Assignment 4 Design Document

Charles Tzou,David Lung, Payton Javete
CruzID: chtzou, dflung, pjavete

1 Goal

The goal of this assignment was to create an AOFS (All-in-One Folder File System) by using the FUSE file system framework to implement it in user space.

2 Assumptions

The program should only work if the user has FUSE and other UNIX based systems.

3 Design

Our approach to implementing the file system was by first adding all of the necessary functions under static struct fuse_operations hello_oper with print statements and then running commands under the FUSE file system framework to see which commands require which functions to be implemented. We referenced the assignment document as well as this website: http://maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/ (which was used by the TA during a discussion section) to identify which functions FUSE supports and which functions we need to write in order to meet the requirements.

For metadata, we decided the most important information to keep track of was: filename, file size, create time, access time, modify time and which block came next if the file was made up of more than 1 block. If the file fit within one block, next was set to 0 to indicate that it was not linked to any other block. Consecutive files after the first block of a file did not contain the metadata information, and simply contained the next value to indicate whether another block comes after it.

The functionality of each function will be explained more in depth in the Explanation section of this design document.

4 Explanation

1. int main()
    a. First check whether the number of blocks exceeds the maximum number of blocks allowed
        i. If so, return an error message

1. NUM_BLOCKS set to 100 since benchmark requires the creation of 100 files
2. MAX_BLOCKS = ((BLOCK_SIZE - 4) / 4)

b. Also check whether the block size is smaller than the minimum block size allowed
   i. If so, return an error message
      1. BLOCK_SIZE is set to 4096
      2. MIN_BLOCK_SIZE = (sizeof(struct metadata) + 1)
         a. We calculated our metadata structure to be 80 bytes

c. Next, open the file with flags O_RDWR and O_CREAT and truncate the file to the correct size
   i. FILENAME = ./FILE_FS
   ii. File size = BLOCK_SIZE * NUM_BLOCKS

d. Put in the bitmap and magic number into Block 0
   i. magic_number = 0xfa19283e
   ii. bitmap[0] = 1, since the first block (e.g. Block 0) will never be empty since it contains the bitmap and magic number

e. Finally, close the file

2. static int hello_getattr(const char *path, struct stat *stbuf)

   a. First, open the file

   b. Then, read in the bitmap

   c. Next, check that we are provided a path to a file
      i. strcmp(path, "/") != 0
      ii. If the path is to a file, then we use a for loop to go through the bitmap to see if bitmap[i] = 1
         1. This means that the block i is occupied by a file
      iii. lseek to the block and read in the metadata to check if the path matches the file name
         1. If so, set file_found to 1

   d. If the path provided was to the directory, then just set permissions to the directory
      i. strcmp(path, "/") == 0

   e. Else if, file_found = 1, we fetch the data located in the metadata structure

   f. Else, print an error message to tell user that the attributes could not be found for the file

g. Finally, close the file

3. static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi)
    a. First, open the file
    b. If the path provided is to a folder that doesn't exit, close the file and return an error
        i. strcmp(path, "/") != 0
        ii. Error = -ENOENT
    c. filler(buf, ".", NULL, 0) allows us to keep track of the current file
    d. filler(buf, "..", NULL, 0) allows us to keep track of the previous file
    e. Use a for loop, go through the bitmap to see if the block is occupied
        i. bitmap[i] == 1
        ii. If so, lseek to the location of the block and read in the metadata
        iii. If strcmp(md.filename, "") != 0, execute filler(buf, md.filename + 1, NULL, 0)
    f. Finally, close the file
4. static int hello_truncate(const char *path, off_t size)
    a. First, open the file
    b. Next, use a for loop to go through the bitmap and check to see if the block is occupied
        i. bitmap[i] == 1
    c. If so, lseek to the location of the block and read in the metadata
    d. Check to see if the filename matches the path provided
        i. strcmp(md.filename, path) == 0
        ii. If so, set file_start to i, file_size to md.file_size, the access time and the modify time
            1. clock_gettime(CLOCK_REALTIME, &md.access_time)
            2. clock_gettime(CLOCK_REALTIME, &md.modify_time)
        iii. Write the changes to the metadata

iv. Break out of the for loop

e. If the file wasn't found, close the file and return an error value
    i. Return -ENOENT
f. If the size is smaller than the file size:
    i. In a while loop:
        1. lseek to the current block and read in the meta data
        2. Set next_block to md.next
        3. If the size is greater than the USABLE_SPACE, decrement the size by USABLE_SAPCE
        4. Else, break out of the loop
        5. Set current_block to next_block
    ii. Outside of the while loop, set the current_block to next_block in case the while loop broke before current_block was set
    iii. In a while loop:
        1. If the current_block == 0, break out of the loop
        2. Else, lseek to the current block and read in the metadata
            a. Set next_block to md.next
            b. Set bitmap[current_block] to 0
        3. Set current_block to next_block
g. Else if the size is larger than file size:
    i. In a while loop:
        1. Lseek to the location of the current block and read in the metadata
        2. If the size is greater than USABLE_SPACE
            a. If md.next == 0
                i. Use a for loop to go through the bitmap and find a free block
                    1. bitmap[i] == 0
                    2. If so, set nextAvailableBlock to i, bitmap[i] to 1 and md.next to i

