Resources:

- http://www.cs.ucsb.edu/~chris/teaching/cs170/projects/proj5.html
- https://stackoverflow.com/questions/8589425/how-does-fread-really-work
- http://www.tuhs.org/cgi-bin/utree.pl?file=V7
- http://www.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/file.c
- Definition of mounting: http://www.linfo.org/mounting.html
- Mount : http://man7.org/linux/man-pages/man8/mount.8.html
- Shell command 'more': to find the window size, use 'ioctl'

Things to pay attention:

1. Shell needs to verify max filepath
2. If a new file is created while calling f_open, then the 'block_offset' field in the file entry is undefined when there is no file data for this file yet; empty free fd index value is UNDEFINED as well
3. Shell parsed out the mounting point path from the filepath, and replace the mounting point path with '/': for instance if mounted at directory 'AB', then the filepath '/AB/HAHA/123.txt' after parsing by the SHELL would be 'HAHA/123.txt', which assumes that directory 'HAHA' is under the root directory
4. Not used entries in the open file table are set to NULL
5. When write files or directory files, if using indirect access, needs to write the index blocks as well
6. Parse the '.', '..' for filepath in shell based on the current working directory, and always pass the full, complete file path to the file system library funcs
7. Offset within the block is < 512, needs to handle in: write, seek, read, readdir

File_System_Design_Document

********** General Ideas that might be helpful for understanding **********

1. File system represents a way of organizing files and directories, including the specific format and guidelines of processing files (read, write, create, and remove)
2. A file system requires specific library functions (f_write, f_read etc.) to access and process its files, which means a corresponding file system program is required to make use of the library functions and execute upon the file system.
3. File system divides into two parts: file system structure: which is the structure in a specific disk; file system program: the program which contains all the API functions to control and maintain the file system structure.

********** Spec Questions & Answers **********

1. Physical structure of your file system: FAT or inodes?

    a. inodes (following the structure you gave us in the last assignment)
2. Virtual structure of your file system: full VFS or just a library?
    a. Just a library
3. If not full VFS, where does the rest of the stuff go? Please describe in detail the steps your OS will perform when an open call is made, for example.
    a. The rest of the stuff (attributes in the vnode) will go inside of inode struct.
    b. We will store the library functions in the same directory as the DISK image th
    c. e shell uses for initialization The implementation for open call refers to ‘f_open’ below.
4. The file table - data structure to keep track of open files
    a. See the ‘structs and enum’ fields for more information about file table
    b. File table also stores unused file descriptors, the reason for this is that, when a file is closed, the memory for storing this file’s information in the file table is not freed, such that the relative location of other ‘file_table_entry’ instances in the ‘entris’ array will not change, and still can be indexed with the file descriptors even if after a file is closed. (for faster access).
5. Free-space management
    a. Free-inodes are tracked through the ‘next_inode’ field in the ‘inode’ struct; in a used inode, this field will be hold a value of -1
    b. Free blocks are tracked through the free block linked list in the super block field, with the first 4 bytes in each free block indicating the offset for the next free block
6. Block-access within a file
    a. Since the file is opened for access, we can use the ‘block_index’ field in the ‘file_table_entry’ for this file to calculate the position of the current block data we are at, and determine whether we are in direct blocks, indirect blocks, double indirect table, or triple indirect table, as well as the specific index of the current data block in those tables, therefore allowing us to locate the next block for access within this file
7. Directory file structures
    a. See below under ‘Disk Image & Directory file structure’
8. superblock and FAT entry/inode layout (as actual structs)
    a. See below for ‘structs’

********** Disk Image & Directory file structure **********

- ❖ Disk image layout:

BOOT BLOCK
SUPER BLOCK (struct): int size; /* size of blocks in bytes */ int inode_offset; /* offset of inode region in blocks */ int data_offset; /* data region offset in blocks */ int swap_offset; /* swap region offset in blocks */

int free_inode; /* head of free inode list, index */ int free_block; /* head of free block list, index */
INODE REGION
DATA REGION
SWAP REGION

- Directory file structure
  - Each directory has its inode
  - Data stored for a directory file:
    - List of inode indices corresponding to the directories and files inside of this directory
    - Directory file layout:

First 4 bytes: stores the number of files in this directory
The inode index for this directory (represents .)
The inode index for its parent directory (..), if root directory, then points to itself
List of file inode indices

