Mohammad Dahbour 921246050
Diego Flores 920372463
Kemi Adebisi 921140633
Github: DiegoF001 CSC415-02 Operating Systems

The Baha Blast

GITHUB LINK: https://github.com/CSC415-2023-Spring/csc415-filesystem-DiegoF001

Description:

We created a file system that uses indexed allocation; each file has indexes for data locations that contain information about the file/directory, such as its size, and pointers to the blocks that make up the file's content. The data blocks containing the actual content of the file are scattered throughout the disk, and are linked to the index block through the use of our data locations array. The index block contains an array of indices that point to the data blocks, allowing for efficient access to the file's content. The advantage of using indexed allocation is that it allows for files of varying sizes to be stored efficiently, as well as allowing for fast random access to the data blocks that make up a file. However, this approach can lead to fragmentation, as small files may occupy a larger number of blocks than they actually need, leading to wasted space on the disk. Additionally, indexing may lead to slower performance in certain situations, as the index block must be read before the file's data blocks can be accessed.

For managing space, we used a bitmap. Our bitmap represents each and every block in our file system. It is a very easy and intuitive way of keeping track of what blocks are being used and what blocks are free.

Mohammad Dahbour 921246050
Diego Flores 920372463
Kemi Adebisi 921140633
Github: DiegoF001

Detail of how our driver program works:

This program is an implementation of a filesystem using indexed allocation. Indexed allocation is a file allocation method where a file's data blocks are not stored in contiguous blocks, but rather in a separate index block, which stores the addresses of the data blocks. This code initializes the root directory of a file system by creating a DirectoryEntry array and setting its values. The root directory is assigned the name "." and its parent directory is assigned the name "..". The root directory is marked as a directory and its creation date, last access date, and last modification date are set to the current time. The size of the directory is calculated based on the number of entries it can hold, and the number of blocks needed to hold it is calculated based on the block size of the file system.

Our file system starts by initializing our Volume Control block and it makes sure that we know how many blocks are part of our file system, where the root starts, and block size, and a magic number. Upon initialization, we assign a magic number to our vcb and ensure that our entire file system is only initialized once.

Next, we initialize our root directory. We set the name to "." and the parent "..", for root, they are both the same. Also, we initialized the other entries in the root to a known free state.

Our bitmap is initialized right after our VCB is, we do simple arithmetics to figure out how many integers you need to represent the entire VCB, we then wrote routines that mark specific bits as used, and mark chinks of blocks as used.

The implementation defines several functions to perform operations on the filesystem. One of these functions, "fs_mkdir", creates a new directory in the filesystem. It first checks the current working directory and then creates a new directory at the specified path. If the path is absolute, the current working directory is set to the root directory. The function also extends the directory if the current directory is full.

Another function, "fs_isDir", checks whether a specified path leads to a directory or not. It first checks the current working directory and then parses the specified path. If the path is invalid, the function returns an error. If the path leads to a directory, the function returns 1, otherwise 0.

"fs_opendir" function opens a directory for reading. It takes a pathname as input and returns a pointer to a directory stream structure, which can be used to read the contents of the directory. The function marks the directory as open and stores the information about the directory in an array of open directories.

"fs_readdir" function reads the next entry from the open directory specified by the directory stream structure pointer. It returns a pointer to a structure containing information about the directory entry, including the name of the entry and its file type.

"fs_rmdir" function removes the directory specified by the pathname. It first checks the current working directory and then parses the specified path. If the path is invalid, the function returns an error. If the path leads to a directory, the function removes the directory, frees its data blocks and updates the parent directory's information. It also handles the case where the directory to be deleted is extended.

The parse_path function takes a file path as input and returns a Container structure. It first initializes the Container and DirectoryEntry structures, and creates a temporary buffer to store the file path. It then tokenizes the file path based on the delimiter "/", and stores each token in an array called path.

The function iterates through the path array and searches for the corresponding directory entry in the file system using the provided entry pointer. It reads the directory entry from the disk and checks if it matches the current token in the path array. If it does, the function moves on to the next token and continues searching. If it doesn't, the function checks if the directory entry has an extended table. If it does, it reads the extended table and continues searching through it until it either finds the correct directory entry or reaches the end of the path.

Once the correct directory entry is found, the function sets the dir_entry and index fields of the Container structure accordingly and returns it.

The check_extends function is a helper function that takes a directory name, starting block, and path piece as input. It reads an extended table from the disk and searches for the corresponding directory entry in a similar manner to the parse_path function. If it finds the correct directory entry, it sets the dir_entry and index fields of the Container structure and returns it. If it doesn't, it checks if the extended table has another extended table and continues searching through it recursively until it either finds the correct directory entry or reaches the end of the path.

Overall, the implementation of indexed allocation allows for efficient storage and retrieval of data in the filesystem. The use of separate index blocks makes it easier to manage free space and allows for extensibility.

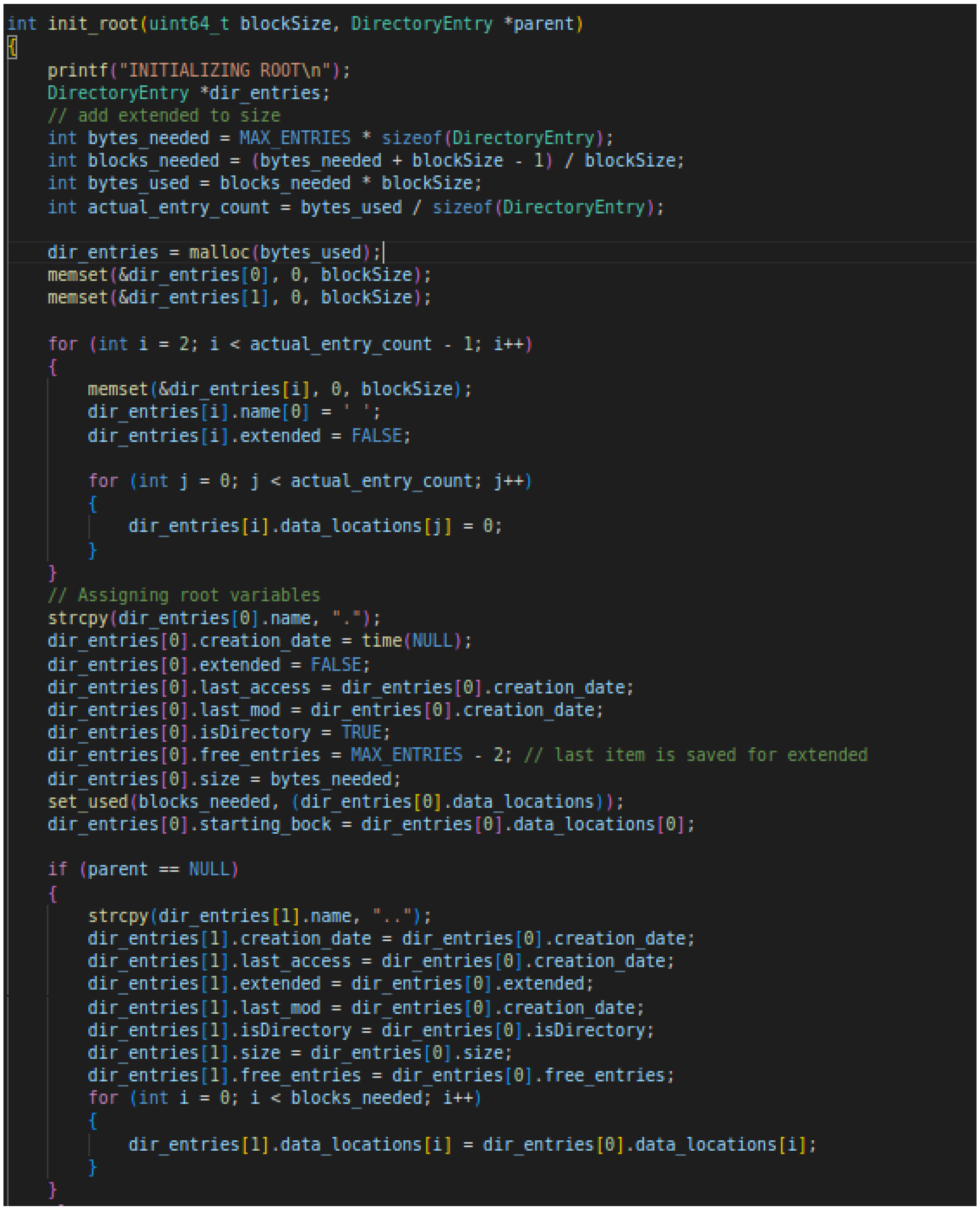
Issues we had:

