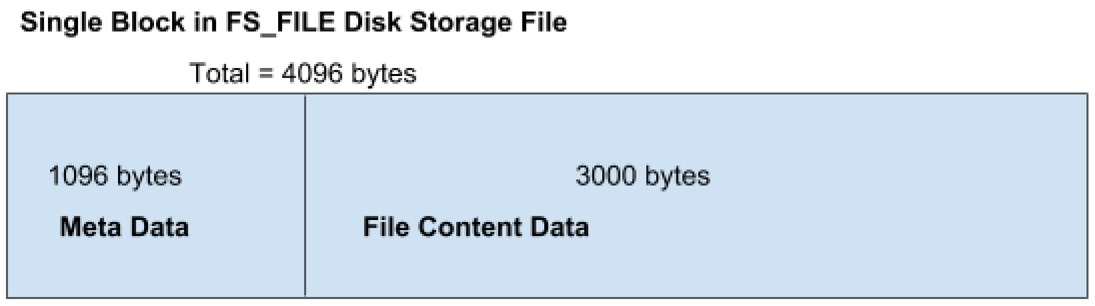
Curtis Lee (cualee)
Brandon Nguyen (brngnguy)
Thomas Zhen (tizhen)

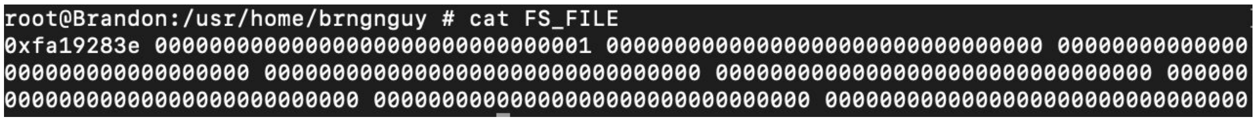
# Assignment 4 Design Document

For this assignment, we were tasked with creating a simple filesystem using FUSE file system framework . This file system is implemented by creating and adding files in one folder. We will call this file system AOFS (All-In-One FileSystem) in short. AOFS is implemented in user space which makes compilation and building the code much faster. We were able to leverage the C user standard libraries for our code. AOFS should be able to support creating files, writing files, reading files, and support files that are bigger than 4KB and reserve an additional block for it. We based our code off of the hello.c skeleton code base which was provided by FUSE and worked from there. In the skeleton base code, a lot of the work in the code was static and wasn't actually stored inside of a file system.

For the design of the code, a block's size is at max 4KB or 4096 bytes. We chose to work with 256 blocks to be able to work with the benchmark's requirements of creating 100 files. This means that our file system should be able to hold up to 1 MB of data (4KB * 256). We created a superblock which holds the magic number and a bitmap of free blocks. The magic number is equal to 0xfa19283e and the bitmap is an unsigned integer array of size 8. We chose 8 because the size of an integer is 4 Bytes which is also 32 bits in total. Since we are working with bits for the bitmap, we divided 256 by 32 bits which results into 8 since each bit represents a block. Within the structure of the code base, we kept a Metadata struct to keep track of the file's details as well as being able to store it into our FS_FILE disk storage file that we initially created at the start up of the code in our main() function. Within our metadata, we keep track of the file's name, size, mode, time accessed, time updated/modified, and time created. We also reserve the first 1096 bytes of the block for meta data and 3000 bytes for the file content data. We also keep a static struct FileSystem throughout the code to be able to make changes to it throughout the code base. Within this File System struct, we keep track of the struct Superblock which contains the bitmap, magic number, and metadata for each file.



For the bitmap vector, there are in total 256 bits and we referenced off of http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html. We were able to define a SETBIT, CLEARBIT, and TESTBIT function for our bitmap. SETBIT takes in a bitmap and index to set the index of the bitmap to 1. CLEARBIT takes in a bitmap and index to set the index of the bitmap to 0. TESTBIT takes in a bitmap and index to check to see if the bit in the bitmap was set to 1. When the bitmap is initialized at the beginning of the runtime, we keep the very first bit intiialized to 1 because the first block in FS_FILE disk storage file is reserved for the superblock with the magic number and bitmap. Below is a screenshot of the inside of FS_FILE disk storage file of the first block at initialization. If bit is equal to 0, block is free. If bit is equal to 1, block is occupied.



```
root@Brandon:/usr/home/brngnguy # cat FS_FILE
0xfa19283e 00000000000000000000000000000001 00000000000000000000000000000000 00000000000000
000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 000000
00000000000000000000000000 00000000000000000000000000000000 0000000000000000000000000000000
```

There are 8 32-bit blocks in the bitmap with the first 32-bit block having a 1 at the end of the block representing that block is reserved for the superblock with the magic number and bitmap.

