block for our freeSpaceManager, it's called initFreeSpaceManager(). The parameters the routine takes in are the total blocks in the VCB, and blockSize defined in the VCB. The routine starts off by calculated how many bytes the free space manager will need to handle (bytesNeeded). This is done by calculating (total blocks / 8) + 1. We divide total blocks by 8 because we need to represent each block by 1 bit, not 1 byte. And 1 byte equals 8 bits. We also add 1 to ensure that everything will fit, due to there being the case that totalblocks / 8 could give us a partial part of a file, since it can't be a decimal value. The + 1 will ensure that we won't hit this edge case. Next, we need to determine how many blocks the freeSpaceManager will require to fill our bitmap. This is found by doing (bytesNeeded / blockSize) + 1. We will then store this value into our VCB under the member name - sizeOfFreeSpaceManager. We will then initalize our freeSpaceManger using these variables, then write it to back to our disk. We will then return 1, since that's where we always will store our free space manager in our disk.

```
25  /**
26   * initFreeSpaceManager initalizes our bitmap array. We reserve space
27   * in memory for the manager, then we fill the first blocks that the manager takes up
28   * as occupied (1). We then fill up the rest of our bitmap array with 0's to indicate free blocks.
29   *
30   * We then write to the disk and return the starting position of the freeSpaceManager
31   */
32
33  int initFreeSpaceManager(int totalBlocks, int blockSize)
34  {
35      // bytes required for our bitmap
36      // 1 bit represent 1 block
37      // 19531 blocks == 19531 bits == 2442 bytes
38      int bytesNeeded = (totalBlocks / 8) + 1;
39
40      // blocks required for our bitmap
41      // 5 blocks
42      int freeSpaceManagerBlocks = (bytesNeeded / blockSize) + 1;
43      vcb->sizeOfFreeSpaceManager = freeSpaceManagerBlocks;
44
45      // initializing our bitmap
46      // freeSpaceManager with size 2442 bytes
47      unsigned char *freeSpaceManager = (unsigned char *)malloc(bytesNeeded);
48
49      // set all the bits to zero
50      memset(freeSpaceManager, 0, bytesNeeded);
51
52      // mark first 6 blocks is used.
53      // 1 block for VCB and 5 blocks for fsManager
54      // total 6 blocks
55      for (int i = 0; i < freeSpaceManagerBlocks + 1; i++)
56      {
57          setBit(freeSpaceManager, i,1);
58      }
59
60      // write back bitMap to disk
61      LBAwrite(freeSpaceManager, freeSpaceManagerBlocks, 1);
62
63      // return the starting block of free space
64      return 1;
65  }
66
```

```
69  void setBit(unsigned char * freeSpaceManager, int bitIndex, int isFree)
70  {
71      // get the byte index;
72      int byteIndex = bitIndex / 8;
73      // get the bit index in the byte
74      int bitIndexInByte = bitIndex % 8;
75
76      // 1 => not free
77      // 0 => free
78      if(isFree == 1)
79      {
80          // setting the bit to 1 if isFree == 1
81          freeSpaceManager[byteIndex] |= 1 << bitIndexInByte;
82      }
83      else
84      {
85          // setting the bit to zero if isFree == 0
86          freeSpaceManager[byteIndex] &=  ~(1 << bitIndexInByte);
87      }
88  }
89
```

```
91  int checkBit(unsigned char * freeSpaceManager, int bitIndex)
92  {
93      int byteIndex = bitIndex / 8;
94      int bitIndexInByte = bitIndex % 8;
95
96      // checking if the bitIndex is zero (free) or not (not free)
97      // return 0 if it is bitIndex is free (represent false in c)
98      // return non-zero integer if not free (retpresent true in c)
99      return (freeSpaceManager[byteIndex] & (1 << bitIndexInByte) ) != 0;
100 }
101
```

```c
int findFreeBlocks(int requestedBlocks)
{

    unsigned char * freeSpaceManager = malloc(5 * vcb->blockSize * sizeof(char));
    LBAread(freeSpaceManager,5,1);

    int startIndex = -1;
    int freeBlocks = 0;

    for(int i = 0; i < vcb->totalBlocks - requestedBlocks; i++)
    {
        // if checkBit return false
        // 0(free) reprenset false in c
        if(!checkBit(freeSpaceManager,i))
        {
            // if first free block
            if(startIndex == -1)
            {
                // set startIndex with first free block
                startIndex = i;
            }
            freeBlocks++;

            if(freeBlocks == requestedBlocks)
            {
                // mark those free contiguous blocks as used
                for(int j = startIndex; j < startIndex + requestedBlocks; j++)
                {
                    setBit(freeSpaceManager,j,1);
                }
                // write back bitMap to disk
                LBAwrite(freeSpaceManager,5,1);
                return startIndex;
            }
        }
        else
        {
            // resetting if block is not free
            startIndex = -1;
            freeBlocks = 0;
        }

    }
    return -1;
}
```

```
152
153     void freeBlocks(unsigned char * freeSpaceManager, int startIndex,int numberOfBlocks)
154     {
155         for(int i = startIndex; i < startIndex + numberOfBlocks; i++)
156         {
157             // setting the bits to zero
158             setBit(freeSpaceManager,i,0);
159         }
160         LBAwrite(freeSpaceManager,5,1);
161     }
162
```