Init root: When we first encountered this code, we noticed that it was not handling the case where a parent directory was not passed in. In this case, the ".." entry in the root directory would be assigned the same values as the root directory itself. To fix this issue, we added an "if-else" statement to check if the parent directory was NULL and assign the correct values accordingly.

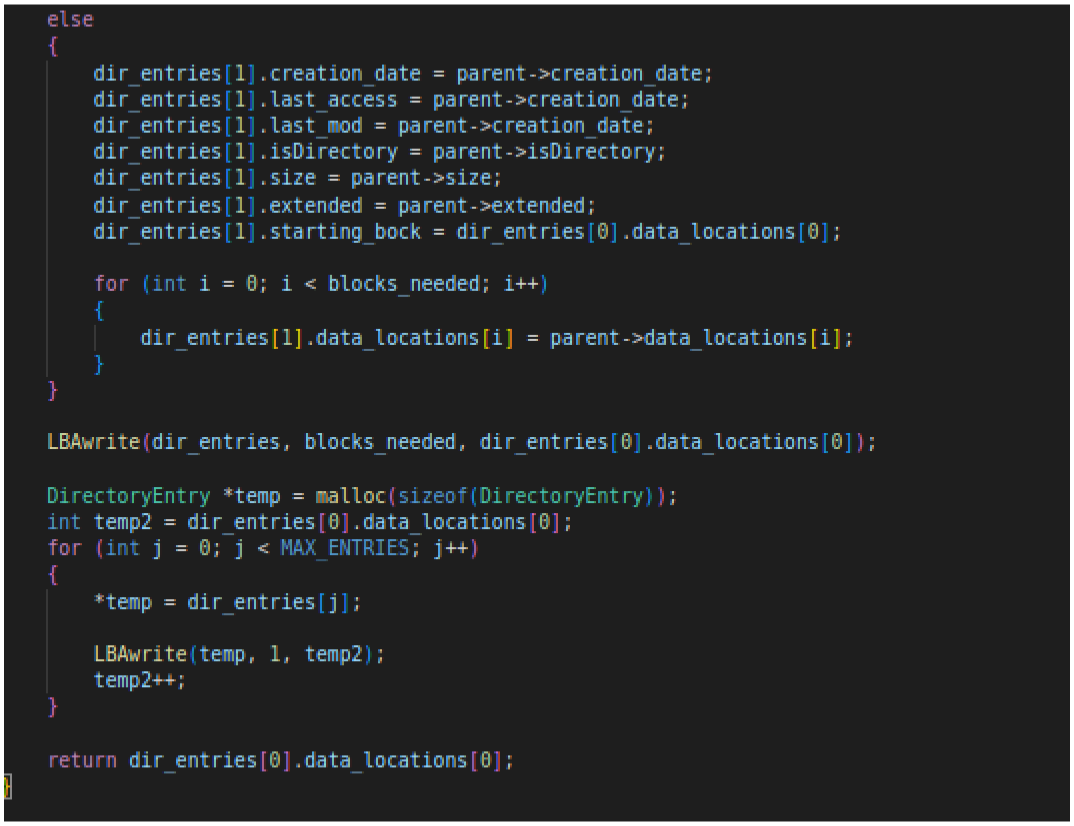
We also noticed that the code was not properly setting the starting block of the root directory. To fix this issue, we assigned the starting block to the first block in the directory's data locations array.

Finally, we noticed that the code was not writing the directory entries to disk correctly. The code was writing each entry individually, but it should have been writing the entire directory array in one call to LBAwrite(). To fix this issue, we changed the code to write the entire directory array at once.

code for init root:



```
int init_root(uint64_t blockSize, DirectoryEntry *parent)
{
    printf("INITIALIZING ROOT\n");
    DirectoryEntry *dir_entries;
    // add extended to size
    int bytes_needed = MAX_ENTRIES * sizeof(DirectoryEntry);
    int blocks_needed = (bytes_needed + blockSize - 1) / blockSize;
    int bytes_used = blocks_needed * blockSize;
    int actual_entry_count = bytes_used / sizeof(DirectoryEntry);

    dir_entries = malloc(bytes_used);
    memset(&dir_entries[0], 0, blockSize);
    memset(&dir_entries[1], 0, blockSize);

    for (int i = 2; i < actual_entry_count - 1; i++)
    {
        memset(&dir_entries[i], 0, blockSize);
        dir_entries[i].name[0] = ' ';
        dir_entries[i].extended = FALSE;

        for (int j = 0; j < actual_entry_count; j++)
        {
            dir_entries[i].data_locations[j] = 0;
        }
    }
    // Assigning root variables
    strcpy(dir_entries[0].name, ".");
    dir_entries[0].creation_date = time(NULL);
    dir_entries[0].extended = FALSE;
    dir_entries[0].last_access = dir_entries[0].creation_date;
    dir_entries[0].last_mod = dir_entries[0].creation_date;
    dir_entries[0].isDirectory = TRUE;
    dir_entries[0].free_entries = MAX_ENTRIES - 2; // last item is saved for extended
    dir_entries[0].size = bytes_needed;
    set_used(blocks_needed, (dir_entries[0].data_locations));
    dir_entries[0].starting_bock = dir_entries[0].data_locations[0];

    if (parent == NULL)
    {
        strcpy(dir_entries[1].name, "..");
        dir_entries[1].creation_date = dir_entries[0].creation_date;
        dir_entries[1].last_access = dir_entries[0].creation_date;
        dir_entries[1].extended = dir_entries[0].extended;
        dir_entries[1].last_mod = dir_entries[0].creation_date;
        dir_entries[1].isDirectory = dir_entries[0].isDirectory;
        dir_entries[1].size = dir_entries[0].size;
        dir_entries[1].free_entries = dir_entries[0].free_entries;
        for (int i = 0; i < blocks_needed; i++)
        {
            dir_entries[1].data_locations[i] = dir_entries[0].data_locations[i];
        }
    }
```


```
    else
    {
        dir_entries[1].creation_date = parent->creation_date;
        dir_entries[1].last_access = parent->creation_date;
        dir_entries[1].last_mod = parent->creation_date;
        dir_entries[1].isDirectory = parent->isDirectory;
        dir_entries[1].size = parent->size;
        dir_entries[1].extended = parent->extended;
        dir_entries[1].starting_bock = dir_entries[0].data_locations[0];

        for (int i = 0; i < blocks_needed; i++)
        {
            dir_entries[1].data_locations[i] = parent->data_locations[i];
        }
    }

    LBAwrite(dir_entries, blocks_needed, dir_entries[0].data_locations[0]);

    DirectoryEntry *temp = malloc(sizeof(DirectoryEntry));
    int temp2 = dir_entries[0].data_locations[0];
    for (int j = 0; j < MAX_ENTRIES; j++)
    {
        *temp = dir_entries[j];

        LBAwrite(temp, 1, temp2);
        temp2++;
    }

    return dir_entries[0].data_locations[0];
}
```