** Global Variables (shell) **
user[] valid_users; //array with super user and normal user values
int is_superuser; // value is 1 when super used logged in, otherwise 0
char* root_dir; // shell needs to store its running directory as its root directory
mounted_disk* disks;
struct* user cur_user;
int disk_fd; // file descriptor for the current disk opened

** Structs and enums **
enum filetype {DIR, REG};
enum fileseek {SSET, SCUR, SEND};
enum permission{SUPER, REGULAR};

/* file system programs maintains a list of mounted directories information, information as stored as the 'mounted_disk' struct */
// not used
~~**struct user** {~~
~~int uid;~~

~~char* username;~~
~~char* password;~~
~~char* current_disk;~~
~~char* cwd; //current working directory~~
~~char* rootdir;~~
~~}~~
//shell?
// changed
~~**struct mounted_disk**{~~
~~int root_inode; // inode index of the root directory of the disk~~
~~char* mp_name; // mounting point absolute filepath~~
~~int blocksize;~~
~~int inode_region_offset;~~
~~int data_region_offset;~~
~~int free_block;~~
~~int free_inode;~~
~~int uid;~~
~~int root_dir_inode_index;~~
~~void* inode_region;~~
~~}~~

```
//shell
struct mounted_disk {
   char mp_name[MAX_NAMELEN]; // mounting point absolute filepath
   int diskfd;
}mounted_disk;

// file system
struct fs_disk{
   int diskfd;
    int uid;
    int data_region_offset; // byte offset
   int rootdir_fd;
   struct inode* root_inode;
   struct superblock sb;
    //char mp_name[MAX_NAMELEN]; // mounting point absolute filepath
    void* inodes;
    //int root_inode; // inode index of the root directory of the disk
    //int blocksize;
    //int inode_region_offset;
```

```
    //int free_block;
    //int free_inode;
    //int root_dir_inode_index;
}fs_disk;
```

```
// inode struct
struct inode {
        char filename[40]; //not absolute path // maybe inside of dirent
        int inode_index; // the index number for this inode
        // (not used) int permissions;
        int inode_parent; // pointer to parent inode
        int type;
        //fs_driver_t *driver; // not needed
        int next_inode; /* index of next free inode */
        int protect; /* protection field */
        int nlink; /* number of links to this file */
        int size; /* number of bytes in file */
        int uid; /* owner's user ID */
        int gid; /* owner's group ID */
        // int ctime; /* last status change time */
        // int mtime; /* last data modification time */
        // int atime; /* last access time */
        int dblocks[N_DBLOCKS]; /* pointers to data blocks */
        int iblocks[N_IBLOCKS]; /* pointers to indirect blocks */
        int i2block; /* pointer to doubly indirect block */
        int i3block; /* pointer to triply indirect block */
        int last_block_index; // the block index of the last data block used for this file, useful in
write when exceeding current blocks
}inode;

// returning value of read_dir
struct dirent {
        int type;
        int filename_len;
        char filename[];
        int inode_index;
}dirent;

// structure for f_stat
struct fstat {
        int uid;
```

```
        int gid;
        int filesize;
        int ctime; /* last status change time */
        int mtime; /* last data modification time */
        int atime; /* last access time */
        int type; // dir or regular file
        int permission;
        int inode_index;
}fstat;

// file table entry struct
struct file_table_entry {
        int inode_index;
        int type; // directory or regular file
        /******** start of file indicator fields ********/
        int block_index; // index of the block relative to this file, for instance this is the 5th data
                         //  block for this file, then 'block_index' of this block is 5
        int block_offset; // offset index of block in the data region
        int offset; //offset within a block
        /********* end of file indicator field *************/
        // if type == dir, this fields indicate the number of open files under this directory
        int open_num;
}file_table_entry;

/*  To access file information corresponding to certain file
        use the file descriptor as the index of the 'entries' array */
struct file_table {
        // number of open files
        int filenum;
        int free_fd_num; //number of unused file descriptor
        int free_fd[30]; // array storing the unused file descriptor
        // array of file table entries, number of entries equal to (filenum - 1)
        file_table_entry entries[];
}

//user type stored in the array of users available in the shell
struct user {
        char *name;
        char *password;
        int permission_value;
        int uid;
        int home; //home directory inode pointer value
} user;
```

