Operating Systems Project #4: Documentation

Jack S. Dulin jsd177
Carl Costanza cc1533

Preface:

Our group attempted the extra credit implementation by filling the indirect pointers within each inode when necessary. The first draft of this documentation was submitted with only the regular credit completed, and the changes that were required to go from the regular credit to the extra credit will be discussed side-by-side.

Running the Benchmark (Regular Credit):

The benchmark was run five different times in order to get an accurate sampling of the time it takes to run the benchmark. The five times that were recorded are the following: 12,121 microseconds, 19,335 microseconds, 417,606 microseconds, 14,423 microseconds, and 11,567 microseconds. As demonstrated by these samplings, the benchmark usually takes about 10,000 to 20,000 microseconds to run, but there was one outlier.

When the benchmark was run, the first thing that was done was to create the "/file" directory for test #1. In order to initialize the file system, the tfs_init() and then the tfs_mkfs() commands had to be run. The superblock, inode bitmap, data block bitmap, one inode directory block, and one data block had to be allocated before the root directory, "/", could be created. In order to add the "/file" directory to the root directory, only one more additional data block had to be allocated for the "/file" directory. So far, this leaves us with six disk blocks that have been allocated. The next thing that was done in the benchmark was to write 16 data blocks to that file for test #2. Since one of the data blocks has been already allocated when the file was created, only 15 more additional data blocks were allocated for the file write operation, leaving us with 21 total blocks allocated. For tests #3 and #4, which consisted of closing the file and then reading the file, no additional data blocks were allocated. For test #5, the file was deleted, which deallocated 16 of the disk blocks that were previously allocated for "/file." Test #6 required one directory to be created, yielding another inode entry in the inode directory (since there are only two entries in the inode directory at this time, no need to allocate another block) and another data block allocated for the "/files" directory, creating one more data block, bringing the total up to 22 blocks. In test #7, 100 subdirectories were created within the "/files" directories, adding 100 more inode entries to the inode directory and allocating 100 more data blocks (one for each sub-directory created). Since 100 more inode directory entries were created and there are 16 inode entries that can be stored in a single block, 7 more blocks were created just for new inode directory entries on top of the 100 disk blocks that were created (one for each directory). Therefore, the total number of disk blocks that were allocated (if you don't subtract the ones that

were deleted after the file unlink operation) during the execution of the benchmark was 22 + 107, or 129 total disk blocks.

Running the Benchmark (Extra Credit):
In order to test whether the implementation can indeed store indirect pointers, simply change the ITERS value to 100 (as it previously was). This will verify that you can write 100 blocks to a single file, which is more than the 16 that would be allowed using only direct pointers and no indirect pointers. ***<give the five new times here once you are finished>***

Implementation/Challenges Faced (Regular Credit):
In order to determine how to properly store inodes in each of the inode directory disk blocks and dirent structs in each of the directory disk blocks, we had to find the size of the inode structs and the dirent structs. We found that both the inode struct and dirent structs were 256 bytes each. Since the size of each of the disk blocks was 4,096 bytes (which is also the size of the BLOCK_SIZE macro), we could store up to 16 inode structs in each of the inode directory disk blocks and we can store up to 16 dirent structs in each of the data blocks of a directory. Even though the superblock, inode bitmap, and data block bitmap took less than one block each, we allocated an entire block for each of them for ease of implementation. In summary, block 0 was the superblock, block 1 was the inode bitmap, block 2 was the data block bitmap, blocks 3 - 66 were used to store all of the inodes in the inode directory, and blocks 67 - 8,192 were used to store user data.