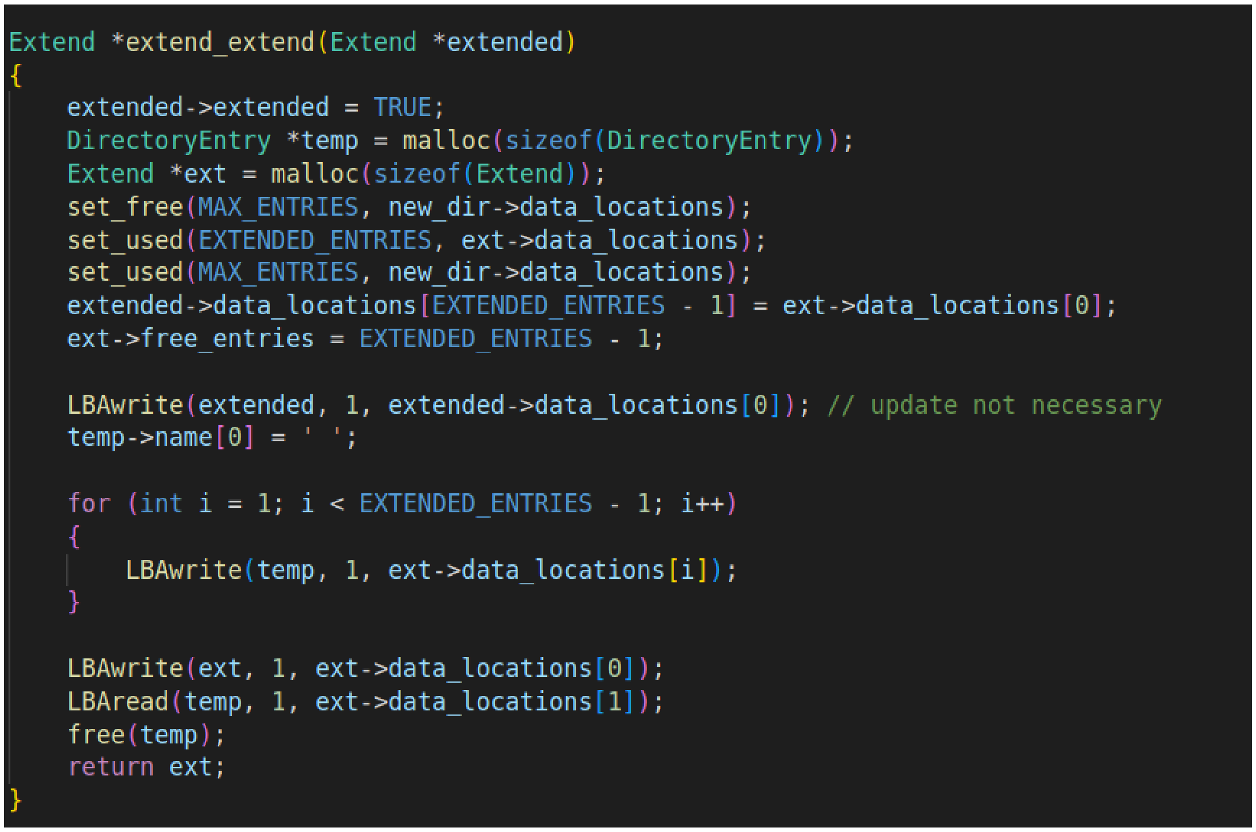
Overall, this code initializes the root directory of a file system by creating a directory entry array, setting its values, and writing it to disk.

We experienced a lot of issues whilst trying to extend our file system project, due to the fact that we are limited with our data_locations array, which only takes up a set amount of entries. We spent a long time trying to figure out the best and most efficient way of extending the file system and ended up creating a new struct called “Extend”. This struct contains a new array of data locations which allows us to create more directories/files.

We ran into an issue whilst trying to link our original directory to the extended one and we solved this by making the last data location of the directory point to the extended ones. The code we used to fo this:

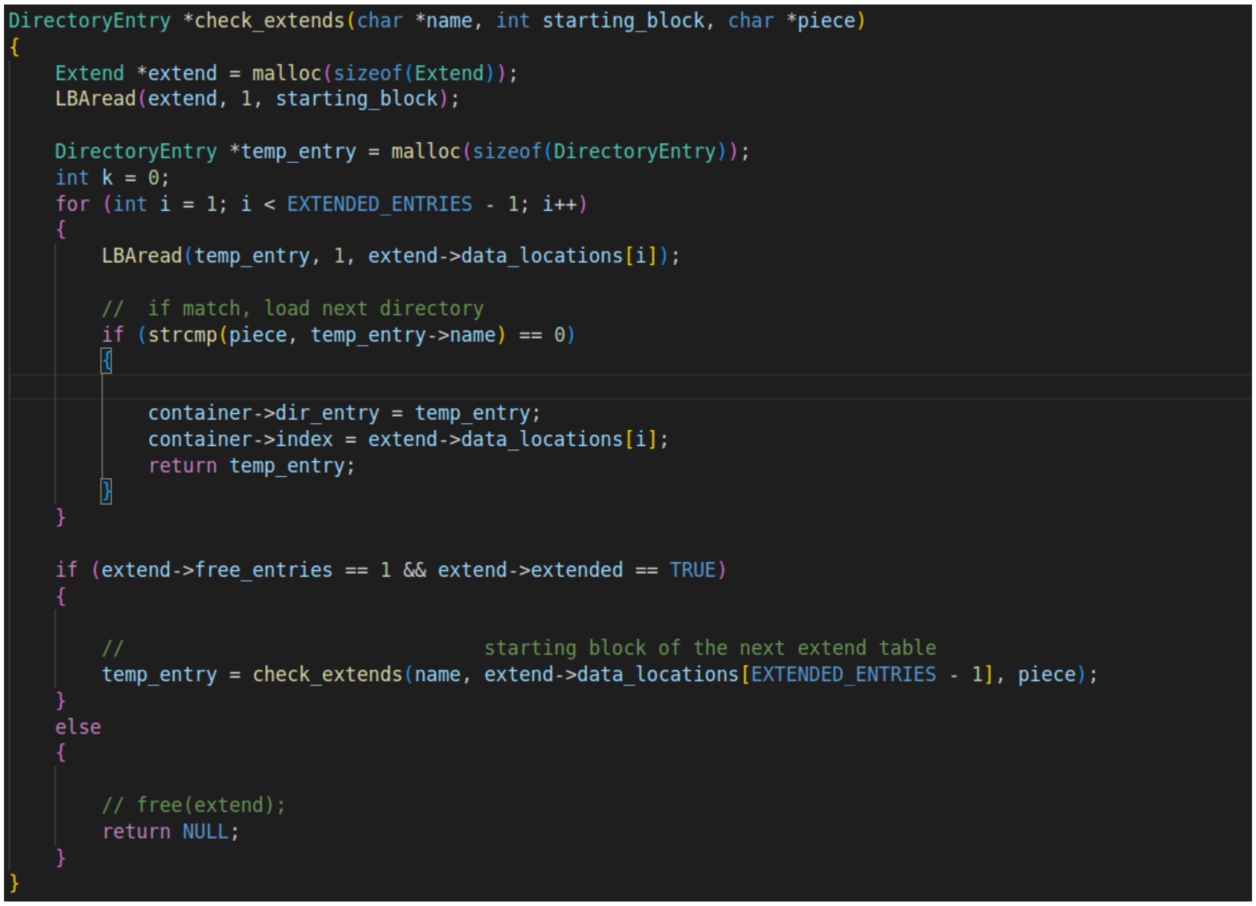
```
dir_entry->data_locations[MAX_ENTRIES - 1] = extended->data_locations[0];
```

We also ran into an issue when we were trying to extend what has already been extended. We solved this issue by creating another function called "extend_extended" which extends what has already been extended. This is a recursive call that keeps happening as soon as we fill up the maximum number of allowed entries.



