Anne Whitman: alh220
Michael Mazzola: mjm706

FUSE FILE SYSTEM – ASSIGNMENT 3

Quick Important Notes:

- Total Data Blocks: 32445 (defined as DATABLOCKS)
- Total inode Blocks: 323 (defined as INODEBLOKCS)
- Blocks Per inode: 100 (defined as BLOCKSPERNODE)
- Maximum Path Size: 50 (defined as PATHSIZE)

} If one of these changes, all of these change.

  - **Important:** In order to increase space inside of the inode to store data, we made the path size 50 (including null terminator), so be particularly careful when creating/nesting long file/directory names that the full path doesn't go over **49**, because the last character is reserved for **'\0'**. Will result in "Numerical result out of range" error

- using 'touch' to create a file is working properly however when it is executed it displays the error "touch: setting times of '<FILENAME>': Function not implemented" which Dan acknowledged is okay.

Metadata about the file:

blocks[INODEBLOCKS + DATABLOCKS] is an array of chars (acting as block indices), partitioned into two regions. The first region references inodes and the second references data blocks.

Extensions/Improvements:

The Extension/Improvement we decided to implement was directory support. We also implemented the Extended Directory Operations for extra credit.

Structs and Enums:

Boolean Enum:
- FALSE (as 0)
- TRUE (as 1)

Filetype Enum:
- FILE_NODE
- DIR_NODE

Inode struct:

The inode struct is exactly 512 bytes (the size of one block) and contains:

- An integer representing the size of the file it references
- An integer representing:
    - For Files: the number of data blocks a file is currently referencing
    - For Directories: the number of files/directories inside of that directory.
- An integer array of all data blocks the file references (empty for directories)
- An integer representing links to the file
    - We did not implement links for files, but this is incremented for directories everytime a directory is made inside of it (because of the '..' inside the new directory) and decremented everytime a directory is removed from within.
- A filetype value which is used to distinguish if the inode is used for a file or a directory
    - we declared an enum for this that can be found at the top of the file below the include header section.
- A mode_t value which is used to store the read, write, and execute capabilities of a file as well as S_IFDIR or S_IFREG which gives further information for files vs directories.
- A time_t value used for the time that the file was created
- A time_t value used for the time that the file was last modified
- A time_t value used for the time that the file was last accessed
- An integer representing userId of the owner of the file.
- An integer representing the groupId of the owner of the file.
- Char array of 50 characters used to store the path.

Things We tested:

Creating Files / Directories

- Creating 10 nested directories
    - mkdir d1; cd d1
    - mkdir d2; cd d2
    - mkdir d3; cd d3
    - mkdir d4; cd d4
    - mkdir d5; cd d5
    - mkdir d6; cd d6
    - mkdir d7; cd d7
    - mkdir d8; cd d8

    - mkdir d9; cd d9
    - mkdir d10; cd d10

    - Expect no errors and for directories to create (Success)

- Using 'touch' to create 50 files
    - touch f1; touch f2; touch f3; … ; touch f50

    - Expect no errors and for files to create (Success)

- Creating 5 different files in 5 different directories
    - mkdir first; cd first; touch firstf; cd ..
    - mkdir second; cd second; touch secondf; cd..
    - mkdir third; cd third; touch thirdf; cd ..
    - mkdir fourth; cd fourth; touch fourth; cd ..
    - mkdir fifth; cd fifth; touch fifthf; cd ..

    - Expect no errors and for directories / files to create (Success)
    - Also expect that each directory only has it's own files (Success)

- Creating 10 directories and removing them in random ordering
    - mkdir d1; mkdir d2; mkdir d3; mkdir d4; mkdir d5; mkdir d6; mkdir d7; mkdir d8l mkdir d9; mkdir d10
    - ls
    - rmdir d10; rmdir d2; rmdir d4; rmdir d1; rmdir d8; rmdir d5; rmdir d3; rmdir d6; rmdir d9; rmdir d7
    - ls

    - Expect to see all directories between mkdir and rmdir (Success)
    - Expect to see no directories after rmdir (Success)

- Creating and Deleting 10 files
    - touch f1; touch f2; touch f3; touch f4; touch f5; touch f6; touch f7; touch f8; touch f9; touch f10
    - ls
    - rm f9; rm f2; rm f4; rm f1; rm f8; rm f5; rm f3; rm f6; rm f10; rm f7
    - ls

    - Expect to see all files between touch and rm (Success)
    - Expect to see no files after rm (Success)

- Trying to create two files with the same name

    - touch testfile
    - touch testfile

    - Expected this to fail. We return EEXIST (which is the error code for a file that already exists). This did fail (did not create the same file twice), but the terminal did not mention it. But our log file does. (Test successful)

- Trying to create two directories with the same
    - mkdir testdir
    - mkdir testdir

    - Expected this to fail. We return EEXIST. Same scenario as the test case above this one. (Test successful)

- Trying to delete a file/directory that doesn't exist
    - mkdir d1; cd d1;
    - rmdir nodir
    - rm nofile