Our code related to our free space manager is shown above.

Now that we've stored the starting block number of our free space manager in our VCB we can continue to store the last value in our VCB, the starting block of our root directory. We have another helper routine set up so we can handle this problem. This one is much less complex to our free space manager. We simply will allocate memory and reserve this for the directory entry. Then we call initDir (previously mentioned in the write up, if curious on implementation) and store it in the newly allocated pointer we just created. Now, we're able to get the location of this directory entry from the struct we malloc'd. We store the size of the root directory into our VCB, then return the location of the root directory back to the caller.

```
29
30  int initRootDir(int blockSize)
31  {
32          // 52 directory entries * 136 sizeof 1 directory entry = 7072 bytes/ 512 chunks = 14 blocks
33
34          directoryEntry *dir = (directoryEntry *)malloc(initDirAmt);
35          dir = initDir(initDirAmt, NULL);
36          int rootposition = dir[0].location;
37          vcb->rootDirSize = dir->fileSize;
38
39          free(dir);
40          dir = NULL;
41
42          return rootposition;
43  }
44
```

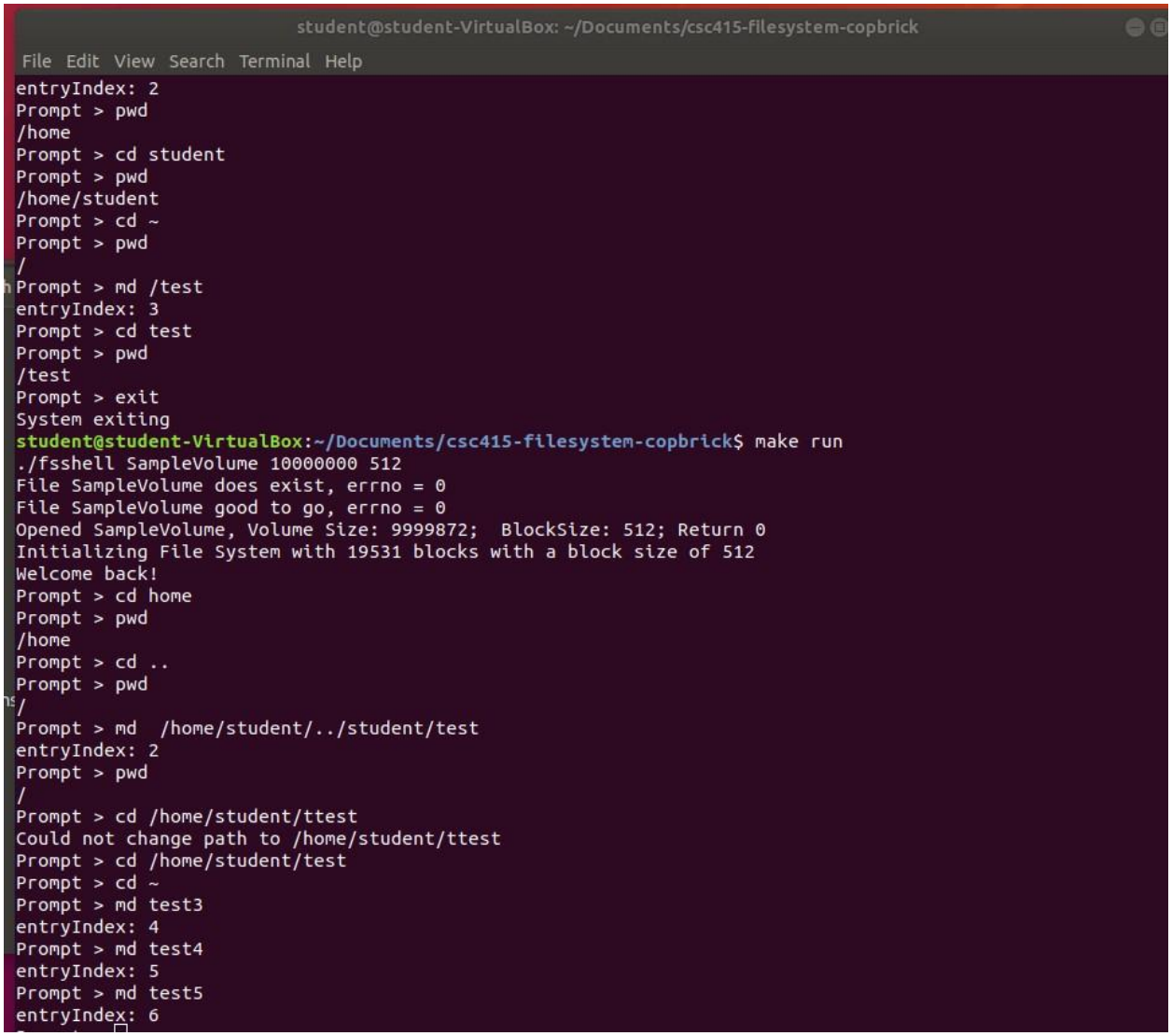
Our initFileSystem code is shown here:

```
83  void exitFileSystem()
84  {
85          printf("System exiting\n");
86          free(vcb);
87          vcb = NULL;
88  }
```

```c
int initFileSystem(uint64_t numberOfBlocks, uint64_t blockSize)
{
        printf("Initializing File System with %ld blocks with a block size of %ld\n", numberOfBlocks, blockSize);
        /* TODO: Add any code you need to initialize your file system. */
        vcb = malloc(blockSize);

        if (LBAread(vcb, 1, 0) == 1) // Read block 0 from disk into VCB
        {

                if (vcb->signature != magicNumber) // System not initalized
                {
