Assignment 4 - File Systems

Group 15
Jinyan Li(jli134@ucsc.edu)
Jacob Berrier (jberrier@ucsc.edu) (captain)
Seth Little (sealittl@ucsc.edu)
Amy Huang (awhuang@ucsc.edu)

Goal:

The goal for this assignment is to implement a FAT-based file system in user-space by using the FUSE system framework.

Introduction:

A file system, is a part of the operating system which deals with how files are stored and managed on a disk. There are many different ways in which a file system could be structured and implemented -- ie: using linked lists, bitmaps, etc. Depending on the structure chosen, there may be different tradeoffs such as access time and efficient disk utilization.

In this assignment, a FAT (File Allocation Table) structure was implemented to manage files on a disk. This structure is similar to a linked-list file system structure, except it uses a file allocation table to keep track of contiguous disk blocks. The advantages of using a FAT structure instead of a traditional linked list structure is that it allows for fast random memory access, and reduces fragmentation by keeping the pointers of each disk block in a table in memory. However, the disadvantage of this structure is that it requires the table to be stored in memory. Larger disks would require larger tables. As a result, this kind of structure would not be practical for large disks. However, this file system structure is simple to implement nonetheless, which is why we would be implementing it for this assignment.

Methods:

To implement the FAT file system, we would need a disk and a framework to implement our file system routines. A user space program would be needed to generate a disk with a set number of blocks, and the FUSE framework would be used to implement our filesystem in userspace.

- Disk structure:
  The disk is broken down into N number of 1kb blocks. The first block (Block 0) is reserved as the Super Block, and the next k number of blocks (Blocks 1 to k) are reserved for the FAT blocks. The value k is determined using N/256, where N represents the total number of disk blocks and 256 represents the total number of 4 byte block pointers a single FAT block can hold. The

remaining blocks are the data blocks, which are used to store files and directories. Since Super Blocks and FAT blocks are reserved blocks, they are marked as unused. An image of the disk layout could be seen in Figure 1 below.