    - Expecting both to fail. (Test successful)

- Trying to delete a directory that is not empty
    - mkdir d1; cd d1
    - touch file1; cd ..
    - rmdir d1

    - Expect to fail because directory is not empty. (Test successful)

Writing to and Reading from Files

- Writing less than a block's worth of characters, checking size, and reading data
    - touch testfile
    - echo [100 bytes] >> testfile
    - ls -al
    - cat testfile

    - expect to see testfile in 'ls –al' and to see 100 bytes (keeping in mind that the last byte is always '\0', so if you manually type 99 characters, it will pass in 100 with the null terminator.) (Success)
    - expect all 100 bytes to print back to screen (Success)

- Writing more than a blocks worth of characters at once, then reading the data
    - touch testfile

    - echo [2000 bytes] >> testfile
    - ls -al
    - cat testfile

    - expect to see testfile in 'ls –al' and to see 2000 bytes (keeping in mind that the last byte is always '\0') (Success)
    - expect all 200 bytes to print back to screen with no gaps between blocks and no characters missing. (Success)

- Writing (with > command) into a file that has not yet been created
    - echo "test" > testfile
    - cat testfile

    - Expect file to be created and for "test" to print to screen (Success)

- Writing (with >> command) into a file that has not yet been created
    - echo "test" > testfile
    - cat testfile

    - Expect file to be created and for "test" to print to screen (Success)

- Writing (with > command) into a file that does already exist.
    - touch testfile
    - echo "test" > testfile

    - Expected this to work, and it did not. This seemed to be an issue with the commands that the operating system runs (not a simple write), so this wasn't working for anyone.

- Writing (with >> command) into a file that does already exist.
    - touch testfile
    - echo "test" >> testfile

    - Expected to write normally (Success)

- Writing less than a block's worth of data into a file repeatedly until forced into more blocks
    - echo [500 bytes] >> testfile
    - echo [500 bytes] >> testfile
    - echo [500 bytes] >> testfile

    - Expected to properly append at the end of the file without overwriting between or gaps between. (Success)

- Writing 51200 bytes (most a file can support)
    - echo [51200 bytes] >> testfile
    - cat testfile

    - Expect no errors and for entire string to write in and to read out to screen

- Writing 51201 bytes (1 more byte than a file can support)
    - echo [51201 bytes] >> testfile
    - cat testfile

    - Originally expected entire operation to fail, but it looks like the system reads and writes in blocks of 4096. So the all except the last write will succeed, and the last write will fail. The cat will print everything except for the last (51201 % 4096) bytes (because those were never written to file). (Test successful)

- We repeatedly tested many more writes than just these to ensure 100% that our write was working properly.

Checking Timestamps

- **Note:** Create an access can only be truly tested through the log file or print statements

- Modify File:
    - touch file
    - ls –al
    - *(note timestamp and wait 1 full minute)*
    - echo “test” >> file
    - ls –al
    - *(note timestamp now)*

    - Expected timestamp to change to new time because file was modified.

- Modify Directory:
    - mkdir d1
    - ls –alt
    - *(note timestamp and wait 1 full minute)*
    - cd d1
    - mkdir d2
    - cd ..; ls –alt
    - *(note timestamp)*
    - *(repeat for rmdir <directory>, touch <file>, and rm <file>)*

- Expected modify time to update based on each transaction (since directory modify time is based on files/directories being created/removed from within.

Additional Functions

int findFirstFreeInode();

This function is used to locate the first available inode in the inode region of the blocks array. If an available inode is not found the function will return -1, otherwise, the block index is returned.

int findFirstFreeData();

This function is used to locate the first available data block in the data region of the blocks array. If an available data block is not found the function will return -1, otherwise the block index is returned.

int findInode(const char* path);

This function is given a path and searches through the blocks array to find an inode with a matching path. If an inode is found, the block index is returned, otherwise, the function returns -1.

int getParentDir(char* path);

This function is given a path and is used to determine the immediate directory of the path. It does this by starting at the end of the path and moving backwards until it reaches the '/' character. It then uses the memset function to set the path to parent directory, which is the path without the characters following '/'.

int getShortPath(char* path);

This function is given a path and is used to find the name of the current directory/file of the given path. It does this by starting at the end of the path and moving backwards until it reaches '/' characterd. It then uses the memset function to set the path to the file/directory name, which is the characters follwingr the last '/' in the path. If the root directory is given to the function it returns immediately.