********** File Functions **********

1. **f_open**(char* filepath, int access, int permission)c
   // access is destined as read or write, that is the action the user want to perform on the file
   a. Things to check:
      i. Check valid file name (length)
      ii. Invalid input flag (not read, write, read/write, append) - return error
   b. How to check if the file exists (could use helper function)
      i. Parse the file path
         - If filepath format does not include '/' or starts with './', look inside of current directory denoted by 'cwd', and if creating file, parent should be the current directory
         - Otherwise follow the path to check validity
      ii. Find and open the directory along the way (needs to put the inode and information associates to each directory along the way into the file table)
   c. If file does not exists
      i. File name cannot exists '/', '.', '..'
      ii. Requires permission as input
      iii. Trying to read - return an error
      iv. Trying to write/append - create the file
      v. Fetch and allocate a free(unused) inode for this file, and update the corresponding information: parent, etc.
      vi. Add the inode index associates to this file to the parent directory, update the parent directory file size
         - How to get which is the parent directory:
            a. Through parsing file path
            b. Or is simply the directory stored in 'cwd'
      vii. Store the file name in inode
      viii. Set the permission of this created file to 0666
         - (based on linux man page)Any created file will have the mode S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH (0666), as modified by the process's umask value
      ix. Add a new or find an existing entry in the file table, and assign this file a file descriptor (which is the index entry in the file table)
      x. Put the corresponding information about this file (inode, location, offset etc.) into the place indexed by the file descriptor in the file table
      xi. Returns the file descriptor
   d. If file exists
      i. Check if this file is already opened by looking through the file table

        ii. Directory files in file table should have an extra field keeps tracking of the number of open files under this directory for purpose of f_close
        iii. If any of the directory does not exists along the way, then file does not exists, otherwise the file exists
        iv. To open an existing file:
            - Get the associating inode (search through the inodes with the file name)
            - Add a new or find an existing entry in the file table for this file
            - Return the file descriptor(file table entry)
    e. Increment the number of opening files in the file table corresponding to the parent directory of this file
    f. If created a file, then add the inode index of the new file into its parent directory data block, increment the number of files inside of its parent directory

2. **f_read** (void* buffer, int size, int ntumes, int file_descriptor)
    a. Check validity of file descriptor
    b. Get the corresponding entry ‘fe’ from the file table indexed with file_descriptor
    c. Uses ‘fe.inode_index’ to get the inode for this file
    d. Uses ‘fe.block_offset’ and ‘fe.offset’ to get to the current file indicator place
    e. Read ‘size * ntimes’ bytes starting from the file indicator position from the disk image with system call ‘read’ and write it to ‘buffer’ with system call ‘write’
        i. When finish reading one block and goes on to the next block, can use the current value of the ‘block_index’ in file table field to calculate, whether we are in direct blocks or indirect blocks, and locate the next block to be read; remember ‘block_index’ needs to be updated as we read
    f. Update the file indicator (‘fe.block_offset’, ‘fe.offset’, ‘fe.block_index’) in file table entry while reading

3. **f _write** (void* buffer, int size, int file_descriptor)
    a. Check validity of file descriptor
    b. Get the corresponding entry ‘fe’ from the file table indexed with file_descriptor
    c. Uses ‘fe.inode_index’ to get the inode for this file
    d. Uses ‘fe.block_offset’ and ‘fe.offset’ to get to the current file indicator place
    e. Read from ‘buffer’ with system call ‘read’, and then write ‘size * ntimes’ bytes starting from the file indicator position into the disk image with system call ‘write’
        i. While writing, keeps track of whether ‘f_write’ has exceeds the current file size:
            - if exceeds the current file size, allocate new blocks for writing using the ‘free_block’ field in the superblock, update the corresponding fields in ‘inode’: ‘dblocks’, ‘iblocks’ etc., and ‘last_block_index’, and also update the ‘size’ field
            - If not, then normal write

        ii. When finish writing one block and goes on to the next block, can use the current value of the 'block_index' in file table field to calculate, whether we are in direct blocks or indirect blocks, and locate the next block to be written; remember 'block_index' needs to be updated as we write
    f. Update the file indicator ('fe.block_offset', 'fe.offset', and 'fe.block_index') in file table entry while writing

4. **f_close** (int file_descriptor)
    a. Check validity of file descriptor
    b. Remove and clears up (bzero) the corresponding entry in the file table
    c. Add the 'file_descriptor' into the 'free_fd' array, and increment 'free_fd_num'
    d. Update the value associate to 'file_descriptor' to NULL
    e. Decrement the number of opening files ('open_num') for the parent directory of this file
        i. If no more file is opened for the parent directory, then close the directory and updates the number of open files for the parent directory for this directory again (each file opened is considered a different open file)