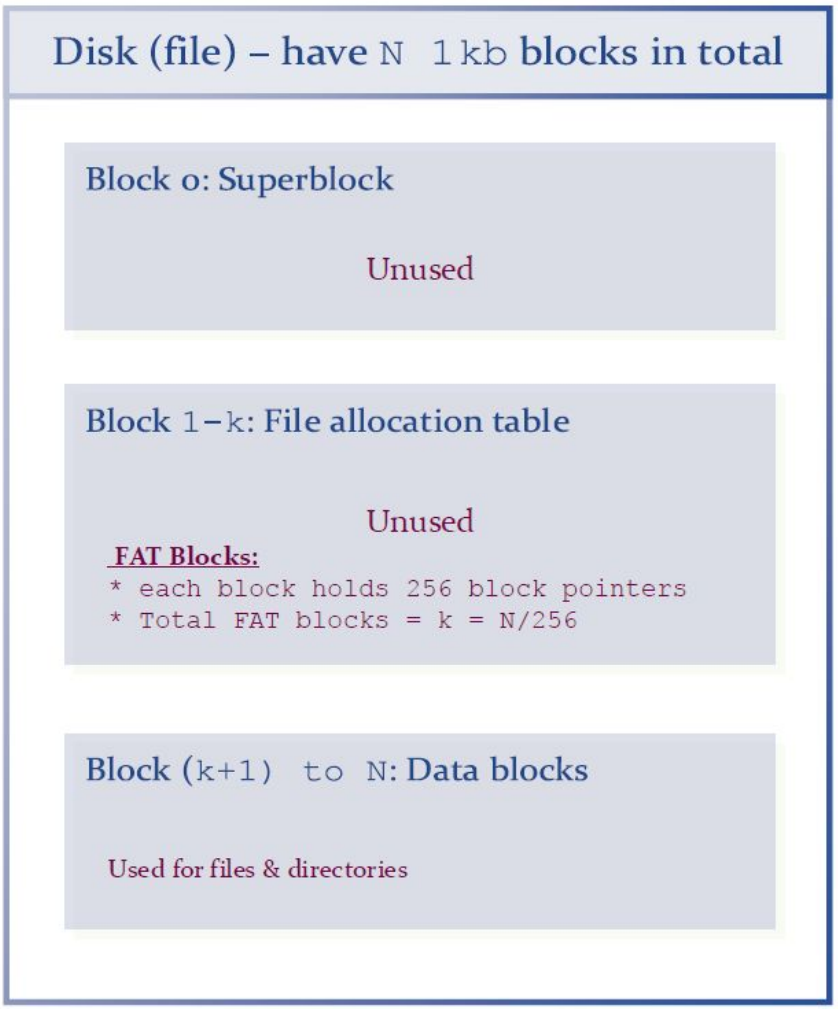

Figure 1: Disk structure

- Block structures:
  - **Super Block**
    The Super Block consists of five 4 byte words which stores the metadata of the disk. The first word stores a magic number which in this assignment is 0x2345beef. This magic number is a unique identifier used to identify the file-system type. The second word indicates the total number of disk blocks on the disk. The third word indicates the total number of FAT blocks, or the value k. The forth word indicates the size of each block. Since we are using 1kb blocks in this assignment, it is set to 1024. Finally the fifth word is used to indicate the starting block

of the root directory. Figure 2 shows the structure of the super block.

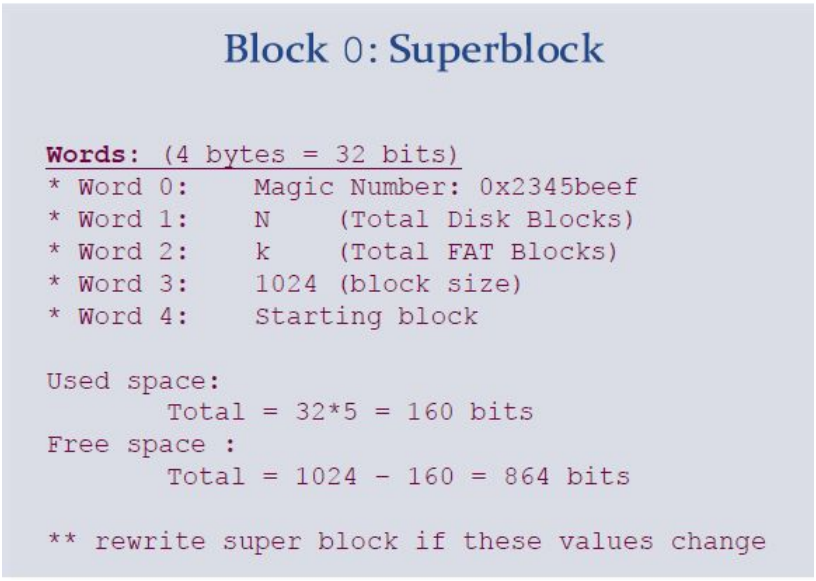

Figure 2: Super Block structure

- **FAT Block**

  FAT blocks are the next consecutive block(s) following the super block. Each FAT block can store up to 256 4 byte block pointers (because 4 bytes * 256 = 1kb). The index of FAT each entry represent a block in the disk, and the 4 byte value stored in each entry represent either the block's current status or it's next block entry. A value of 0 as an entry represents a free block. A -1 as an entry means that FAT entries should never be allocated for those blocks, and an entry of -2 means that that block is the last block for that file. Any other number inside a block would represent the block's next block entry. Figure 3 shows the structure of the FAT blocks.

