Project4: Tiny File System
Operating Systems (CS 416)
Sudarsun Kannan
David Domingo Michael Wu

Alfonso Buono(ajb393)
Ali Mohamad(aam345)

Due: May 3rd, 2020

1   Introduction

The goal of this project was to teach both persistence data and how file systems work. We were tasked with creating a tiny file system using FUSE and could perform basic file system tasks (such as creating directories, removing directories, creating files, editing files, etc). Furthermore, the project had to be persistent in the sense that if we were to unmount our file system and then remount it, the data should still exist. This is done because our File System treats a file, called DISKFILE, as our file systems disk. Here we store all of the information that is needed in a file system, such as the super block, bitmaps, inode table, and data segment. Reading and writing to this file is how we keep track of our data and make sure that everything is persistent. Overall the project was challenging but fun. We learned a lot about how a file system works and how certain file systems function. Below we describe how our code works, how to compile, some issues that we faced, and the benchmarks that we ran.

2   Summary of Code

The general idea of our code is that we have a DISKFILE that is acting as our disk. Here we separate our DISKFILE into different chunks or blocks. In our file system we made the 0th block the super block. This would hold information in regards to our file system and information that we will need throughout the program. Next we have two bitmaps in the 1st and 2nd blocks. These are for the inodes and for the data blocks. We want to keep track of which inodes and which data blocks are in use and which are free to use. Next we have the inode table. This spanned from the 3rd block to the 66th block. We found this out by using the MAX_INUM definition given to us. In these blocks we store inodes. There can be 16 inodes per block. The last section is from block 64 and onwards. These are all of the data blocks that will hold either directory

information or the information needed for a file.

Our code was split into different sections. The first section that we had to create were the sections that dealt with the DISKFILE. These functions were what we did in order to find new inode numbers, or block numbers as well as writing inodes and reading inodes from the disk. These were a foundation to the project as it was imperative that we could write and edit inodes. They are similar to the bio_read() and bio_write() that were given to us except we had to utilize these two functions in order to make us able to write in granularity inode.

The next section is mostly the helper functions that were called and used repeatedly. This included the directory functions(dir_add(), dir_find(), and dir_remove()). These functions were very important as it allowed us to find entries inside a directory, add entries and also remove them. This was done by going to an inode and checking the direct pointers. We check their direct pointers and then are able to get the appropriate data blocks. In these blocks we then can loop through 19 times (this 19 is the number of dirents we found that can fit in a block) and try to match the dirents to what we need. Depending on our goal we can add or remove the dirents (once we check if it either does or does not exist). These functions were important in keeping track of what existed in directories (as we write the dirents and their corresponding data blocks back to disk).

Another function we included was trimBlocks(). This was a function that was called by rmrdir. This would remove data blocks that had no valid dirents and would help free up data blocks.

Finally once we finished all of the helper functions we were able to move onto our tfs functions. These functions were called by FUSE and in turn these functions would call our helper functions.

All of our information is initialized in tfs_mkfs(). This function is called once by tfs_init() (which is called when our file system is mounted). Here we create all of the important information, such as the bitmaps, and then write it to the DISKFILE in order to make sure that the data is persistent. tfs_init() runs in order to make sure that we have the DISKFILE as well as we ready up any in memory datastructures (in our case the only in memory data structure is our super block pointer). After this has run then depending on the command given to FUSE certain tfs functions would run. Most of the functions were very similar in the sense that they might call get_node_by_path() to get the inode of a specific file or directory given the path. With this we can then check to make sure that file or directory exists, add to it or remove from it. These functions make a combination of calls to both our directory functions as well as keeping track of inodes using our readi() and writei() functions. This is to make sure that our data stored in the data blocks as well the data stored in our inode table are consistent and to make sure all of it is properly stored on the disk. The two functions that were a little harder to implement were tfs_write() and tfs_read(). These functions were very similar to get and put in project 3 (Virtual Memory) as we were either given data to write to the disk or were asked to get data from the disk and write it to a buffer back to the user. In these functions we had to

make sure the file had enough memory to store the information, by comparing how much they were writing to how much the file has, as well as making sure that the amount the user wanted to write was not over our file limit (if this was the case we just wrote as much as we could until the file filled). We also had to make sure that we were properly using offsets and that we only used it for the first block read or write. If the data that was being written was greater than the current file size then we had to allocate more memory for the file (up until the the 16* BLOCK_SIZE file limit).

In the end our tfs functions were not as much code as they were calling all of our helper functions to make sure that our data was persistent as well as the same for both the data stored in inodes (making sure each directory or file had an inode) and the data stored in the data blocks (making sure that files and directories in other directories matched the inode data).