5. **f_seek** (int file_descriptor, int offset, int whence)
    a. Check validity of file descriptor
    b. Check if the 'offset', that is, the position to seek is out of the file, if exceeds current file size, return fail.
    c. Uses the 'file_descriptor' as index to access the specific 'file_table_entry', 'fe', for this file.
    d. Update the file indicator fields for this entry ('fe.block_offset', 'fe.offset', and 'fe.block_index') based on 'offset' and 'whence'.
        i. Calculation required, and multiple cases should be considered

6. **f_rewind** (int file_descriptor)
    a. Check validity of file descriptor
    b. Uses the 'file_descriptor' as index to access the specific 'file_table_entry', 'fe', for this file
    c. Update the file indicator fields for this entry ('fe.block_offset', 'fe.offset', and 'fe.block_index') to the first block offset of this file (can be obtained through this file's inode), '0' as 'fe.offset', and 'fe.block_index' is 0

7. **f_stat** (char* filepath, struct stat* st) //what to return??? Current implementation returns 'stat' structure
    a. Find the inode corresponding to 'filepath'
    b. Update the fields in 'st' with the information in the inode found in step 'a'

8. **f_remove** (char* filepath)
    a. Find the corresponding inode for this filepath

b. Finds the parent of this file (through parsing ‘filepath’ or is simply ‘cwd’)
    i. Remove the inode index of this file from the list of inode indices from the parent directory data
    ii. Move the last inode index in the parent directory(can be obtained in inode with some calculation) data to the current freed slot, decrement the number of files in the parent directory.
c. Add the data blocks used by this file to the free block list:
    i. Starting from the first data block for this file
    ii. Append this data block as the head of the free block list (for better performance)
    iii. Update the free_block head field
    iv. Goes on to the next data block of this file
d. Mark the inode for this file as free_inode, again with the same scheme, append this inode to the head of the free_inode as the new head of this free inode list.
e. Decrement the number of opening files (‘open_num’) for the parent directory of this file if the removed file is opened and in file table
    i. If no more file is opened for the parent directory, then close the directory and updates the information for the parent directory for this directory again

9. **f_opendir**(char* dir)
    a. Check if the directory exists (same as checking file path)
    b. If exists:
        i. Check in the file table whether this directory is already opened, if yes then return the corresponding file descriptor
        ii. If not in the file table, then find the inode for this directory, create a ‘file_table_entry’ struct for this directory, update the corresponding fields, and put it into the file table, and assign a file descriptor (either found through free_fd or assign a new one)
        iii. Return the file descriptor
    c. If does not exists:
        i. Find a free inode for this directory (updates the free_inode list as well)
        ii. Assign this inode to this directory, and also add the inode index for this directory to its parent directory, update the number of files in the parent directory
        iii. Create an entry for this directory in the file table, assign this directory a previously freed file descriptor (update free_fd and free_fd_num) or a new file descriptor

10. **f_readdir**(int file_descriptor)
    a. Check the validity of ‘file_descriptor’ (whether it is opened and exists in the file table)
    b. ‘f_readdir’ takes in the file descriptor of the targeting directory

c. Find the file table entry for this directory file and the position of the current file indicator inside of this directory file
d. Goes on to in the next directory entry (which is an inode index of a file within this directory), and update the file indicator fields in the file table ('fe.block_offset', 'fe.offset', and 'fe.block_index').
e. Get the info for this file entry with the inode index obtained in step 'c'
f. Create a new dirent struct, and update the fields of dirent based on the information in the inode obtained in step 'd'
g. Return the newly created dirent struct
h. If there is no more entries after the current file indicator, or current directory does not contain any files, return NULL

11. **f_closedir**(int file_descriptor)
    a. Check validity of 'file_descriptor'
    b. Add the 'file_descriptor' to the list of 'free_fd' and increment the value of 'free_fd_num'
    c. Clears the 'file_table_entry' associates to this file descriptor (set it to NULL)