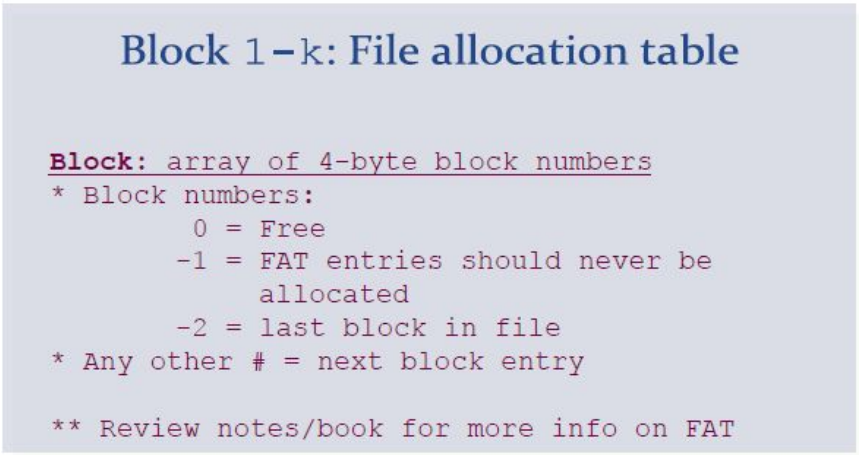

Figure 3: FAT Block structure

- **Data Blocks**

  Data blocks contain both information about files and directory, as well as data for each file. The metadata for files and directories follow the structure of 64 bytes. The first 6 words (or 24 bytes; 1 word=32 bits or 4 bytes) store a null-terminated string as the file or directory's name of up to 24 bytes long. The next two words (8 bytes) are stores the file/directory's creation time. The following two words sotre the modification time, and the next two words store the file access time. The next word stores the length of the file/directory. This value represents the number of entries in that file/directory. The next word following that stores the start block number. This number points to the block that stores its first file/directory entry. If the value is 0, it means that there are no entries, or that the entry is free. The next word is a flag used to indicate whether the entry made is a file or a directory. A value of 0 for the flag represents a file entry and a value of 1 represents a directory entry. The last word (word #15) is left as unused, or may be used for other purposes. Figure 4 shows the metadata structure of a data block.

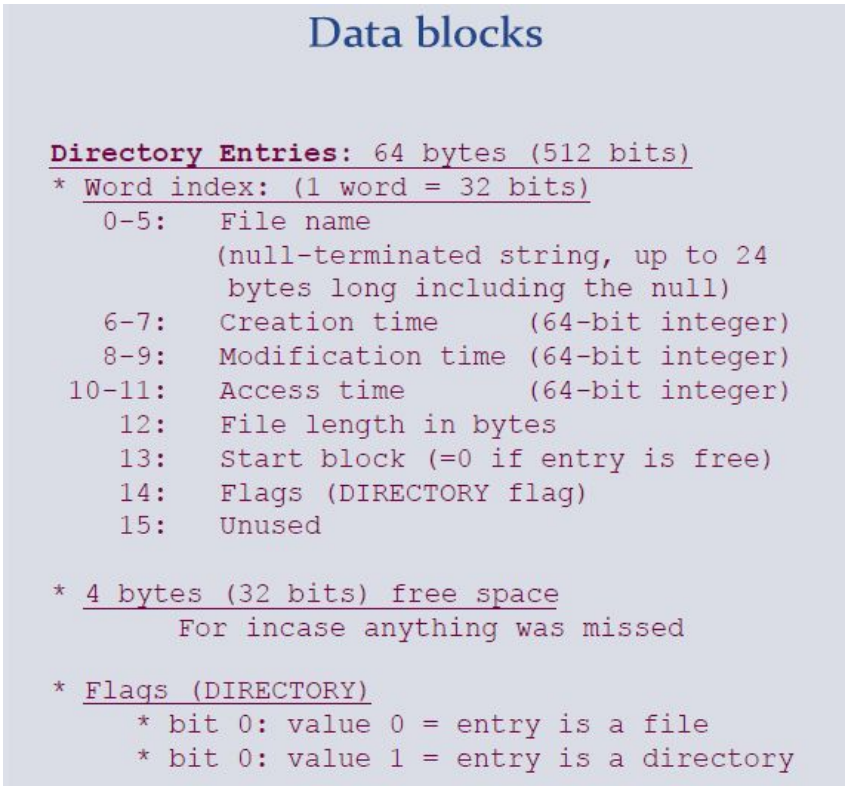

Figure 4: Data Block structure

Assumptions:

In this assignment, we would assume that each file and directory entry would be stored in one disk block. Traversing through a path would look at the starting block of each file/directory entry and the FAT table would be used to locate the other files and subdirectories within a given directory.

We also assume the FUSE write routine does not work. We tried testing the FUSE example with append, file redirection, and vim, but none of those commands seem to call the write subroutine.

Implementation:

- Disk structure

  To implement the disk structure, we would write a user-space program which will take two parameters: "Disk Image Name" and "Number of disk blocks". Based on the two parameters, the program would create a new file with a size equivalent to (Number of Disk blocks)*(1kb block sizes)

  - Pseudocode

```
// check for valid parameters
if(argc !=3)
  // bad input
  print(error, display usage)
  return EXIT_FAILURE
else
  // argv[2] = number of blocks
  // check to see if it's a number
  get length of 2nd argument (argv[2])
  For each character in argv[2]
        if( not a digit)
           print(error, display usage)
           Return EXIT_FAILURE

// After above, should mean valid arguments were given
// Calculate and store information about disk
diskImageName = argv[1]
blockNum      = atoi(argv[2]) // total disk blocks
k             = ceiling of (blockNum/256)
// calculate ceiling incase block num is not evenly
// divided by 256

// Create and initialize disk
// store file descriptor to new file using fd
fd = open(create new file with name = diskImageName)
```

lseek(go to the start of the file (fd) (block 0)) //1024*0
for every byte in (blocksize * blockNum) // blocksize=1024
  Write a 0 to each byte of data