Functions Provided by FUSE:

void *sfs_init(struct fuse_conn_info *conn);

This function initializes the blocks array and sets all of the inode and data blocks to ‘0’ which indicates they are not in use. It also sets all of the attributes for the root directory inode and then sets the first block in the block array to 1 (since the root node now

exists). Lastly, it uses the block_write function to write the root inode to the file and return a void pointer to SFS_DATA.

void sfs_destroy(void *userdata);

This function uses disk_close to close the currently opened file.

int sfs_getattr(const char *path, struct stat *statbuf);

This function is given a path and a stat. It uses the findInode function to determine which block that the path references. If findInode returns -1, indicating that the path doesn't exist, the function returns -ENOENT, otherwise, it uses block_read to read the block into a buffer. It then fills in the given stat struct with all information in the inode.

int sfs_create(const char *path, mode_t mode, struct fuse_file_info *fi);

This function is given a path, mode, and a fuse_file_info and creates and opens a file in the current directory if it doesn't already exist. It uses the findFirstFreeInode function to locate the first available inode location and then enters all applicable data into the inode. The function then writes the data to the file using block_write and sets its index in the block array to 1, to indicate that it is in use. Upon successful completion the modify time for the parent directory is updated, the create time for the file is updated, and the function returns EEXIST, however, if there are no available inodes for the function to use, the function returns ENOSPC.

int sfs_unlink(const char *path);

This function is given a path and is used to remove a file from the current directory. It uses the findInode function to locate the inode block that corresponds to the given path. If an inode block is not located the function returns -ENOENT, otherwise, the function sets all blocks the inode references in its inner blockNum array to 0. It also sets those same block values to zero in the blocks array to indicate they are no longer in use. Lastly, the given block index is set to 0 in the block array, the modify time of the parent directory is updated, and the function returns 0.

int sfs_open(const char *path, struct fuse_file_info *fi)

This function is given a path and fuse_file_info and is used to open a file. It uses the findInode function to verify the path exists. If the block is not found the function returns -ENOENT, otherwise, the file is confirmed to exist is ready to be opened, and the function returns 0.

int sfs_release(const char *path, struct fuse_file_info *fi);

This function is given a path and fuse_file_info and is used to close a file. It uses the findInode function to verify the path exists. If the block is not found the function returns -ENOENT, otherwise, the access time of the file is updated and the function returns 0.

int sfs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi);

This function is given a path, buffer, size, offset, and fuse_file_info and is used to read data from a file. First, the findInode function is used to locate the block that corresponds to the given path. If a block is not found the function returns -ENOENT, otherwise, the data from the file is copied into the given buffer using memcpy. Upon success, the number of bytes read is returned.

int sfs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi);

This function is given a path, buffer, size, offset, and fuse_file_info and is used to write data to a file. First, the findInode function is used to locate the block that corresponds to the given path. If a block is not found the function returns -ENOENT, otherwise, the amount of bytes to be written is checked to ensure that space is available. If there is not enough space for the write to complete, ENOSPC is returned, otherwise, the findFreeData function is used to find available data blocks to write the data to. Overwriting bits in file is not allowed and therefore the function either appends or does nothing. Upon successful completion the modify time for the file is updated and the amount written is returned.

int sfs_mkdir(const char *path, mode_t mode);

This function is given a path and a mode and is used to create a directory. First, it uses the findInode function to check if the path already exists in the current directory and, if it does, returns EEXIST. It then uses the findFirstFreeInode function to find an available inode for the directory to use. If an inode cannot be found ENOSPC is returned, otherwise, the inode is filled with the appropriate data for the directory being created. The inode is set to 1 in the blocks array indicating that it is in use. Upon successful completion the modify time for the parent directory is updated, the create time for the directory is updated, and 0 is returned.

int sfs_rmdir(const char *path);

This function is given a path and is used to remove a directory. It uses the findInode function to locate the block index of the given path. If a block index is not found the function returns -ENOENT otherwise the function verifies that the path given is a directory. If it is not a directory the function returns ENOTDIR, otherwise the function then checks if the directory is empty. If it is not empty the function returns ENOTEMPTY, otherwise the block index in the block array is set to 0 indicating that it is no longer in use. It's data is then cleared out to default values and the function returns 0 upon successful completion.

int sfs_opendir(const char *path, struct fuse_file_info *fi);

This function is given a path and fuse_file_info and is used to open a directory. It uses the findInode function to locate the inode block index of the given path. If an index is

not found, the function returns -ENOENT, otherwise, the inode is read into a buffer. The inode is then checked to verify that it is a directory and if it isn't the function returns ENOTDIR. Upon successful completion 0 is returned.

int sfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi);

This function is given a path, buffer, fuse_fill_dir_t, offset, and fuse_file_info and is used to read a directory. It first checks to verify that the path exists and if it doesn't it returns -ENOENT. It then checks if the inode is a directory, and if it isn't it returns ENOTDIR. If the inode found is a directory all of its sub-file/directory names are retreived using the getShortPath function and these are then added to fuse_fill_dir_t. Upon successful completion modify time for the parent directory is updated and 0 is returned.

int sfs_releasedir(const char *path, struct fuse_file_info *fi);

This function is given a path and fuse_file_info and is used to close a directory. It uses the findInode function to verify the path exists. If the block is not found the function returns -ENOENT, otherwise, the function checks if the path is a directory. If it is not a directory the function returns ENOTDIR, otherwise, the access time of the directory is updated and the function returns 0.