3   Compiling Code

To compile the code you can do as follows:

1. make clean

2. make

3. (if you need a fresh disk) rm DISKFILE

4. ./tfs -s /tmp/ajb393/mountdir (/tmp/ajb393/mountdir is the path to the mount directory)

    (a) If you would like to run in debug mode:
        ./tfs -s /tmp/ajb393/mountdir -d

To unmount you need to type the following:
fusermount -u /tmp/ajb393/mountdir

4   Issues Faced

While working on this project there were some issues that seemed to pop up. Whether it be our initial understanding of FUSE and File Systems, to just not coding correctly, we recognized we needed to fix a lot of things (and we think we did correctly). The first issues that we were facing were understanding FUSE. While we had some semblance of what it was we did not comprehend how it worked and the associated function calls that were say with the command "mkdir". However, we were instructed that the best course of action would be to implement a pass through file system. This would allow us to understand FUSE better and conceptually grasp what we had to do. So, after creating the pass through file system we began working on the actual project. From here we ran into some issues with reading and writing to files. For some reason when we use vim, or even echo, to add data to a file the offset passed to tfs_write() is

4096. This does not make much sense as even if we are writing onto the same line or just after the original 3 characters I do not understand why it would offset by a full block. Due to this, we are having issues where each consecutive write to a file causes the file to create a new block. However, the read and write's appear to be working as you can re-vim a file and the data is there or you can cat out a .txt file and the data comes out. On top of this we have one more issue that only appears to be of our concern not in the benchmarks (I assume this because we pass our benchmarks). When I write to a file and then cat out the file, for some reason there exists another new line. This is strange and I am unsure as to why this is happening. After testing repeatedly (and emailing David my thoughts) we came to the conclusion that it had to do with the current file size. When you append to a file tfs_write() is called with an offset of the current file size. Since we were originally only writing 3 bytes to a file but saying the size was 4096 the offset being sent to the function was 4096. This was causing issues with the data we were reading and writing as well as constantly creating more blocks. To combat this we changed it so that the size of the file gets updated with the new changes and is in bytes that we have actually written (as opposed to the original granularity of bytes we have allocated). After making these changes the benchmarks still worked, cat-ing a file did not have the weird new line and appending also correctly added data as well as did not allocate extra blocks.

Finally, while we did not complete the large files we were able to make dir_add(), dir_find(), and dir_remove() to work. Due to time constraints and the issues with the ilabs we were not able to finish. Our implementation that we were doing and hoping to accomplish goes as follows:

When a file or directory runs out of memory in their original direct_ptr data blocks then we go to their indirect_ptrs. Here we check to see if any of them have a corresponding data block number (we initialize them to -1 so it is easy to check if we have set them). If they do not, then we allocate a new data block and set the indirect ptr value to that new data block. We then allocate another data block and store that block number in the indirect block (we stored all data block numbers as if it was a linear array in the indirect block). From here we then just treated the data block as if it were a direct_ptr data block and would either store dirent there or user data. Theoretically this would immensely increase the size of the files and directories as each indirect block could hold $\frac{4096}{4} = 1024$ pointers to data blocks. The functions that we did not finish we had worked out what we believed we needed to change and did not think it was a lot more work to do. Our extra credit attempt is saved in: tfsEC.c.

5   Benchmarks

When running the benchmarks we get the following results:

1. For simple_test we pass of the benchmarks. Furthermore, we got our code to write out the number of data blocks that are currently being used. The amount we got was 7. When checking what simple_test does, we can see

that 7 seems to match with what is in our file system. First root points to a data block that holds the subdirectory files. Thus we have 1 data block. Then inside files there are 100 directories. Here $\frac{100}{19}$ gives us 5.2. This means that there should be about 6 blocks in total to hold all of these directories (their dirents) inside the directory "files". Therefore we have $6 + 1$ data blocks. Our speed tests are shown below in this screenshot:

```
[ajb393@ls benchmark]$ time ./simple_test
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: File unlink success
TEST 6: Directory create success
TEST 7: Sub-directory create success
Benchmark completed

real    0m0.049s
user    0m0.000s
sys     0m0.005s
```

2. For test_case we pass all of the benchmarks except for large files. This is because we did not implement large files. When checking the number of data blocks that were used in this benchmark we came up with the number 23. This appears to be right when we take a look at the actual benchmark. First we get the same number of 7 from simple_test. Then we have a file named largeFile also in root. Here I am assuming that all 16 of its direct pointers point to data blocks and thus $7 + 16$ equals 23. Our speed tests are shown below in this screenshot:

```
[ajb393@ls benchmark]$ time ./test_case
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: File unlink success
TEST 6: Directory create success
TEST 7: Directory remove success
TEST 8: Sub-directory create success
TEST 9: Large file write failure

real    0m0.098s
user    0m0.004s
sys     0m0.007s
```