3. lseek to the location of the current block and write in the changes to the metadata
4. Set next_block to md.next and perform a memset(&md, 0, sizeof(struct metadata))
5. Then break out of the loop

ii. If a free block wasn't found, close the file and return an error value

1. nextAvailableBlock == 0
2. Return -ENOMEM

b. Else, set next_block to md.next and lseek to that location

c. Set the current_block to next_block and decrement size by USABLE_SPACE

3. Else, break out of the while loop

h. lseek to the location of the bitmap and write in the changes

i. Finally, close the file

5. static int hello_open(const char *path, struct fuse_file_info *fi)

a. First, open the file

b. Next, use a for loop to go through the bitmap and check to see if the block is occupied

i. bitmap[i] == 1

c. If so, lseek to the location of the block and read in the metadata

d. Check to see if the filename matches the path provided

i. strcmp(md.filename, path) == 0

ii. If so, set access time of the metadata to the current time

1. clock_gettime(CLOCK_REALTIME, &md.access_time)

iii. Write the changes to the metadata

iv. Close the file

e. If it exits out of the for loop without finding a file that matches the path, the file is closed and an error value is returned

        i. return -ENOENT

6. static int hello_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
    a. First, open the file
    b. Next, use a for loop to go through the bitmap and check to see if the block is occupied
        i. bitmap[i] == 1
    c. If bitmap[i] == 1, lseek to the location of the block and read in the metadata
    d. Check to see if the filename matches the path provided
        i. strcmp(md.filename, path) == 0
        ii. If so, set a variable file_size to the file size in metadata and the file_start to i
    e. If file_start = 0, this means that a file with the filename was never found, so we close the file and return an error value
        i. Return -ENOENT
    f. lseek to the location of the file we want to read from
    g. In a while loop:
        i. If the read failed, then close the file and return an error value
            1. read(fd, buffer + byte_count, USABLE_SPACE)
            2. return -ENOBUFS
                a. No buffer space available
        ii. Next increment byte_count by USABLE_SPACE
        iii. As long as md.next != 0, we lseek to the next block and read in the metadata
            1. Else, break out of the loop
    h. Next, we need to check the size of the read

i. If the file_size is greater than the offset:
    1. Check if the offset + size is greater than the file_size
        a. If so, size = file_size - offset;
    2. Otherwise, we perform a memcpy(buf, buffer + offset, size);
ii. Else, set size to 0

i. Set the access time of the metadata to the current time
    i. clock_gettime(CLOCK_REALTIME, &md.access_time)
j. lseek to the first block and then write to the metadata structure
k. Finally, close the file

7. int hello_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
    a. First, open the file
    b. Next, use a for loop to go through the bitmap and check to see if the block is occupied
        i. bitmap[i] == 1
    c. If bitmap[i] == 1, lseek to the location of the block and read in the metadata
    d. Check to see if the filename matches the path provided
        i. strcmp(md.filename, path) == 0
        ii. If so, set a variable file_size to the file size in metadata and the file_start to i
    e. If file_start = 0, this means that a file with the filename was never found, so we close the file and return an error value
        i. Return -ENOENT
    f. lseek to the location of the file we want to write into
    g. In a while loop:
        i. If the read failed, then close the file and return an error value
            1. read(fd, buffer + byte_count, USABLE_SPACE)
            2. return -ENOBUFS
                a. No buffer space available

ii. Next increment byte_count by USABLE_SPACE
iii. As long as md.next != 0, we lseek to the next block and read in the metadata
    1. Else, break out of the loop
iv. Set a variable new_file_size to offset + size
v. If the new_file_size is greater than the MAX_REQUEST_SIZE, close the file and return an error value
    1. MAX_REQUEST_SIZE = 128 * 1024
    2. Return -ENOENT
vi. Perform a memcpy(buffer + offset, buf, size), lseek to the start of the file,read in the metadata and set the metadata fields
    1. md.file_size = new_file_size;
    2. clock_gettime(CLOCK_REALTIME, &md.access_time);
    3. clock_gettime(CLOCK_REALTIME, &md.modify_time);
vii. lseek to the start of the file and write to the metadata structure
viii. In a while loop:
    1. If the new_file_size is greater than the USABLE_SPACE:
        a. USABLE SPACE = BLOCK_SIZE - sizeof(struct metadata)
        b. If md.next == 0:
            i. Using a for loop, loop through the bitmap to find a block that is free
            ii. If the block is free, set the nextAvailableBlock to i, bitmap[i] to 1 and md.next to i
            iii. lseek to the location of the current block that we are observing and write into the metadata structure
            iv. Set next_block to md.next and perform a memset(&md, 0, sizeof(struct metadata))
            v. Then break out of the for loop