// Initialize Super Block
// Super Block is our root directory
lseek(go to the start of the file (fd) (block 0)) //1024*0
// Write five words (words = 4 bytes) using write(to fd);
Word 0: 0x2345beef
Word 1: blockNum
Word 2: k
Word 3: blocksize = 1024
Word 4: k+1 // start of data blocks

// Initialize FAT Blocks
lseek(fd to Block 1 (1*1024))
for(entries 0 to k)
  write(-1 to indicate that they're unused)

// wordsize = 4
// if there are more FAT entries available than there are
// disk blocks, set extra entries to -1 (unused)
// only occurs if N is not a multiple of 256
seek(fd to Block N (1*1024 + (N*wordsize)))
for(each FAT entry following entry N-1)
  Write (fd -1 to indicate that it's unused)

// ** TESTING:
// Create sample file & directory entry in root directory

// Create and initialize a directory in root
- setup metadata for dummy folder called 'This_is_a_directory'
- create .. entry for root directory

// Create and initialize a file in root directory
- Setup metadata for dummy file called 'hello'
- Write the data for hello in a separate block
- Fill in the data with 'Hello World'

// Setup FAT to link disk blocks
// -- Data Blocks --

```
// k+1 (5):  this_is_a_directory (directory header)
// k+2 (6):  ..
// k+3 (7):  hello (file header)
// k+4 (8):  hello (file data)

// -- FAT Block Entries --
// In first block (entries 0-255):
//   0:   -1 (Super Block)
// 1-4:   -1 (FAT Blocks)

//   5:    7 (k+1) this_is_a_directory
//   6:   -2 (k+2) ..
//   7:   -2 (k+3) hello
//   8:   -2 (k+4) hello (file data)
```

- Fuse Routines
  To implement the file system, FUSE would be used as our file system's framework. In reviewing the example fuse programs provided by FUSE, we plan to modify the following routines to have our file system support basic operations such as cat, mkdir, and ls.
  - Pseudocode
  - **main()**
    - Open disk image generated by diskinit
    - Go to the super block 0, and read out its metadata using read()
    - Words read:
      - Word 0 store as magic number
      - Word 1 store as total_blocks
      - Word 2 store as k
      - Word 3 store as block_size
      - Word 4 store as rootBlockNum
    - Allocate an array large enough to store all entries in the FAT table
      - This would help us find a free block quicker than using lseek all the time
    - Store all these values as global variables so that they may be accessed in FUSE subroutines
  - **fat_getattr(*path, *stbuf)**
    - Initialize variables and memory buffer
    - Set current block to root block
    - If the path is root
      - Set stbuf's st_mode to S_IFDIR
      - Set stbuf's st_nlink to 2

- Else
    - Remove the initial slash
    - While-loop (inifinite until broken)
        - While-loop (inifinite until broken)
            - Lseek to current data block
            - Read the filename
            - Set found to 0
            - If the filename is equivalent to the remaining path name,
                - Set found to 1
                - Read Creation, Modification and Access Time, File Length, Starting Block, and Flags.
                - ***Break***
            - Set nextBlock to fat[thisblock]
            - If the you are at the end of the file block
                - ***Break***
            - Current Block is set to Next Block
        - If not found
            - Return Result = -ENOENT
            - ***Break***
        - Set remaining pathname to strtok(NULL, "/")
        - If the remaining pathname is NULL, then ***break***
        - Set current block to the starting block
    - If the flag is set to directory
        - Set stbuf->st_mode to S_IFDIR | 0755
        - Set stbuf->st_nlink to 1
        - Set stbuf->st_size to fileLength
    - Else-If the flag is set to file
        - Set stbuf->st_mode to S_IFREG | 0444
        - Set stbuf->st_nlink to 1
        - Set stbuf->st_size to fileLength
    - Else
        - Set Return Result to -ENOENT
    - Set stbuf->st_atim.tv_sec to accessTime;
    - Set stbuf->st_mtim.tv_sec to modificationTime;
    - Set stbuf->st_birthtim.tv_sec to creationTime;
- Return res

- **fat_readattr(*path, *buf, fuse_fill_dr_t, off_t, fuse_file_info)**
    - Thisblock = rootblock
    - If (path is root) {
        - while(1)