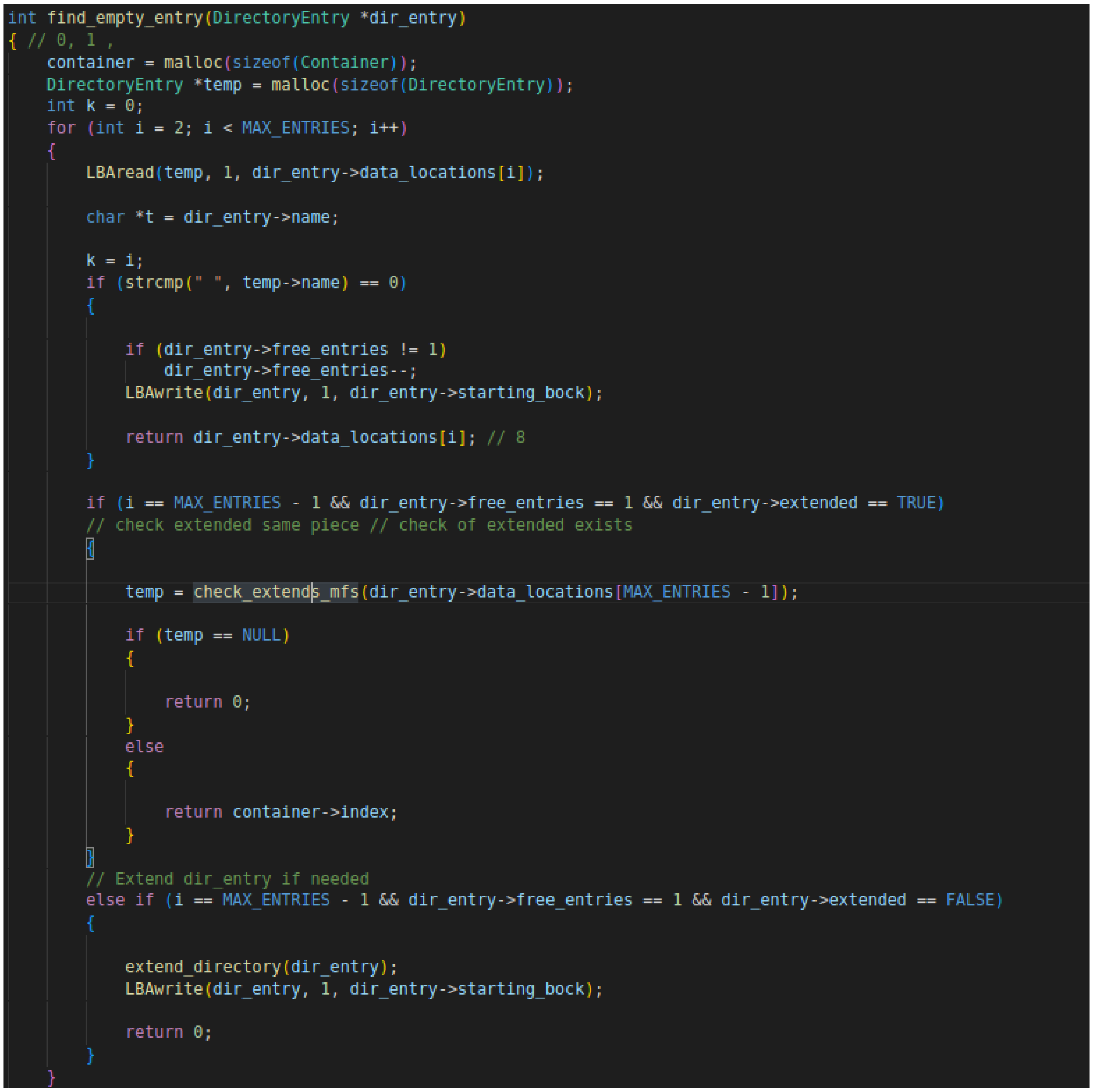
```c
Extend *extend_extend(Extend *extended)
{
    extended->extended = TRUE;
    DirectoryEntry *temp = malloc(sizeof(DirectoryEntry));
    Extend *ext = malloc(sizeof(Extend));
    set_free(MAX_ENTRIES, new_dir->data_locations);
    set_used(EXTENDED_ENTRIES, ext->data_locations);
    set_used(MAX_ENTRIES, new_dir->data_locations);
    extended->data_locations[EXTENDED_ENTRIES - 1] = ext->data_locations[0];
    ext->free_entries = EXTENDED_ENTRIES - 1;

    LBAwrite(extended, 1, extended->data_locations[0]); // update not necessary
    temp->name[0] = ' ';

    for (int i = 1; i < EXTENDED_ENTRIES - 1; i++)
    {
        LBAwrite(temp, 1, ext->data_locations[i]);
    }

    LBAwrite(ext, 1, ext->data_locations[0]);
    LBAread(temp, 1, ext->data_locations[1]);
    free(temp);
    return ext;
}
```

We had issues with parsepath, as we did not know how to loop through the directory entries in the extended version. We solved this by making a recursive call to a function called check_extends, which goes through the extended data locations to find whatever path we passed in.



```c
DirectoryEntry *check_extends(char *name, int starting_block, char *piece)
{
    Extend *extend = malloc(sizeof(Extend));
    LBAread(extend, 1, starting_block);

    DirectoryEntry *temp_entry = malloc(sizeof(DirectoryEntry));
    int k = 0;
    for (int i = 1; i < EXTENDED_ENTRIES - 1; i++)
    {
        LBAread(temp_entry, 1, extend->data_locations[i]);

        //  if match, load next directory
        if (strcmp(piece, temp_entry->name) == 0)
        {

            container->dir_entry = temp_entry;
            container->index = extend->data_locations[i];
            return temp_entry;
        }
    }

    if (extend->free_entries == 1 && extend->extended == TRUE)
    {

        //                             starting block of the next extend table
        temp_entry = check_extends(name, extend->data_locations[EXTENDED_ENTRIES - 1], piece);
    }
    else
    {

        // free(extend);
        return NULL;
    }
}
```

