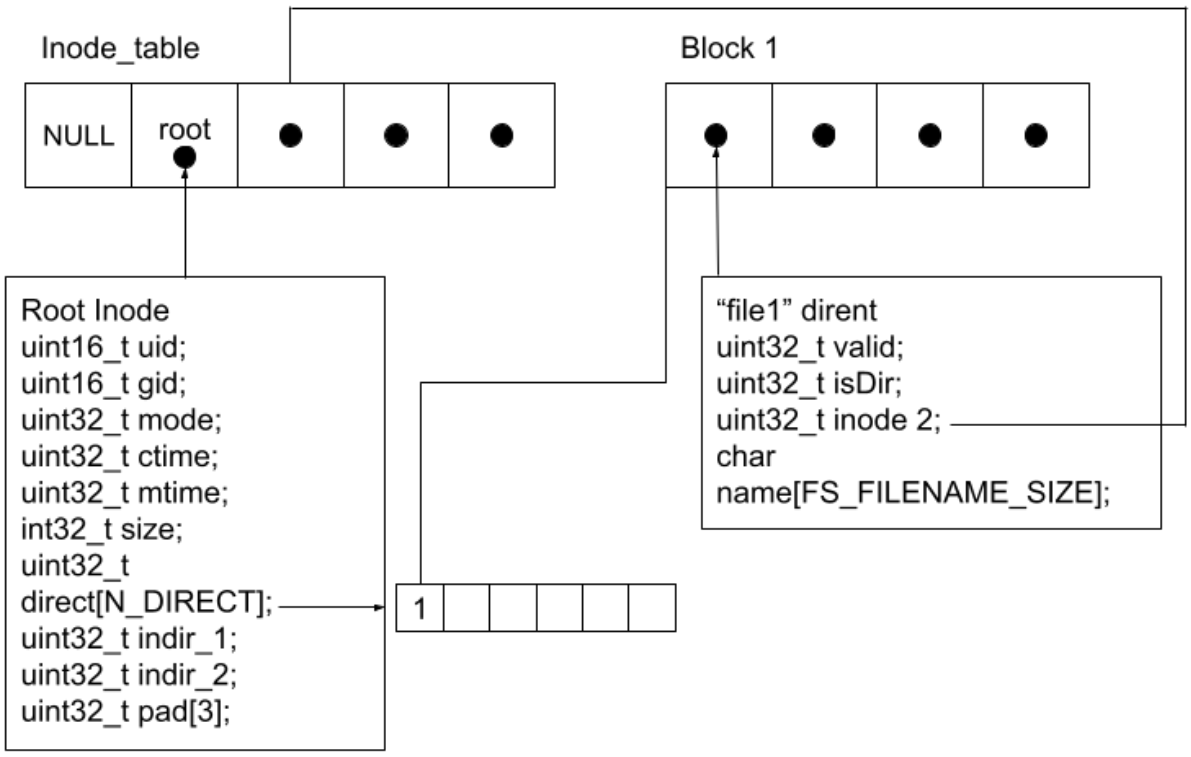
Design and implementation of the file system

When writing implementing fsx492, we had to get information from the blkdev. It was already initialized so we had to be able to extract all the information we can from it using block pointer arithmetic. For our implementation of our filesystem we stuck close to the implementation of a linux filesystem. As directed in the assignment specifications.



Simple figure to visualize structure

Our initializing variables include:

- **struct fs_super fsx**
  - the superblock read off of the first block in the disk
- **fd_set inode_map**
  - an fd_set that represents the bitmap of inodes that we also read from the disk
- **fd_set block_map**
  - an fd_set that represents the bitmap of data blocks that we also read from the disk
- **struct fs_inode* inode_table**
  - This represents a table of all the available inodes that we read in from the disk
- **Unsigned int err**
  - Our own errno
- **struct fs_inode* root**
  - A pointer to the inode table of the root directory’s inode

The purpose of these variables is to give us a localized version of the resources available to us on the disk thus giving us easier access.

Main Helper Functions

For this project we made two helper functions where one was in charge of getting us the inode of the path that was passed in and the other was in charge of getting the directory entry, dirent, of the path that was passed in. These functions were named getInodeFromPath and getDirentFromPath respectively. The following is a quick rundown of the functions. They are both fairly similar.

```
struct fs_inode *getInodeFromPath(const char *path, struct fs_inode *result)
```

To start off this function tokenizes the path, utilizing strtok() to get the number of tokens in our path. Then we run through the tokens again doing the actual work. The purpose of the two calls to tokenize the path is to allow us to check if the intermediate steps of the path are directories, but the last element could be a directory or a file so that was how we handled that. On to the main loop, we loop through the tokens, and read the first element of the direct array to find the block where the dirents of our current location are stored. Am fs_inode* called current is used to keep track of our current inode as we traverse the path. It is initialized to be the root inode. After finding the block number we begin another while loop to read the dirents inside of the block. To do this we use image.read to read in the block into our void* block array where our first block is the block number found in the direct[0] of current. While checking the dirents and storing them into a directory entry struct called temp. We compare the names to see if it exists, as well as checking if that entry is a directory and valid. If it isn't a directory then we set err and free and return NULL.If this check goes through we move our current pointer to the next inode, found by checking the temp.inode and using some pointer arithmetic.We set a found variable to 1 and this is used to signal to the loop that we don't need to keep checking that block since we found what we're looking for. If we found the last token in the path then we can just return right there. Otherwise if the token was not found in that block we know that a directory can only have 32 entries (1 block) so we know that the token in the path was not found so we set err no and free and return NULL. Building this helper function was essential and helping us understand the structure of fs_inode(s), fs_dirent(s) and the blocks.

```
struct fs_dirent getDirentFromPath(const char *path, struct fs_dirent result)
```