        - lseek(fd, thisblock*1024, SEEK_SET)
        - read(fd, &filename, 24)
        - If filename = tok {
        - Seek to starting block
        - Read from the starting block
    - Nextblock = FAT[thisblock]
    - If nextblock == -2 break
    - Thisblock = nextblock
- else
    - while(1) {
        - while(1)
            - Seek 1024 bytes forward
            - Read filename
            - If filename = tok, found = 1, break
            - Nextblock = fat[thisblock]
            - If nextblock == -2 break
            - Thisblock = nextblock
        - If not found, return -ENOENT
        - Tok = strtok(NULL, "/")
        - If tok IS NULL break
        - Thisblock = startingblock

    - Thisblock = starting block
    - while(1)
        - Lseek by 1024 forward
        - Read the filename
        - If (tok isnt NULL)
            - If filename is tok
                - Lseek to thisblock*1024+4*13
                - Read 4 bytes from startingBlock

        - nextBlock = fat[thisBlock]
        - If nextBlock == -2 break
        - Thisblock = nextblock
- Return success

- **fat_read(*path, *buf, fuse_fill_dr_t, off_t, fuse_file_info)**
    - If path is root return -ENOENT
    - Else
        - Tok = strtok(path, "/")
        - while(1)
            - while(1)
                - Lseek (thisblock*1024)
                - Read filename

            - If filename is tok
                - Found = 1
                - Read all of the fields
                - Break
            - Nextblock = fat[thisblock]
            - If nextblock == -2 break
            - Thisblock = nextblock
        - If not found
            - Return -ENOENT
        - Tok = strtok(NULL, "/")
        - If (tok == NULL) break
        - thisBlock = startingblock
    - Create buffer for reading
    - Read data from file blocks
    - Put data into buffer
    - Return size of data read
- **<u>fat_open(*path, *fi)</u>**
    - If path is root, return -ENOENT
    - Tok = strtok(copy, "/")
    - while(1)
        - while(1)
            - Lseek thisblock*1024
            - Read filename
            - If filename is tok
                - found=1
                - Read attributes
                - Break
            - Nextblock = fat[thisblock]
            - If nextblock == -2 break
            - Thisblock = nextblock
        - If not found break
        - Tok = strtok(NULL, "/")
        - If tok is NULL break
        - Thisblock = startingBlock
    - If found
        - fi->fh = startingBlock
    - Else
        - Get an empty block for files directory entry by incrementing through FAT until a 0 is found
        - Set end of last file in FAT to -2
        - Get an empty block for the starting block
        - Set end of new file in FAT to -2

- Write filename, creation time, modification time, access time, file length in bytes, start block, directory flag
- Write FAT back to disk

- Return success

- **fat_mkdir(*path, mode)**
  - If path is root return -errno
  - Tok = strtok(copy, "/")
  - while(1)
    - Parentblock = thisblock
    - while(1)
    - Seek, read filename
    - If filename is tok
      - Found = 1
      - Read attributes
      - Break
    - Nextblock = fat[thisblock]
    - If nextblock is -2 break
    - Thisblock = nextblock
  - If not found
    - Get an empty block for the new directory
    - Get an empty block for its .. entry
    - write new directory attributes (filename, creation time, modification time, access time, file length, start block, flag)
    - write parent attributes (same as above)
    - Write FAT back to disk
    - Break
  - Tok = strtok(NULL, "/")
  - If tok is NULL break
  - Thisblock = startingBlock

fat_rmdir(path)

- thisblock=rootblock
- If (the path is root){
- return error
- }Else{
- //get the starting block of the target directory from its directory entry
- // parse path with delimiter "/"
- token = strtok(path , "/")
- while(1) {
- while(1){
- lseek(fd, thisblock*1024, SEEK_SET);
- filename=get_file_name()

```
            if (filename==token){
                this_file_is_found=true
                startingblock=Find the starting block of this file()
                break
            }
            previousblock=thisblock
            nextBlock=fat[thisblock]
            if(nextBlock==-2) Break
            thisblock=nextBlock;
        }
        if(file is not fountd){
             return error
        }
        token = strtok(NULL, "/")

        if(token is empty) break
        thisblock=startingBlock
      }
    }
        //check if the target directory is empty
        while(1){
          //Go to block position of the target directory's directory entry
          lseek(fd, thisblock*1024, SEEK_SET)
          filename=Read_file_name()
          if(filename!="..")
            Return "this directory is not empty"
        }

        if(this directory is empty){
            //delete .. entry
            fat[startingBlock]=0
            thisblock=currentblock

            if(this directory is the last block of its directory file){
                fat[previousblock]=-2
                fat[thisblock]=0
            }else{
                nextBlock=fat[thisblock]
                fat[previousblock]=nextBlock
                fat[thisblock]=0
            }
        }
```

-
-

fat_mkdir(const char *path, mode_t mode)

This FUSE routine would be used to handle making new directory entries in the file system.