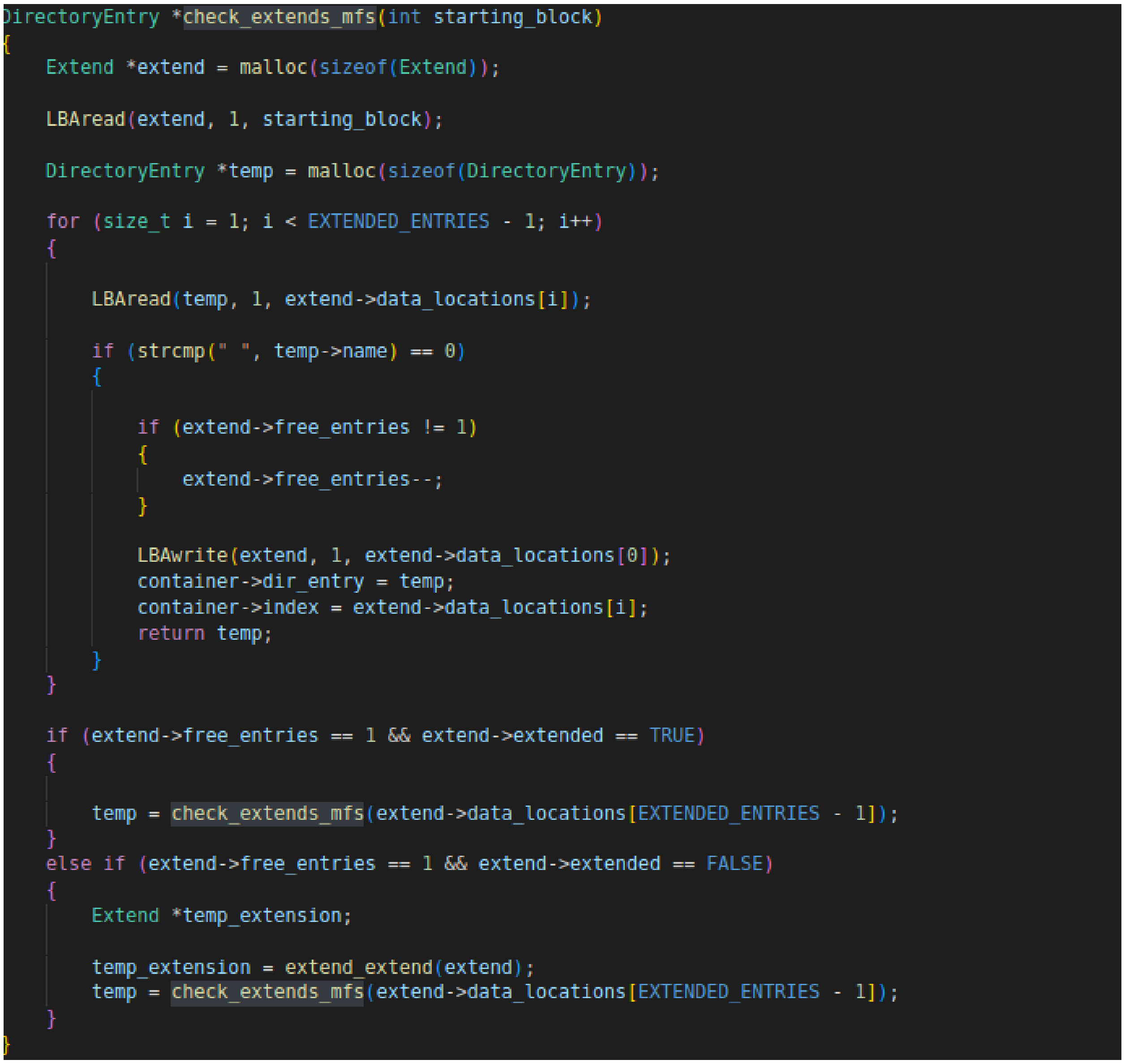
Making a directory was challenging due to the fact that we had to loop through the data locations of the parents directory we wanted to create our new directory inside of, we solved this issue by using multiple loops that go through the data locations of the directory until they find a directory that has a space as the first character in its name, which indicated that that directory is free to use and fill up with whatever we are creating. We called this function to find an empty entry and it returns whichever block is free to write to.



```
int find_empty_entry(DirectoryEntry *dir_entry)
{ // 0, 1 ,
    container = malloc(sizeof(Container));
    DirectoryEntry *temp = malloc(sizeof(DirectoryEntry));
    int k = 0;
    for (int i = 2; i < MAX_ENTRIES; i++)
    {
        LBAread(temp, 1, dir_entry->data_locations[i]);

        char *t = dir_entry->name;

        k = i;
        if (strcmp(" ", temp->name) == 0)
        {

            if (dir_entry->free_entries != 1)
                dir_entry->free_entries--;
            LBAwrite(dir_entry, 1, dir_entry->starting_bock);

            return dir_entry->data_locations[i]; // 8
        }

        if (i == MAX_ENTRIES - 1 && dir_entry->free_entries == 1 && dir_entry->extended == TRUE)
        // check extended same piece // check of extended exists
        {

            temp = check_extends_mfs(dir_entry->data_locations[MAX_ENTRIES - 1]);

            if (temp == NULL)
            {

                return 0;
            }
            else
            {

                return container->index;
            }
        }
        // Extend dir_entry if needed
        else if (i == MAX_ENTRIES - 1 && dir_entry->free_entries == 1 && dir_entry->extended == FALSE)
        {

            extend_directory(dir_entry);
            LBAwrite(dir_entry, 1, dir_entry->starting_bock);

            return 0;
        }
    }
```

We encountered another issue with this find empty entry function and it was that we couldn’t find a way to also loop through the extended data locations and find a free block to write to. We solved this by reimplementing the check_extends function we used in parse path into our mkdir function in a way that loops through the extended entries until it finds a

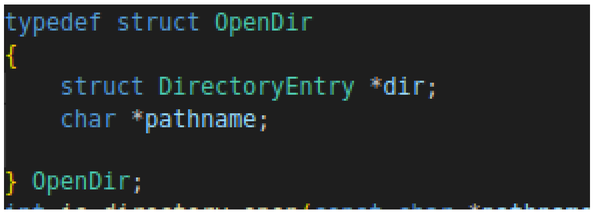
directory that has a space as its first character in the name.



```
DirectoryEntry *check_extends_mfs(int starting_block)
{
    Extend *extend = malloc(sizeof(Extend));

    LBAread(extend, 1, starting_block);

    DirectoryEntry *temp = malloc(sizeof(DirectoryEntry));

    for (size_t i = 1; i < EXTENDED_ENTRIES - 1; i++)
    {

        LBAread(temp, 1, extend->data_locations[i]);

        if (strcmp(" ", temp->name) == 0)
        {

            if (extend->free_entries != 1)
            {
                extend->free_entries--;
            }

            LBAwrite(extend, 1, extend->data_locations[0]);
            container->dir_entry = temp;
            container->index = extend->data_locations[i];
            return temp;
        }
    }

    if (extend->free_entries == 1 && extend->extended == TRUE)
    {

        temp = check_extends_mfs(extend->data_locations[EXTENDED_ENTRIES - 1]);
    }
    else if (extend->free_entries == 1 && extend->extended == FALSE)
    {
        Extend *temp_extension;

        temp_extension = extend_extend(extend);
        temp = check_extends_mfs(extend->data_locations[EXTENDED_ENTRIES - 1]);
    }
}
```