***It is also important to note that when running our file, the grader needs to run “sudo ./hello newHelloFS -f”* The -f is important as it runs it foreground and cwd is changed to / https://sourceforge.net/p/fuse/mailman/message/30477051/ ***

The functions that we created and used for this assignment are listed below:

- main()
- filesys_init()
- superblock_init()
- filesys_load()
- filesys_find_file()
- filesys_write_bitmap()
- aofs_getattr()
- aofs_readdir()
- aofs_create()
- aofs_write()
- aofs_open()
- aofs_read()
- aofs_unlink()
- aofs_truncate()

The system calls used for this assignments are listed below:

- fwrite()
- fclose()
- write()
- read()
- close()
- open()
- ftruncate()

- lseek()
- sprintf()
- printf()

***IMPORTANT TO NOTE: That all functions that mention writing to FS_FILE is using the open(), close(), write(), and lseek() system calls.***

main()

- Our main function is what starts up the FUSE file system structure but before we go into using the FUSE apis, we needed to create our FS_FILE disk storage file where all of the files are stored inside here. When we create our FS_FILE, we call filesys_init() function to initialize the superblock and filesystem struct to be used throughout the whole code base. We pass the file system, number of blocks (256), and max block size (4096 bytes) into filesys_init to initialize the super block and file system. After initializing the file system and superblock, we use the FUSE apis.

filesys_init()

- This function accepts the filesystem struct, number of blocks, and block size as parameters. We truncated the FS_FILE storage to a specific size which is 256 * 4096 = approximately 1 MB. When we truncate the FS_FILE, all of the contents of the file is initialized to 0. After truncating the FS_FILE, we call superblock_init to initialize the superblock and write it to FS_FILE into the first block.

superblock_init()

- We initialize the bitmap bits to 0 and initialize the first bit of the bitmap to 1 by using the SETBIT() call since the superblock is reserved for block 1. We also initialize the magic number to 0xfa19283e. We write to FS_FILE with the magic number and values of all of the bits in the bitmap by using the lseek() system call to point to the offset of the FS_FILE and write() system call to write to the FS_FILE.

filesys_load()

- For filesys_load, it accepts a filesystem struct as a parameter and this is only called if the FS_FILE is already initialized. Initially this is suppose to be used for loading the already existing contents of FS_FILE into the bitmap but since this wasn't required for our assignment, we left it there as testing purposes so that we didn't have to keep deleting and creating the FS_FILE.

filesys_find_file()

- This function accepts a filesystem struct and a filename as parameters. This function is used to return the index of the file name only if it the bit inside of the bitmap is initialized to 1 and the file name matches with the file name in our data structure of the Metadata. Otherwise, if it is not found, we return -1 signaling that the index was not found.

filesys_write_bitmap()

- This function accepts a file system struct as a parameter and writes into FS_FILE by using lseek() and a for loop to write to the FS_FILE with the correct 0 and 1 bits from the bitmap by using the write() system call.

aofs_getattr()

- This function is connected to FUSE's getattr API and accepts a path and stat as parameters to the function. When we get the path, since we are working with files and not directories, we are able to extract the file name from the path. First, we check to see if it is a root directory from the path and if it is, we return the permissions and attributes to the root directory. Otherwise, we call filesys_find_file() and pass the file system and file name into the parameters and if the filename is found inside of the file system, we return the index of that file. If the index that was returned isn't equal to -1, then we set a flag that the file was found. We check to see if the flag was found and if it is, we return the correct attributes of that file in the file system. Otherwise, if the file was not found in the file system, we return -ENOENT which means that path/file does not exist in the file system. Returning -ENOENT allows the passing of the create() function if the user is trying to create a file that isn't in the file system.

aofs_readdir()

- This function is connected to FUSE's readdir API and we only account for the path and filler as the important parameters. We use filler to put the current directory and parent directory when a user tries to ls into the directory. We use a for loop to iterate through the blocks and check if the length of the metadata filename is not 0. If it's not 0, we use filler to put the file's name.

aofs_create()

- This function is connected to FUSE's create API and used to create the file into the FS_FILE disk storage file. This function has two parameters we use which are path and mode. We extract filename from path and use the mode to store the file's mode. First, we want to check to see if there is an available free bit inside of the bitmap by using the TESTBIT(). When there is a free bit, we use that index to store the file within that block with the index as the number of block. We also keep track of the time creation and time accessed of the file. After finding a free block to create our file into, we want to write into the block's metadata which is reserved for the first 1096 bytes. To find FS_FILE's offset of the free block, we multiply the index by the max block size (4096) to iterate through the blocks in FS_FILE. We use lseek() system call to position the cursor of FS_FILE to the offset so that we can write the metadata into it. We write the file name. File size, block index, mode, time creation, time updated, and time accessed into the metadata. After writing to FS_FILE's block with the metadata, we then reinitialize the FS_FILE's offset by adding 1096 bytes into it to write into the last 3000 bytes of the block. We then proceed to use lseek() system call for the new offset that was calculated and write the file's content data into it which is an empty buffer since this is an empty file. After writing to the file content data, we call SETBIT() to initialize the bit to 1 from the bitmap. We also store the file's metadata and information to the metadata data structure and call ftruncate() system call to keep the size as 1MB for FS_FILE since the size of FS_FILE seems to change and go smaller if we write without calling this. We then call filesys_write_bitmap to update the first block in FS_FILE with the updated bitmap.