This function works almost exactly the same way as getInodeFromPath except it does not use a temp to store the dirents from the block it stores it directly into result and returns result when finished or sets the inode number of result to 0 if path was not found.

Functions Implemented

```
static int image_num_blocks(struct blkdev *dev)
```

The function needed us to counter the number of blocks within the device. In order to complete this task, we needed to our image dev struct, dereference it and point it to the private block device state from the blkdev header file. With this being done, we were then able to have an imager point to the number of blocks on the device given.

```
static int image_read(struct blkdev *dev, int first_blk, int nblks, void *buf)
```

In order to implement image_read, we needed to implement the fcntl macro on the imager fd in order to get the file descriptor of the flags. When this macro is used, it will throw and error and display a message in order to clarify unavailability. We then use assert in order to return an error if 0 is present. We set res = to pread which will essentially read from the set offset which is the overall block size in bytes * the index of the block to start reading from for us. On success, the number of bytes read is returned which grants us the functionality of reading the block device at the given block index.

```
static int image_write(struct blkdev *dev, int first_blk, int nblks, void *buf)
```

Similar to image_read only in this sense, we are considering the utilization of pwrite within our function in order to write the offset that is displayed after the process is run. Primarily writing to the file descriptor at the given offset.

```
static int image_flush(struct blkdev *dev, int first_blk, int nblks)
```

image _flush as mentioned is supposed to flush the block device. For starters, we check when the block device is null, and check if the block device is already closed. We then consider if the buffer considers the correct amount of zeros to store within the actual device by filling the memory with zeros to set it to null or in our case 0. After checking that the buffer was properly initialized or not, we then ensure that there is no failure within the index of the blocks that we will eventually start flushing. We then set res = to pwrite which takes those zeros from the buffer and inputs them into the image file descriptor. After freeing the buffer finally, we perform a check on res to see whether the correct number of zeros could be written to the image and if not we throw a failure at the end of the program. Though if all goes well, we return a success at the end.

```
static void image_close(struct blkdev *dev)
```

Just needed to close the file descriptor as well as set an error if the device has already been closed. After closing though we want to ensure that nothing is open. After, we free imager and the block device, we set both to null to ensure that the device is closed.

```
static int fs_getattr(const char *path, struct stat *sb)
```

Getattribute utilizes the getInodeFromPath function that we created and sets current equal to it, having the file path as a parameter. We first want to build the stat struct in sb from the path that is provided. If not successful, we throw the errors that are required for the fs_getattr function. We then set the pointer to stat struct data fields equal to the fs_inode struct data fields in order to obtain whatever path that we are searching for attributes back to the user.

```
static int fs_opendir(const char *path, struct fuse_file_info *fi)
```

The way we implemented opendir, utilized the getDirentFromPath function in order to point to the directory entry. In order to save information about the current directory that we have just opened, as given, we used fi -> fh in order to set the temporary (directory stream) inode (the current opened directory / entry node) to that saved information. When handling errors, if temp.inode == 0 as seen in the code, specified errors that were given will be enacted.

```
static int fs_readdir(const char *path, void *ptr, fuse_fill_dir_t filler,
                      off_t offset, struct fuse_file_info *fi)
```

Readdir utilizes the fi -> fh functionality from opendir and is essentially a continuation of the opendir process. Within the function we want to look at the first directory entry of the array in order to then loop through the directory entries in the block. Then once we figure whether the directory entries are empty or not determines whether we call the filler function or not. When handling errors, if current == Null as seen in the code, specified errors that were given will be enacted.

```
static int fs_mknod(const char *path, mode_t mode, dev_t dev)
```

This function also tokenizes the path and gets the length, it then builds the path excluding the last token. It then passes the built up into getinodefrompath and returns an error based on a problem. Once you get the pointer to the inode, it then checks the direct array and gets the block with all of the dirents. It compares the name with all of the other dirents to make sure that an entry does not already exist for it. It stops at the first empty dirent and changes the values of isDir and valid to 1 to make the entry valid. It then iterates through the inode_map and identifies the fist empty inode, marks it as 1, then it gets the index of the empty inode and passes that number to the empty dirent entry. A new inode is created and initialized and entered in at the inode region in the respective empty index. Then the function updates and writes it all to the disk.

```
static int fs_chmod(const char *path, mode_t mode)
```