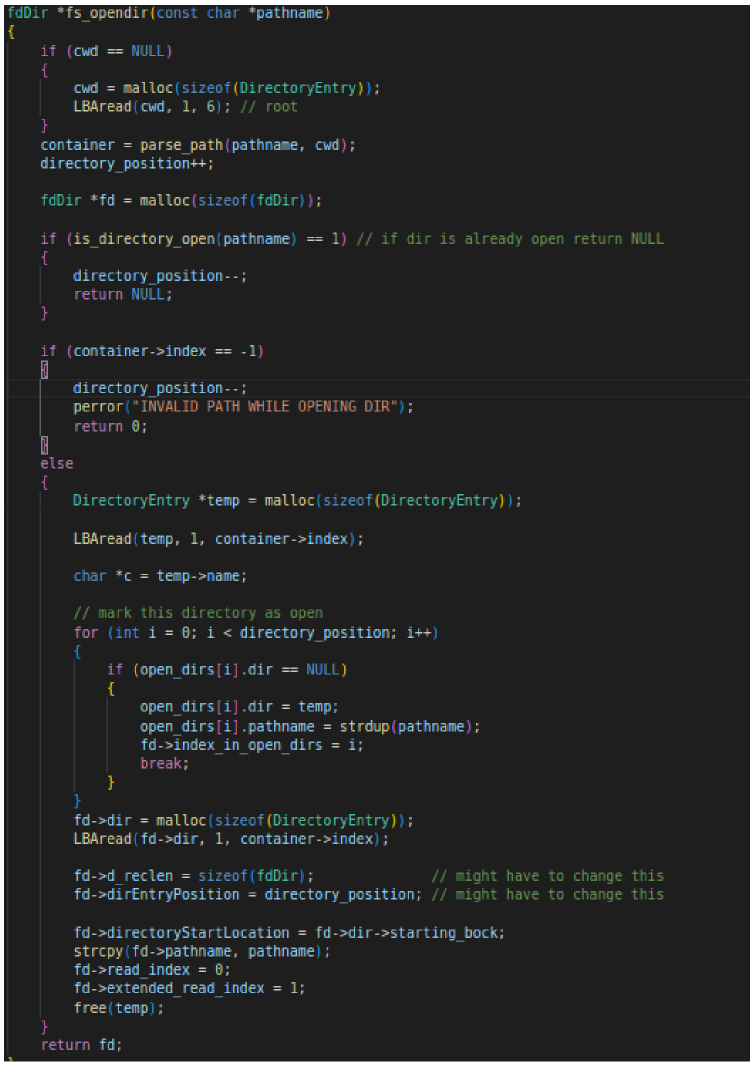
We ran into an issue whilst trying to implement the openDir function. The issue was that we could not manage to keep track of whichever directory was already opened or not. We solved this by creating a new array of structs called open_dirs that contains all the information about whichever directories are open at that moment. This was also helpful with implementing close dir because all we had to do was remove that directory from the array of open_dirs.

The struct of open_dirs:



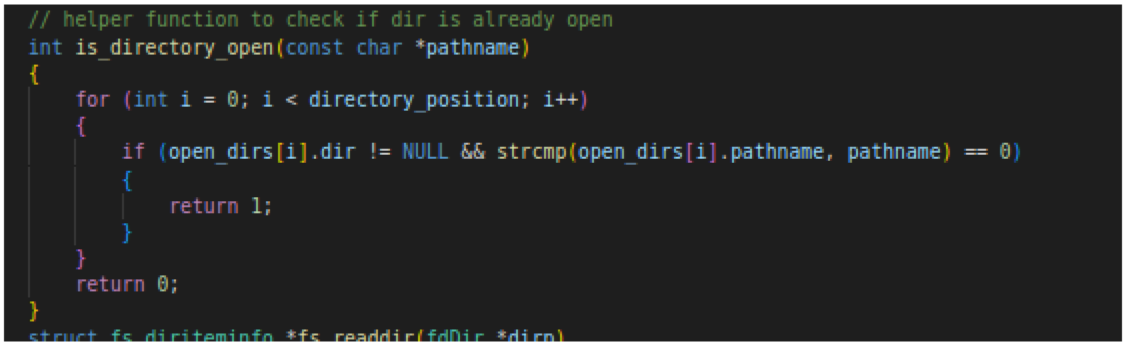
```
typedef struct OpenDir
{
    struct DirectoryEntry *dir;
    char *pathname;

} OpenDir;
```

Implementation of open dir:



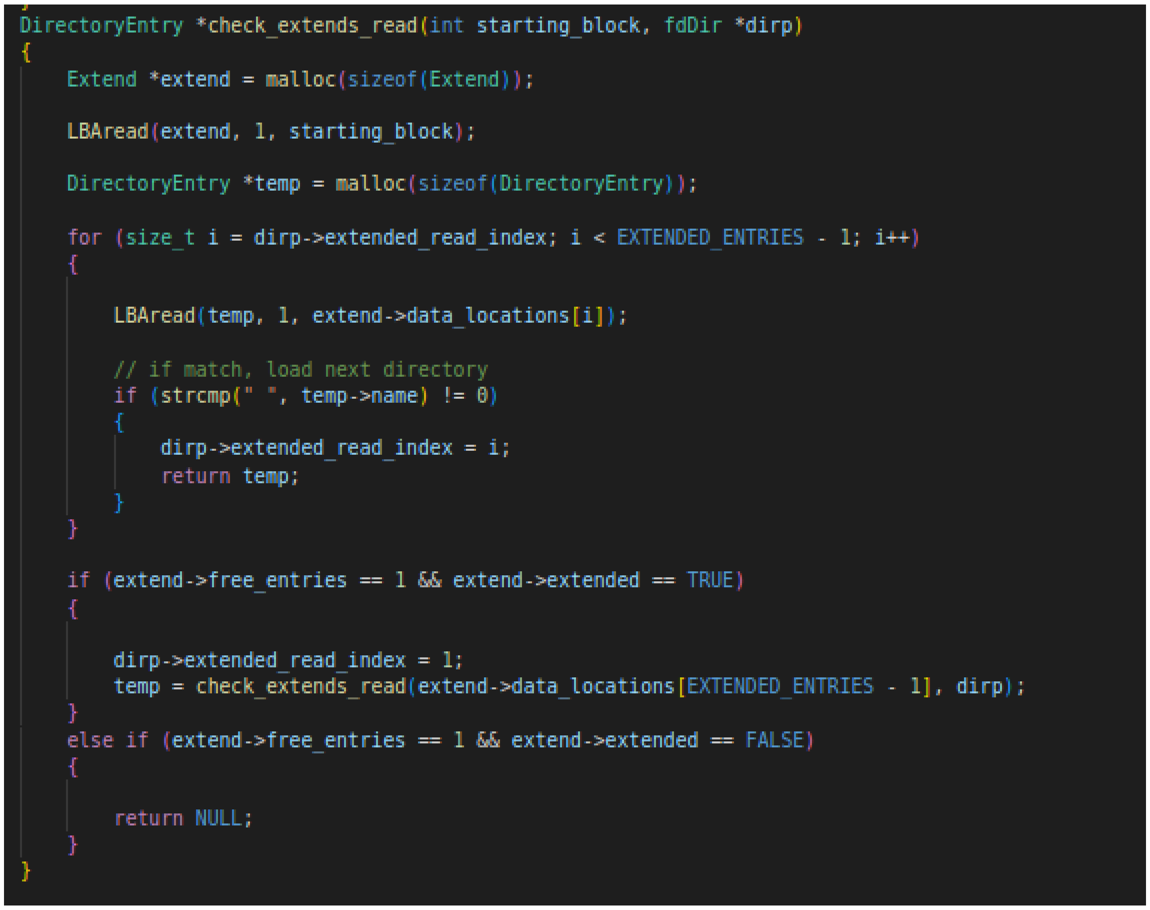
```
fdDir *fs_opendir(const char *pathname)
{
    if (cwd == NULL)
    {
        cwd = malloc(sizeof(DirectoryEntry));
        LBAread(cwd, 1, 6); // root
    }
    container = parse_path(pathname, cwd);
    directory_position++;

    fdDir *fd = malloc(sizeof(fdDir));

    if (is_directory_open(pathname) == 1) // if dir is already open return NULL
    {
        directory_position--;
        return NULL;
    }

    if (container->index == -1)
    {
        directory_position--;
        perror("INVALID PATH WHILE OPENING DIR");
        return 0;
    }
    else
    {
        DirectoryEntry *temp = malloc(sizeof(DirectoryEntry));

        LBAread(temp, 1, container->index);

        char *c = temp->name;

        // mark this directory as open
        for (int i = 0; i < directory_position; i++)
        {
            if (open_dirs[i].dir == NULL)
            {
                open_dirs[i].dir = temp;
                open_dirs[i].pathname = strdup(pathname);
                fd->index_in_open_dirs = i;
                break;
            }
        }
        fd->dir = malloc(sizeof(DirectoryEntry));
        LBAread(fd->dir, 1, container->index);

        fd->d_reclen = sizeof(fdDir);              // might have to change this
        fd->dirEntryPosition = directory_position; // might have to change this

        fd->directoryStartLocation = fd->dir->starting_bock;
        strcpy(fd->pathname, pathname);
        fd->read_index = 0;
        fd->extended_read_index = 1;
        free(temp);
    }
    return fd;
```