aofs_write()

- This function is connected to FUSE's write API and is used to write to files in the FS_FILE disk storage file. This function has 3 parameters that we use which are path, buf, and size. We extract the filename from path and the buf is the file content data being written to the file. The size is the length of bytes of the buf. First, we use filesys_find_file() to find the index of the filename that is being requested to write to and make sure that the file is located in our FS_FILE disk storage file. If the index returned is equal to -1, then we return -1 for aofs_write() to indicate that the file to write to isn't located in FS_FILE. After we find the first index, we want to check to see if the size of the buf is greater than 3000 (max block size (4096) - meta range (1096)) and if it is, we will need a second free index/block to hold the rest of the file content data. There are two cases that we have when we do this. Case 1 is if buf size is less than or equal to 3000 and case 2 is if buf size is greater than 3000.
- **CASE 1**: This is the case where the file content data does not exceed 3000 bytes.
  - We keep track of the time for time updated and time accessed since these change whenever we modify and access a file.
  - We set the FS_FILE offset to index * 4096 and use lseek to position the cursor to the correct offset of the block number and it's metadata content.
  - We then proceed to write to the metadata content with the file's metadata.
  - After writing to the metadata content, we look into writing into the file content data by adding FS_FILE offset an additional 1096 (metadata size) which will bring the offset to the file content data position.
  - We use lseek() to position the cursor to the correct offset of the file content data and write the buf content in there up to the size amount.
  - After, we store the metadata into the metadata data structure as well as truncating the file after writing to make sure that the file size doesn't drop below 1 MB for whatever reasons.
  - We then call filesys_write_bitmap() to update the first block in FS_FILE with the updated bitmap.
- **CASE 2**: This is the case where the file content data exceeds 3000 bytes.
  - We only go into case 2 by keeping a flag to see if the buf's size exceeds 3000 bytes.
  - We keep track of the time for time updated and time accessed since the change whenever we modify and access a file.
  - Since our buf over exceeds 3000 bytes, we want to split the buf into two char arrays so that we will be able to write to the first block and the next free block that can hold the rest of the file content data.

    - We used a for loop to store the first 3000 bytes of the buf into a char array buf1 and store the leftover bytes into a char array buf2
    - First, we will look into writing into the metadata of the first block by multiplying the index by 4096 which becomes the FS_FILE offset and using lseek to position the cursor of FS_FILE there.
        - This holds the true file size that over exceeds 3000 since our data structure will be pointing to the next block that contains the rest of the file content data.
    - Second, we will look into writing in the file content data of the first block by adding 1096 bytes into the FS_FILE offset and use lseek to position the cursor of FS_FILE there.
        - We use the buf1 char array which contains the first 3000 file content data to write into the first block.
    - Third, we will look into writing into the metadata of the second block by multiplying the second block index by 4096 which becomes the FS_FILE offset and using lseek to position the cursor of FS_FILE there.
        - This holds the leftover file size.
    - Fourth, we will look into writing into the file content data of the second block by adding 1096 bytes into the FS_FILE offset and use lseek to position the cursor of FS_FILE there.
        - We use the buf2 char array which contains the leftover file content data after buf1 and write into the second block.
    - After successfully writing into the two blocks, we call SETBIT() to set the bit of where the second block was used.
    - We update the metadata data structure and assign the second block's index in the metadata data structure so that when we go into aofs_read(), we will be able to read all of the file content data.
    - We then truncate FS_FILE to make sure the size of the file is 1 MB incase whatever happens to the file.
    - We then call filesys_write_bitmap to update the bitmap in the superblock.

aofs_open()

- For this function, we use the path parameter passed into this function to extract the file name.
- We only check to see if the file name exists in the file system by calling filesys_find_file by passing the filesystem and file name.
- As long as the return value of filesys_find_file isn't -1, we record time accessed and return 0 for successfully found.
- Otherwise, we check if the requested permissions are available and if they aren't, we return -EACCES which means requested permission isn't available.
- Otherwise, we return -ENOENT which means the file isn't located.

aofs_read()