Chmod is similar to a lot of the previous functions. It allocated memory for an inode then it called the get inode from path. From here the function takes this inode and changes the mode to the one passed into the function. current->mode = mode. It then calls the image_write function at the start block and copies over SZ_INODE_REGION blocks. After that the function returns and completes.

```
static int fs_statfs(const char *path, struct statvfs *st)
```

Design and implementation of testing

Testing was going to be the most difficult part of the project as there are many factors to test what was going on. For our implementation the majority of the testing was done in GDB. Although we didn't get all the functions to work, we spent the majority of the time making sure the ones that we did write worked well. In gdb we would set breakpoints at functions that we knew would be called. We would then run the fuse filesystem in command line mode. The first function we tested was fs_getattr and we got to it by calling the cd command. Once this breakpoint was hit, we would have the file loaded and now we could use the call function to call any function we wanted to test. The backbone of all the implementations was getInodefrompath, and this one required the most testing. We would go through it line by line inspecting the pointers in memory every time they were changed. This allowed us to identify the problem at hand. A big problem that we encountered was the return pointer pointing to a block in the memory. To diagnose this problem we dumped the entire disk into the memory. Then we looked for specific pattern matching of characters within the entire disk. This would give us an address which we would then modulo 1024(the block size) which told us that we were copying the wrong block and it was off by one.

When testing functions that resulted in any type of error or segfault, we would Backtrace and see the values of the called functions. Inspecting all of these values would inform us what is not implemented correctly and at what line it was called. Whether it was a problem in image.c or any of the helper functions. We also ran the command p errno to read the error number and cross referencing it with default linux error numbers.

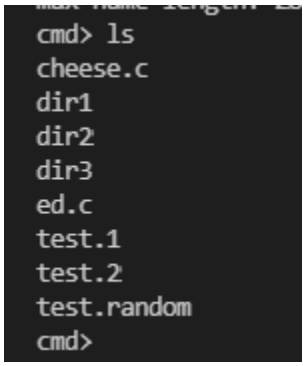
```
blkdev.h  fs.c  fsx492  fsx492.h  image.c  image.h  main.c  Makefile  README.m
root@debian:/home/student/cs492-course-project-coyster12# gdb fsx492
fsx492    fsx492.h
root@debian:/home/student/cs492-course-project-coyster12# gdb fsx492
```

- After Making

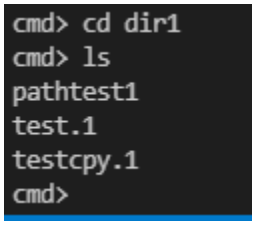
```
Reading symbols from fsx492...done.
(gdb) r -cmdline -image test/fsx492.img /tmp/use/
Starting program: /home/student/cs492-course-project-coyster12
```

- Used run the program, and allowed us to explore our FUSE implementation

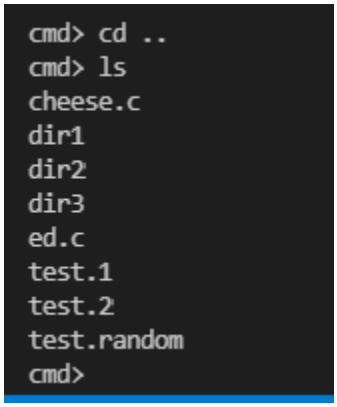
Screenshots of working implementations:



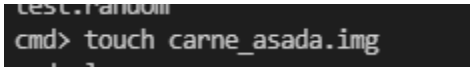
- Shows ls working within our FUSE



- Change directory working in the FUSE going into dir1



- Going backwards in the FUSE back to the root directory



- Creating an image file *carne_asada.img*

```
cmd> ls
carne_asada.img
cheese.c
dir1
dir2
dir3
ed.c
test.1
test.2
test.random
cmd>
```

- Performing ls in order to see the newly created *carne_asada* file

```
cmd> statfs
block size: 1024
no. blocks: 1024
avail blocks: 0
max name length: 28
cmd>
```

- Checking to see if statfs implementation works

```
cmd> stat carne_asada.img
    0 -rwxrwxrwx  2    0    0        0 Wed Dec 31 19:00:00 1969 carne_asada.img
cmd>
```

- Performing a stat on our previously created *carne_asada.img* file and getting the results back

Group information

Members : Dave Taveras, Eduardo Mena, Sequoy Young-Garcia

Contribution of each member: Each group member proactively contributed by formulating ideas in order to come up for solutions for each section of the code in both image.c, fs.c, and even when testing. Communication with each member was very transparent, exclaiming the ideas of updating the github when changes were made as a group, and even elaborating on what each new part of the code implemented means to those that do not have a clear understanding. If confusion was present as mentioned, we would stop to enhance our knowledge in order to gain a better understanding of the current task. The utilization of live-share on vscode, made the collaborative experience better for everyone as well. We were able to work efficiently and simultaneously with this extension of vscode. Another point of emphasis was the utilization of GDB when testing our functions. Given that Ed and Sequoy had prior knowledge with GDB from CS-576, it made testing manageable and granted the ability for Dave to learn the fundamentals of the debugger. The project took a toll on the group due to the amount of questions we had for the Professor and Ta's, which subjected us to only implement a minimal amount of functions for this project. If another week was provided, we possibly would have figured most of the functions.