implementation of is_directory_open:



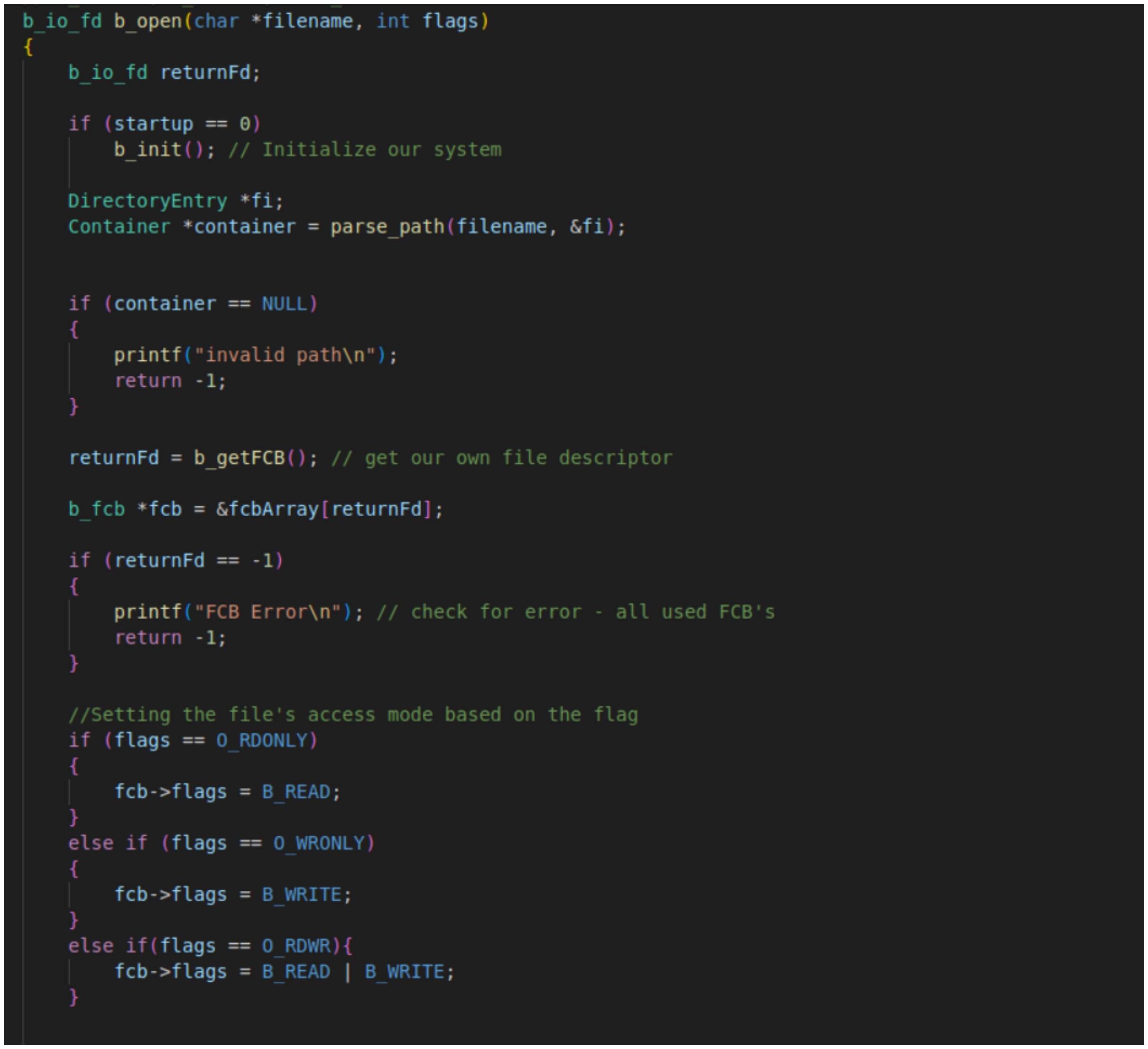
```
// helper function to check if dir is already open
int is_directory_open(const char *pathname)
{
    for (int i = 0; i < directory_position; i++)
    {
        if (open_dirs[i].dir != NULL && strcmp(open_dirs[i].pathname, pathname) == 0)
        {
            return 1;
        }
    }
    return 0;
}
struct fs_diriteminfo *fs_readdir(fdDir *dirp)
```

Reading a directory was a challenge for us because we had to figure out a way to go through our extended data locations in order to try and find a directory entry to read. We fixed this by reimplementing the check_extends function once again to match what we need when reading a directory.