- For this function, we use the path, buf, and size parameters. The path is used to extract the file name, buf is used to return the file content data read by using the read() system call. First, we use filesys_find_file to check to see if the file exists and if it does, it returns the index of the found file. Before we go into reading the file content data, we check to see if the metadata data structure's contains a second block index called nextBlock. This indicates that a file needed a second block to write it's overflowing file content data which exceeded 3000 bytes. This splits into 2 cases for aofs_read().
- **CASE 1**: Did not need an additional block to write data to.
    - First, we keep track of the time accessed since we are accessing the file to read content.
    - We multiply the index by 4096 bytes to get the FS_FILE offset and add it by 1096 bytes to reach the file content data part of the block and use lseek() to position the cursor in FS_FILE.
    - We then use the read() system call and read the file content data into buf where buf will now contain the file content data.
    - After reading, we return the amount of bytes read into buf.
- **CASE 2:** Exceeded the 3000 bytes for file content data and needed an additional block to write data to.
    - First, we keep track of the time accessed since we are accessing the file to read content.
    - We use 2 char arrays called buf1 and buf2 where buf1 will read the first 3000 bytes in the first block and buf2 will read the leftover bytes into buf2.
    - We set FS_FILE offset to index of first block * 4096 and add it by 1096 bytes to reach the file content data position and use lseek() to position the cursor in FS_FILE to the offset.
    - We then proceed to use the read() system call to read the data into buf1 but we make sure to read 3000 bytes only otherwise it becomes Input/output error.
    - After reading from the first block, we set FS_FILE offset to index of second block * 4096 and add it by 1096 bytes to reach the field content data position and use lseek() to position the cursor in FS_FILE to the offset.
    - We then proceed to use the read() system call to read the data into buf2 but we make sure to read the leftover bytes.
    - After reading into buf1 and buf2, we look into combining buf1 and buf2 into buf so that it can be returned to the user which is implemented using a for loop.
    - We return the file size that is read in total.

aofs_unllink()

- For this function, it only has one parameter which is the path and we extract the file name from the path. We first call filesys_find_file to search and check to make sure that the file exists in the file system or else we will not be able to remove or delete a file. We initialize two empty char arrays of size 1096 and file size of the found file of 0. We then check to see if the file we want to delete used an additional block to write its content data. This separates it into two cases.

- **CASE 1:** File did not need an additional block to write its file content data.
    - First, we set offset of FS_FILE to index * 4096 and use lseek() to position the cursor of FS_FILE to the offset.
    - We write into metadata content data of all 0's and empty to make it a free block.
    - We set the offset of FS_FILE by adding an additional 1096 and use lseek() to position the cursor of FS_FILE to the offset.
    - We write into file content data of all 0's and empty to make it a true free block.
    - We use CLEARBIT() to update the bitmap and set it to 0 to indicate that the block is free and also clear out the metadata data structure as well.
    - We call ftruncate incase if FS_FILE size drops and call filesys_write_bitmap() to update the superblock.
- **CASE 2:** File needed an additional block to write its file content data
    - Since an additional block was needed, we set an additional empty buf char array of size 3000 to all 0's.
    - First, we set offset of FS_FILE to first block index * 4096 and use lseek() to position the cursor of FS_FILE to the offset.
    - We write into metadata content data of all 0's and empty to make it a free first block.
    - Next, we add 1096 to the FS_FILE offset to write to its file content data and use lseek() to position the cursor of FS_FILE to the offset.
    - We write into the file content data of all 0's and empty to make it a truly free first block.
    - Second, we set offset of FS_FILE to second block index * 4096 and use lseek() to position the cursor of FS_FILE to the offset.
    - We write into the metadata content data of all 0's and empty it to make it a free second block.
    - Next, we add 1096 to the FS_FILE offset to write to its file content data and use lseek() to position the cursor of FS_FILE to the offset.
    - We write into the file content data of all 0's and empty to make it a truly free second block.
    - After successfully emptying these blocks, we use CLEARBIT() twice for the first and second blocks to initialize the bits back to 0 to indicate free bits.
    - We then call ftruncate to truncate FS_FILE incase of any weird size issues and use filesys_write_bitmap to update the bitmap in the superblock.

aofs_truncate()

- This function did not have much purpose except for when a user is trying to edit an already existing file. This is needed after writing to a file. We just return 0 upon success.

**Benchmark**: For our benchmark, we used a shell script csh file since FreeBSD uses csh as it's default shell. We created 100 files within the benchmark and write some text into it and read them as well. We made it so that the benchmark is able to print simple text to a file but in order to test for files that are bigger than 4KB, we suggest that the grader manually creates a file with byte size of about 4000 to test that functionality. We designed the code to only handle buffer sizes of at most 6000 bytes. The Benchmark contains three parts: create 100 files, write to 100 files and read from 100 files. When you want to use the Benchmark, first do sudo, them use the mv command to move the Benchmark into the newHelloFS directory, and then sh Benchmark.sh.