- Look at the given path and check to see if it's / or just the root directory
  - If so print(error)
  - Exit
- Otherwise, parse the path by '/' and look at each word (token) in the path
- Set thisblock to to root block
- Loop starting at the beginning of the root block
  - Set the parent block to thisblock
  - Loop
    - Read the metadata to get the file/directory's name
    - Check to see if the filename read from the block matches that token
      - If found read, read and store the entry's metadata
      - Set a found flag to 1 to indicate a match was found
      - And break out of loop
    - Set the nextblock have the value in the fat block indexed at thisblock
    - If the nextBlock have a value of -2, it means the block reached the end, so break out of the inner loop
    - Otherwise, set thisblock=to nextblock and continue the inner loop
  - In the outer loop, if the token was not found, create the directory
    - Get an empty block for its directory by traversing through the FAT table until a value of 0 is found
    - Set the FAT table at entry (this block) to the new block number
    - Set the entry of the FAT table at new block number to be the new end block (-2)
    - Find an empty block for this new directory's .. entry.
    - Use a loop again to traverse through the FAT table until a value of 0 is seen.
    - Once a new free block is found, setup the metadata for the new directory and write it to the first new block found.
    - Next, setup the metadata for the parent directory and write it to the second new block found.

        - Write to the FAT table on the disk to reflect the file links made
        - Exit outer loop
    - Otherwise, if the token was found, check to see if there is another token in the path. If there's none, exit out of the outer loop, otherwise, set thisBlock to the starting block and repeat

fat_write(path, *buf, size, offset, fuse_file_info *fi)

- Initialize Variables
- Get the starting block of the file
    - int file_starting_block=fi->fh
    - if (file_starting_block== -1)
        - return -errno
- Calculate the size in blocks
    - float d=(float)size
    - num_of_blocks=(int)ceil(d/1024.0)
- if(num_of_blocks==1)
    - lseek(fd, file_starting_block*1024, SEEK_SET);
    - res = pwrite(fd, buf, size, offset);
    - if (res == -1)
        - res = -errno;
- Else,
    - return -errno;
- return res;

fat_rename(*path, *newname)

- Thisblock = rootblocknum
- If path is root return -ENOENT
- Tok = strtok(copy, "/")
- while(1)
    - while(1)
        - Lseek by 1024 foward
        - Read filename
        - Found = 0
        - If filename is tok
            - Found = 1
            - Lseek to starting block, read startingblock
            - Break
        - Nextblock = fat[thisblock]
        - If nextblock is -2 break
        - Thisblock = nextblock
    - If not found
        - Return -ENOENT

        - Tok = strtok(NULL, "/")
        - If tok is NULL
            - Break
        - Thisblock = startingblock
- Copy newname to a 24 char "namebuffer"
- Lseek to name
- Write namebuffer
- Return success

fat_unlink(*path)

- Thisblock = rootblocknum
- If path is root
    - Return -ENOENT
- Tok = strtok(copy, "/")
- while(1)
    - while(1)
        - Lseek 1024 forward
        - Read filename
        - Found = 0
        - If filename is tok
            - Found = 1
            - Lseek to startingblock
            - Read startingblock
            - Break
        - PreviousBlock = thisblock
        - Nextblock = fat[thisblock]
        - If nextblock is -2
            - Break
        - Thisblock = nextblock
    - If NOT found
        - Return -ENOENT
    - Tok = strtok(NULL, "/")
    - If tok is NULL
        - Break
    - Thisblock = startingBlock
- //Delete the target file directory and its data
- Fat[startingblock] = 0
- Nextblock = fat[thisblock]
- If nextblock is -2
    - Fat[previousblock] = -2
    - Fat[thisblock] = 0
- Else
    - Nextblock = fat[thisblock]
    - Fat[previousblock] = nextblock

    - Fat[thisblock] = 0
- Write FAT back to disk
- Return success

- Testing:
  In testing our FUSE routine, we would implement a sample file and directory in our disk image, so that we may test basic commands. A shell program (test.sh) would be used to test a list of basic commands. Refer to the README file in the asgn4 directory for information on how to run the script.