vi. If there wasn't an available block found, close the file and return an error
    1. Return -ENOMEM
c. Else, if there md.next does not equal to 0 (means that md.next is set to the value of the next block),
    i. Lseek back to the current block we're looking at and read in the metadata
    ii. Set next_block to md.next and lseek to that location and read in the metadata
d. At the end of the conditionals, we will lseek to the current location and start writing the file.
    i. Increment the number of bytes written by the USABLE_SPACE and decrement the new_file_size by USABLE_SPACE
    ii. Set the current_block to the next_block
2. Else, lseek to the current location and start writing the file.
    a. Increment the number of bytes written by the new_file_size and set the new_file_size to 0
    b. Then break out of the while loop

ix. lseek to the location of the bitmap and update it
x. Finally, close the file

8. int hello_unlink(const char *path)
    a. First, open the file
    b. Then, lseek to the location of the bitmap and read it in
    c. Use a for loop to go through the bitmap to find the location of the file we are trying to unlink
        i. If the filename matches the path, then set the bitmap[i] to 0
        ii. Next, use a while loop to go through the connected blocks that hold the file

1. Set bitmap[md.next] to 0
2. lseek to the location of the block and read in the metadata

iii. lseek to the location of the bitmap and write in the changes

iv. Close the file and return 0

d. If we go through the entire for loop without hitting the return inside of for loop, this means that the file doesn't exist, so we just close the file and send an error value