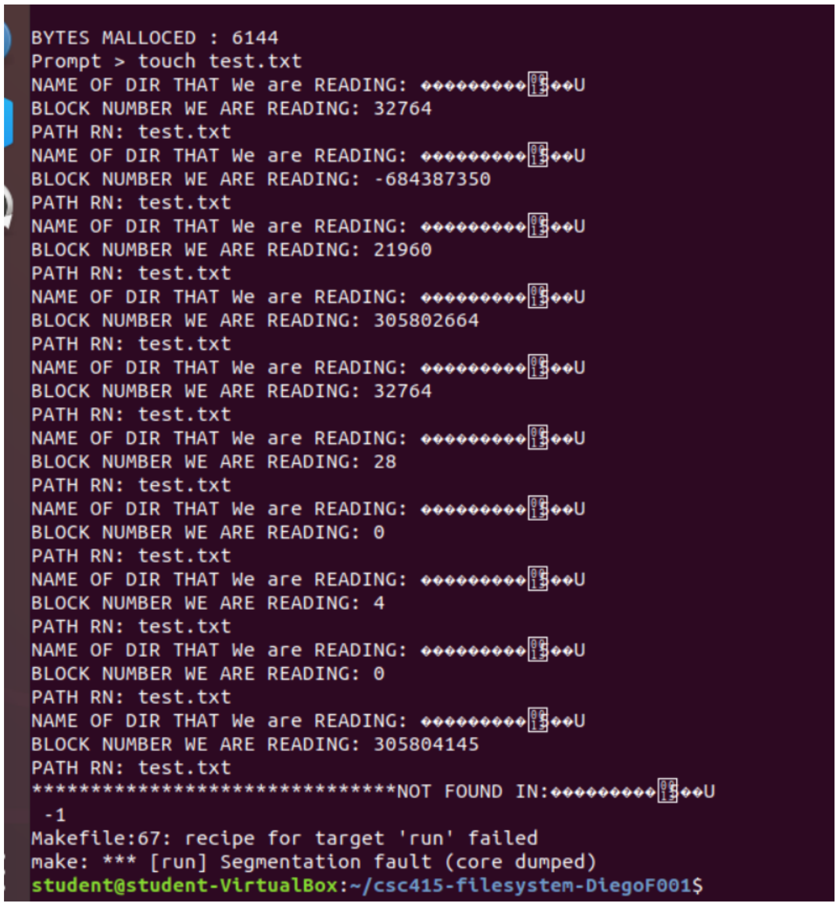


```
DirectoryEntry *check_extends_read(int starting_block, fdDir *dirp)
{
    Extend *extend = malloc(sizeof(Extend));

    LBAread(extend, 1, starting_block);

    DirectoryEntry *temp = malloc(sizeof(DirectoryEntry));

    for (size_t i = dirp->extended_read_index; i < EXTENDED_ENTRIES - 1; i++)
    {

        LBAread(temp, 1, extend->data_locations[i]);

        // if match, load next directory
        if (strcmp(" ", temp->name) != 0)
        {
            dirp->extended_read_index = i;
            return temp;
        }
    }

    if (extend->free_entries == 1 && extend->extended == TRUE)
    {

        dirp->extended_read_index = 1;
        temp = check_extends_read(extend->data_locations[EXTENDED_ENTRIES - 1], dirp);
    }
    else if (extend->free_entries == 1 && extend->extended == FALSE)
    {

        return NULL;
    }
}
```

We started implementing the b_io.c functions pretty late so we were not able to complete the implementations. As of right now, the function below doesn't correctly handle the O_CREAT and O_TRUNCATE flags to create a file when it doesn't exist. This also prevents the other file operation functions from working as they should.



```
b_io_fd b_open(char *filename, int flags)
{
    b_io_fd returnFd;

    if (startup == 0)
        b_init(); // Initialize our system

    DirectoryEntry *fi;
    Container *container = parse_path(filename, &fi);


    if (container == NULL)
    {
        printf("invalid path\n");
        return -1;
    }

    returnFd = b_getFCB(); // get our own file descriptor

    b_fcb *fcb = &fcbArray[returnFd];

    if (returnFd == -1)
    {
        printf("FCB Error\n"); // check for error - all used FCB's
        return -1;
    }

    //Setting the file's access mode based on the flag
    if (flags == O_RDONLY)
    {
        fcb->flags = B_READ;
    }
    else if (flags == O_WRONLY)
    {
        fcb->flags = B_WRITE;
    }
    else if(flags == O_RDWR){
        fcb->flags = B_READ | B_WRITE;
    }
```

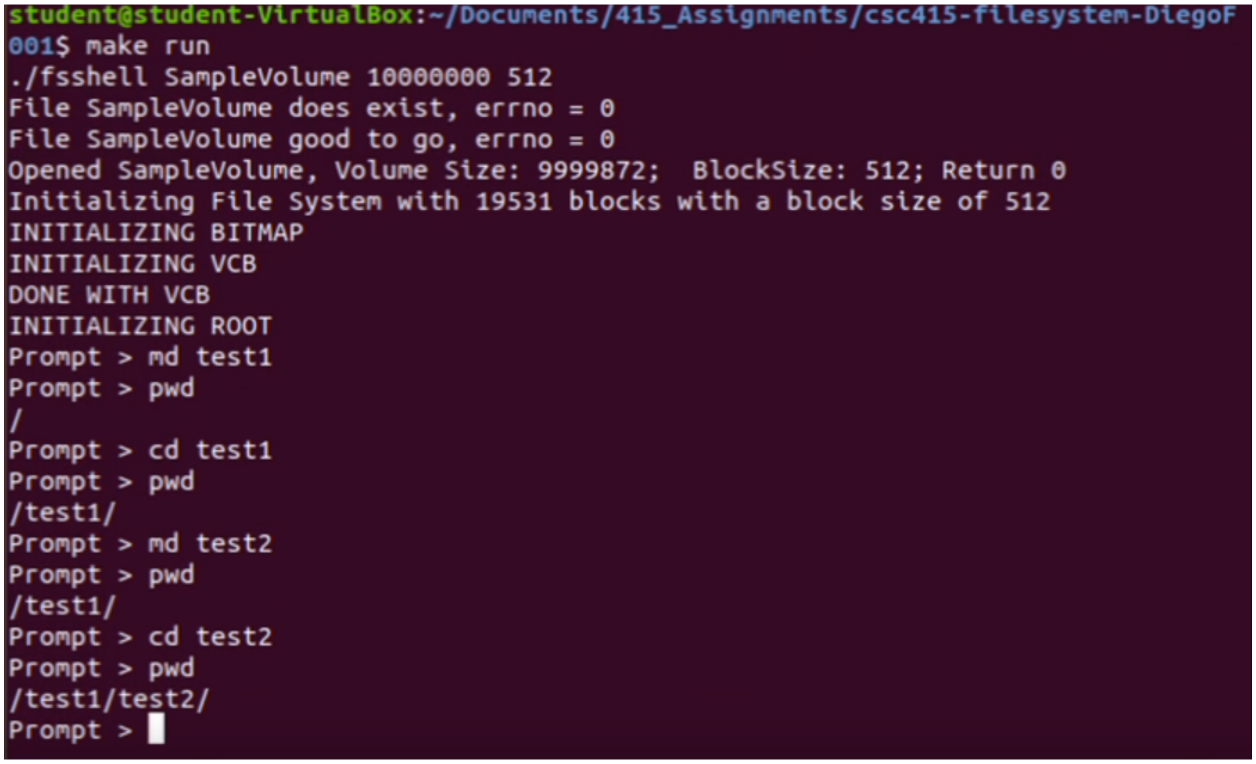


```
BYTES MALLOCED : 6144
Prompt > touch test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 32764
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: -684387350
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 21960
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 305802664
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 32764
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 28
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 0
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 4
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 0
PATH RN: test.txt
NAME OF DIR THAT We are READING: ��������������U
BLOCK NUMBER WE ARE READING: 305804145
PATH RN: test.txt
*******************************NOT FOUND IN:��������������U
 -1
Makefile:67: recipe for target 'run' failed
make: *** [run] Segmentation fault (core dumped)
student@student-VirtualBox:~/csc415-filesystem-DiegoF001$
```

This is the error we got from using the touch command to test b_open. The program tries to read the file to check if it exists but it just ends up reading the directory it is expected to be in so it prints out random characters because it's accessing uninitialized memory.

Mohammad Dahbour 921246050
Diego Flores 920372463
Kemi Adebisi 921140633
Github: DiegoF001

SCREENSHOTS OF WORKING COMMANDS:



```
student@student-VirtualBox:~/Documents/415_Assignments/csc415-filesystem-DiegoF
001$ 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
INITIALIZING BITMAP
INITIALIZING VCB
DONE WITH VCB
INITIALIZING ROOT
Prompt > md test1
Prompt > pwd
/
Prompt > cd test1
Prompt > pwd
/test1/
Prompt > md test2
Prompt > pwd
/test1/
Prompt > cd test2
Prompt > pwd
/test1/test2/
Prompt > 
```