To ensure that the disk block wasn't corrupted, we ensured that the superblock had the correct magic number at the beginning of each execution. If the correct magic number wasn't found, then the disk would be reformatted. While the FUSE functions were executing, no global-in memory data structures were used. Instead, we allocated in-memory buffers as they were needed (to store an inode or a bitmap, to read to or write from a data block, etc.) and then they were deallocated as soon as they were finished being used to prevent any potential memory leaks. The valid attribute found in the inode struct and the dirent struct was used to determine if a file has been deleted or not. If the file has been deleted, the valid attribute would show a "0" and a "1" if the file hasn't been deleted yet (and is therefore still valid). The type attribute of an inode was a "0" if the inode in question referred to a file, and it was a "1" if the inode referred to a directory. The link attribute is always equal to 1 for files since we aren't dealing with hard or soft links for this project. For directories, the link attribute was equal to 2 plus the number of subdirectories found within the inode. Within the stat structure found in the inode, only the main attributes that were deemed important on one of the Piazza posts were filled out and that list of attributes included: st_ino (inode number), st_mode (gives information about whether it's a file or directory along with its permissions), st_size (size of the file), st_blksize (size of each block in the file system), and st_blocks (the number of blocks that the file takes up).

There were two great challenges that we faced when trying to get this file system up and running. The first one was encountered when we tried to test tfs_mkdir() because FUSE kept saying the directory was not found after we made the directory. In order to solve this issue, we had to use side-by-side window debugging with GDB along with print statements within the tfs_mkdir() function. The print statements weren't executing at all when we tried to execute the mkdir command in the second window. After a little bit of searching online, we found out that tfs_getaddr() was called before tfs_mkdir() in an online FUSE forum, something that wasn't very clear to us when we first embarked upon this project. In order to solve the issue, we fully implemented tfs_getaddr() and then returned the right error code in the case that a file or a directory wasn't present (in this case, it was -ENOENT, which signified that the file or directory didn't exist). Following this series of steps allowed the mkdir command to work properly.

The last major challenge was faced when test #7 failed while running the benchmark. Upon further investigation using GDB side-by-side window debugging, it turned out that test #7 worked until the benchmark tried to create the seventeenth sub-directory in the "/files" directory. It turns out that this happened because the direct_ptr[1] attribute within the inode was still equal to -1 even though that shouldn't have been the case. We found that the cause of this bug was because the inode struct passed into the dir_add() function wasn't actually modified because it was passed by value, not passed by reference. In order to solve this, we slightly modified the method signature of dir_add() so that the first parameter was a struct pointer instead of a regular struct. Making this modification allowed the struct to be properly modified, and then all of the test cases successfully worked after that modification. Additional details with regards to the implementation can be found in our source code in the tfs.c file.

Additional Changes to Implementation (Extra Credit):
For each inode, the maximum number of blocks that can be used to store a file or a directory is changed. Instead of only being able to store 16 blocks as in the regular credit, supporting indirect pointers would allow a file or directory to be able to store 16 + 8(16)(16) possible blocks. In addition to the direct blocks, there are 8 indirect pointers. Within each of these indirect pointers, there is a number that stores the data block which contains other direct pointers. These data blocks, which are BLOCK_SIZE (4,096) bytes, can store up to 16 inode structs (which are each 256 bytes) and each of these inode structs store up to 16 direct pointers. Therefore, in addition to 16 direct blocks, there are also 8(16)(16) additional blocks for each inode thanks to using indirect pointers.

Compilation Set-Up (Regular and Extra Credit):
In order to properly compile our code, the mount directory has to be created if it hasn't been created already. This would be done using the command "mkdir /tmp/jsd177." If that command

worked (meaning this directory wasn't created before), we would then run "mkdir /tmp/jsd177/mountdir". After that's done, modify the TESTDIR variable found in the simple_test.c file in the benchmark folder. Since we only attempted the regular credit, the ITERS macro should be changed to 16 since we can only store up to 16 direct pointers for each file. From there, we would go back into the project directory and then run the command "make" and then run the simple_test.c benchmark after making the simple_test executable. To determine the amount of time that the benchmark took to execute, we imported the <sys/time.h> library and then we created two timeval structs to keep track of the amount of time it took the file to execute. After testing is done, clean all executables, unmount the directory, and then delete the disk file by issuing the command "rm DISKFILE".