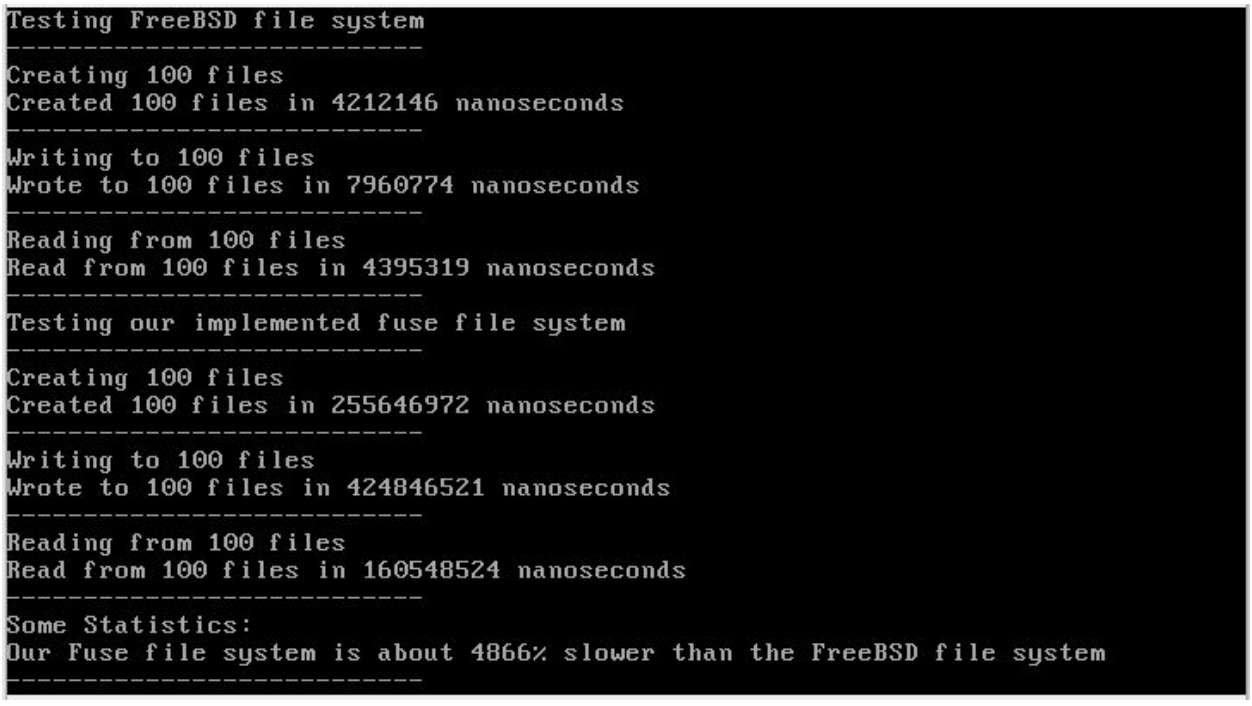
i. Return -ENOENT

9. int hello_create(const char *path, mode_t mode, struct fuse_file_info *fi)
    a. First, open the file
    b. Then, lseek to the location of the bitmap and read in the bitmap
    c. Using a for loop, go through the bitmap and compare to see if the file already exists
        i. If the file already exists, close the file and return an error value
            1. Return -EEXIST
    d. Use another for loop to go through the bitmap to find a free block
        i. If bitmap[i] == 0, this means that the block is free
            1. Set the nextAvailableBlock to i, bitmap[i] to 1
            2. lseek to the location of the bitmap and write in the changes
            3. Break out of the for loop
    e. If nextAvailableBlock == 0, this means that a free block wasn't found
        i. Close the file and return an error value
            1. Return -ENOMEM
    f. Set the offset to nextAvailableBlock * BLOCK_SIZE
    g. If the name of the file we are trying to create exceeds MAX_FILENAME_LENGTH, close the file and return an error value
        i. MAX_FILENAME_LENGTH = 20
        ii. Return -ENAMETOOLONG
    h. Set the filename to the path provided and set file_size to 0

i. Set the create_time and modify_time to the current time
    i. clock_gettime(CLOCK_REALTIME, &md.create_time);
    ii. clock_gettime(CLOCK_REALTIME, &md.modify_time);
j. Set md.next to 0
k. Lseek to the location of the metadata and write the changes
l. Finally, close the file

4 Benchmarking

The issue with an AOFS is that it contains all the files within one directory, negating any possible organization. Additionally, with our code, finding files is much slower as they are not indexed. This means that all of our operations are significantly slower to complete. When we run the benchmark we test the time it takes for the FreeBSD file system to complete three tasks: creating 100 files, writing to those files and then reading from them. We then chdir to our mounted FuseFS (located in newHelloFS) and run the same three tests also noting the time it takes. Finally at the end we take the percentage slowdown for all three functionalities and average them to produce a total slowdown amount. This slowdown amount may vary.



NOTES

ERROR CODES
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/errno.h.html

ENOSYS is the silent failure for each function in the FUSE FS. Seems ENOENT is the similar variable in hello.c (our c file for implementing AOFS).

Line 122 http://fxr.watson.org/fxr/source/sys/stat.h?v=FREEBSD11
Is where **struct stat** is defined. This is what we want for setting labels/flags for a file. Does have fields for setting access and creation time.

Line 299 https://github.com/libfuse/libfuse/blob/master/include/fuse.h
Is where **fuse_operations** is defined and this is what our c file maps to so that is can communicate with the FUSE filesystem.

Line 59 http://fxr.watson.org/fxr/source/sys/statvfs.h?v=FREEBSD11
Seems **struct statvfs** keeps track of stuff having to do with blocks

Line 37 https://github.com/libfuse/libfuse/blob/master/include/fuse_common.h
Is where **struct fuse_file_info** is defined.

Section Notes:
AOFS = 1 root folder and all the files are in it
- -no directories
- -open, read, write, create, delete in the folder

FUSE is a user space file system drive
- -gives you more control over the file operations
- -archive more portability without implementing a kernel module
- -bug may not bring the kernel down

Work Flow
- -when a user wants to create/read/write a file it will send a request to virtual file system and will pass the file operation to the FUSE
- -FUSE will use your logic to perform the operation and then send back the result to the reader

maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/

Important functions to implement
- Getattr
- Unlink
  - -dont need link
  - -used to remove the file
- Open

Read
Write
-can ignore basically all directory files

Basic Structure of AOFS
-easy to implement, bad performance
-entire disk managed by FUSE is contained in one file in the underlying FreeBSD file system
-file system is divided into 4KB blocks
Super block
-a magic number
-a bitmap indicating if block is free or not
-File blocks
-file may have >1 block that need to be chained
-use a pointer at the end of the block to point to the next block
-essentially one giant file that you are treating as a filesystem

-Superblock contains both the magic number and the bitmap
-need to update the bitmap as you write into blocks
-the superblock is made of regular blocks, not its own type

-Each block has 2 parts: file metadata and the file data

Functionality to implement
-create a file
-read a file
-write to a file
-write to file if size >4K
-remove a file

Need to print error message if functionality is unsupported

Call functions like ls through FUSE to see what functions from FUSE are involved
-something about sudo touch???
-printf messages in fuse function like read/write

100 blocks x 4k

lseek used to to find open block
http://man7.org/linux/man-pages/man2/lseek.2.html
Block of data created when file created
First block contains the superblock data

What goes in the metadata (in order):
File Name, Block Length, Creation Time, Access Time, Modified Time, Next Block
(20 B), (4 B), sizeof x 3, (4 B)

For consecutive blocks (e.g. not the first block in the file), set File Name to NULL