                        // init vcb since signature is missing or does not match
                        vcb->signature = magicNumber;
                        vcb->blockSize = blockSize;
                        vcb->totalBlocks = numberOfBlocks;
                        vcb->freeBlocks = numberOfBlocks;

                        // struct fs_stat *test;
                        // int result = fs_stat("~",test);
                        // printf("RESULT : %d\n",result);

                        // Set vals returned from init procedures
                        vcb->freeSpaceManagerBlock = initFreeSpaceManager(vcb->totalBlocks, vcb->blockSize);
                        vcb->rootDirBlock = initRootDir(vcb->blockSize);

                        LBAwrite(vcb, 1, 0); // LBAwrite VCB back to disk
                }
                else
                {
                        // loadFreeSpace into memory
                        printf("Welcome back!\n");
                }
                return 0;
        }

        return -1; // Unable to read from disk
}

```

And then once the user wants to exit from the file system, we will call our exitFileSystem function that frees up all of the VCB's resources.

Screenshots of showing each of the command listed


student@student-VirtualBox: ~/Documents/csc415-filesystem-copbrick
File Edit View Search Terminal Help
entryIndex: 2
Prompt > pwd
/home
Prompt > cd student
Prompt > pwd
/home/student
Prompt > cd ~
Prompt > pwd
/
Prompt > md /test
entryIndex: 3
Prompt > cd test
Prompt > pwd
/test
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-copbrick$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Welcome back!
Prompt > cd home
Prompt > pwd
/home
Prompt > cd ..
Prompt > pwd
/
Prompt > md /home/student/../student/test
entryIndex: 2
Prompt > pwd
/
Prompt > cd /home/student/ttest
Could not change path to /home/student/ttest
Prompt > cd /home/student/test
Prompt > cd ~
Prompt > md test3
entryIndex: 4
Prompt > md test4
entryIndex: 5
Prompt > md test5
entryIndex: 6