Chris Bechter
CIS3207

Lab04 File System Documentation

1.1 Directory Structure

The structure I'm planning, and have already begun coding for, in hopes of, is the File Allocation Structure. I plan to also use the logical directory of files and folders. As discussed in class, I will be using the FAT (File Allocation Table) file system style as well. This will all stem from a Fat - Root Directory - Data Blocks style filesystem. For the storage this will take (size of int * (size of the drive over the size of a block)) bytes in the drive. The first entry in the FAT system would be initialized -1 although the rest would begin at 0. To indicate the end of the block in a FAT system would require the -1 for the chain and 0 indicates the block hasn't begun yet. The root directory starts with block 0, this means nothing can link back to the root directory, making the block chains discourteous and the block x in the system is made FAT[x].

Considering every piece of data will hold some kind of block address, or own a certain percent of the block, every directory and file will have a block address associated with that address. If the block contains a file, or directory listing, the block may or may not depending on the testing hold one or serval data structs for the said block, all of which will have a size defined. Defining the type of struct may help as well, whether dealing with a file or a directory. These are used with just int values.

2.1 & 3.1 Details of Implementation and Pseudocode

Structs:

Data struct:
    char name[]
    int size, type, begin
This is data struct for size, type, and beginning node

Block struct:
    char file block_size
    data directory block_size
The size of each variable is defined that represents the block file and directory

Drive struct:
    Int fat num_blocks
    block info num_blocks

fptr_file struct:
    data *contents
    int pointer, current
This struct if used for files that will either read or write from or too. The * refers to the files meta data

Functions:

make_file(char *path, int type, drive *dataInfo)
    error handling to check for valid path or if file exists
    find empty slot in dir
    error handling to make sure block has space
    write to dir

open_file(char *path, vdrive x)
    use path to get data from file

```
        make a fptr_file struct for data
        return fptr_struct

read_file(fptr_file *f, int len, vdrive *x)
        copy data till end of block of the buffer
        copy next blocks data into buffer until end of file or len

write_file(fptr_file *f, char *info, int len, vdrive *x)
        fill the block with passed in info
        continue to write all nesscary blocks
        update current block and fptr_file

close_file(fptr_file f)
        free fptr_file struct

remove_file(char *f, vdrive *x)
        reset all blocks to zero
        remove data from dir
```

4.1 Memory Mapping & Testing

I will be using the memory mapping functionality to map the virtual drive of the program, so that the program can basically see the entirety of the logical device on any machine its ran on. This code, which was found mainly in the text book and used in previous labs, should map the drive to the drive struct and let the program be able to both READ and WRITE, protectively, to said virtual drive, while MAP_SHARED will sync the drive in memory to created virtual drive.

```
int f = open("drive_file_name", O_RDWR);
drive *D = mmap(0, sizeof(drive), PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
close(f);
```

After this has been mapped, the program will run its course, and right before end of program, use the unmap function to close the virtual drive.

As for testing purposes, run the drive with different commands and outputs, to make sure the virtual drive was created, make sure the structs are made and handle their functions, and then try out each function and hopefully won't get any errors.