12. **f_mkdir**(char* filepath, int mode)
    a. Check the validity of the filepath
        i. Parse the filepath
        ii. Check whether the directory along the way exists
            - If not return FAIL
        iii. Check whether is filepath indeed indicates a directory
    b. Allocate a free inode for this new directory
        i. Use the head of the free_inode list as the inode for this file
        ii. Update the free_inode list head to the next inode
    c. Update the filename (should be the relative directory name), type, and parent in the inode struct for this file
    d. Add the inode index of this file to its parent directory, and increment the number of files inside of its parent directory
        i. How to find parents directory:
            - Parse 'filepath'
            - Or is simpy 'cwd' based on the parsing results
    e. Return SUCCESS

13. **f_rmdir**(char* filepath) // remove the directory only when its empty
    a. Check the validity of 'filepath'
    b. Find the inode corresponding to this directory file
    c. Check the number of files inside of this directory(the first int in the file data of this directory)
        i. If no more files in this directory

- Add the data blocks used by this directory file to the list of free_blocks
- Add the inode of this directory to the free_inode list (add as the new head of free_inode list)
- Remove this file's inode index from the data file of its *parent* directory, and decrement the number of files stored by its parent directory
    a. How to find parents directory:
        i. Parse 'filepath'
        ii. Or is 'cwd' based on the parsing results

ii. If there are still files in this directory
- Iterate through the files inside of this directory, and calls 'f_remove' on the files inside of this directory, and recursively calls 'f_rmdir' on the directories within this directory.

14. **f_mount**(char* diskimg, char* mounting_point)
    a. Check whether 'diskimg' exists should be shell's job
    b. Create 'mounted_disk' struct for this disk image, which is stored in the shell program
    c. Read in the super block and inode region of this disk image, and update the corresponding fields in 'mounted_disk' for this disk
    d. Return a pointer to the 'mounted_disk' struct created

15. **f_umount**(char* mounted_diskname)
    a. Remove the 'mounted_disk' struct corresponds to the 'mounted_diskname' (needs to search),
        i. If found, free the corresponding memory
        ii. If not found, return FAIL
    b. If there is no mounted disk corresponding to this mounting point, return fail

********** Shell **********

Resources:

- Chmod
    - Symbolic mode https://docs.oracle.com/cd/E19683-01/806-4078/6jd6cjs34/index.html
    - Absolute mode https://docs.oracle.com/cd/E19683-01/806-4078/6jd6cjs32/index.html
- ls
    - ls -F (man page info)
        - -F Display a slash (`/') immediately after each pathname that is a directory, an asterisk (`*') after each that is executable, an at sign (`@') after each symbolic link, an equals sign (`=') after each socket, a percent

sign (`%') after each whiteout, and a vertical bar (`|') after each that is a FIFO.
- -l: use a long listing format

Commands:

1. more <filepath>
    a. Check the validity of file path
        i. File must be regular files only, cannot be directories
        ii. File path does exists in the current disk (current mounting point directory)
    b. Keyboard input ‘q’ will quit ‘more’ and space for goes down to the next page
        i. How to implement this?
    c. Uses ‘Ioctl’ to obtain current window size (below is a resource link):
        i. https://stackoverflow.com/questions/1022957/getting-terminal-width-in-c
    d. Uses f_read to read in specified bytes corresponding to the window size from the file, and print out to console
    e. Do not plan to support resizeable windows(window size for display should be constant within one more)
2. ls(needs to re-write)
    a. Get the current working directory (through ‘cwd’ global)
    b. Find the inode associates to ‘cwd’, and open ‘cwd’ by calling ‘f_opendir(‘cwd’)’, obtain the corresponding file descriptor ‘fd’
    c. Uses ‘f_readder(fd)’ to access each entry in the current directory
        i. If ‘-F’ flag is specified, then add the specific postfix to the filename for display, based on the information obtained through ‘dirent’. If more information is needed, uses the ‘inode_index’ field in ‘dirent’ to obtain the info needed from inode
        ii. If ‘-l’ is specified, obtain the inode of each file and print out the required info
        iii. If no flag is specified, simply print out the file names
3. chmod <mode> <filepath>
    a. Symbolic - character (tie character input string to permissions)
    b. Absolute - integers (tie integer value to character input)]
    c. Parse ‘mode’ to get the new permissions
    d. Find the inode associates to ‘filepath’, and update the permission of this file to the new permission obtained in step ‘c’
4. mkdir <directory name> <mode>
    a. Use the f_mkdir() function: call ‘f_mkdir(‘name’, ‘mode’)’
    b. If ‘mode’ is not specified, then use bitwise-inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO
        i. According to OpenGroud: “The value of the bitwise-inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO is used as the *mode* argument.”, http://pubs.opengroup.org/onlinepubs/009696799/utilities/mkdir.html

5. cd <path>
    a. Parse the file path and validify the filepath:
        i. The filepath must exists and is within the disk image
        ii. Directory along the way strictly follows ‘parent and child’ relation: cannot access children of other parent directory
        iii. The final destination of the file path must be a directory
    b. Change ‘cwd’ in ‘cur_user’ to filepath is validity passed
    c. ‘.’ changes ‘cwd’ in ‘cur_user’ to current directory, or just do nothing
    d. ‘..’, then changes ‘cwd’ in ‘cur_user’ to the parent directory
    e. If filepath with ‘~’, then starts with the current user’s home directory, which can be found in the ‘user’ struct.
    f. If filepath starts with ‘/’, goes to the root directory
    g. If no argument is given for cd, then simply goes to the user’s ‘root_dir’ stored in current user global
6. pwd
    a. Print out ‘cwd’
7. cat <files> or cat <redirection> <file> or cat <files> <redirection> <file>
    a. Cat allows multiple files as input
    b. Parse the input for cat
        i. Look for redirection symbols
            1. If needs redirection
                a. Get the output filepath for the output file
                b. Call ‘f_open(output_fp)’ to open the output file
                c. Get the input files,
                    i. First check the validity of those input files: must exists and must be regular files rather than dictionary files
                    ii. uses ‘f_open’, ‘f_read’, ‘f_write’, ‘f_close’ to open, read, write, and then close the input files into the output file in sequence
                d. If the input source is console, the get the console input and uses ‘f_write’ to write the input to the output file
            2. If does not involve redirection
                a. If no input files are related, then get the console input and output to console again
                b. If there are input files:
                    i. Check the validity of input files: exists and must be of regular file type
                    ii. Uses ‘f_open’, ‘f_read’, ‘f_close’ to open, read the file data in sequence, output to console, and then close the files
    c. Limitation:
        i. Only supports one output file

        ii. Right not does not support any flags
8. rm <path>
    a. Call f_remove('path')
9. rmdir <path>
    a. Call f_rmdir('path')
10. mount <disk_img> <mounting_point>
    a. Check whether 'disk_img' exists
        i. If not asks user to run 'format'
        ii. If existsqw, call 'f_mount(disk_img, mounting_point)'
    b. After mount, ask user to log in, then update the 'cur_user' global, and also the logged in user's info
        i. Update the 'root_dir' and 'cwd' to 'mounting_point'
        ii. Update 'current_disk' to the 'disk_img'

11. unmount <target>
    a. Call 'f_unmount (char* target)'
    b. Log out user.
    c. Not sure how to handle which working directory shell should direct to
12. format
    a. Created a disk image
    b. The disk image needs to at least contains the root directory (maybe root directory could always use the first inode)
    c. First write boot block and super block info:
        i. block size = 512 bytes
        ii. 20 inodes, that means data offset of inode is 7
        iii. Disk image size be 1G

********** Helper Functions **********

1. check_login() //checks the login is valid for the shell
2. get_inode(int inode_index) //find the corresponding inode address of the specified inode index
3. make_inode(args)//create inode with the specified informations
4. find_inode(char *filename) //filename can be the directory name or the file name
5. make_dirent(args) //create a dirent stuct with the specified arguments
6. check_valid_descriptor(int fileDescriptor, int type) - write, read, seek, close
    a. Check whether this file descriptor is indeed in the file table, with the specified type
7. parse_file_path(char* path) // parsing for shell
    a. Split into tokens with delimiter '/';
    b. If contains '~', then simply this with '/home/<username>/'
    c. If starts with './' or not starting with '/', append the current working directory of the user to the 'path'

    d. If starts with '../', then append the parent directory(extracted from current working directory) to the 'path':
        i. For instance if 'path' = ../haha,txt, and the current working directory for current user is '/home/hehe/C', then the path returned will be '/home/hehe/haha.txt'
    e. Then output the path after parsing
8. filepath_parser(char* abspath, int type):
    a. Always starts from the root directory, check along the way
    b. Check for whether directory along the way exists or not
    c. Return 'True' if the 'abspath' exists, return 'False' when not exists or is not of the specified 'type'

**********Questions**********

1. When user typed in '/' and '~' in file path, do these symbols refers to the same directory, that is, the root directory of the mounted disk?