**ThreadOS Final Project Report**

**CSS430: Prof. Dimpsey**

**Members:** Ji Kang, William Eng

**Synopsis of testing**

For the purposes of debugging and ensuring that our file system had correct functionality and followed the document’s specifications, we initially only used Test5.class to test by loading it into the ThreadOS console via the command line.   
Passed all Test5.class tests.

**Format Testing**

It passed formatting of 48 total and compared the location of totalBlocks, inodeBlocks, and freeList variables by reading in an integer from the formatted superblock in offsets of 4 (integer = 4 bytes). Taking this one step further, I did further formatting tests in FSTest.java which can be found in the submission folder. I did further format testing with 16 files which passed the testing like initially. Since 16 inodes can fit in one block, we wanted to see what would happen if we attempted to format the block using a number of files that weren’t cleanly divisible by 16. To see the results, we used a file number of 28 to which would offset the freeList block number by 1. Passed this testing further.

For testing afterwards for consistency results and to avoid any further complications, we formatted the block to 64 blocks.

**Open Testing**

Passed the Test5 test of just opening a single file in w+ mode. In our FSTest, we opened multiple files in different modes to see the dependencies. Initial basic testing including opening multiple files in ‘w’, ‘w+’, and ‘a’ mode. Attempting to open a file in read mode while another process has it opened in write (Would cause a wait() – this was taken out but we used to see it working functionally correct). We tried to open a file in ‘r’ mode that didn’t exist yet and another attempt to open a file in ‘r’ mode that exists and was previously closed. All of our initial testing worked and we utilized the file descriptor integer value returned from each to ensure that the allocation of inodes worked correctly. For this test, we opened a total of 3 files

**Write Testing**

Passed all of Test5’s tests involving SysLib.write(…). Using one of the files we opened previously, we did a series of writes. Test5 did things very similarly so we followed the same structure. We saw Test5 wrote out a file size of 6656 bytes so we assumed that any additional testing that went to that magnitude would follow the same behavior. We did a series of writes just to ensure consistency was upheld by writing a bunch of bytes and checking the amount of byes written to the file. This is further propagated in our Read tests. For simple amount of writing, we wrote out 16, 256, and 314 bytes (To commemorate pi-day). With the max file size being 267 blocks, we were well in the range of a file’s size limits.

**Close Testing**

Test5 had SysLib.close(fd); calls scattered throughout to ensure other testing so functionality-wise the method calls were working as intended. We felt not too much testing was needed to ensure SysLib.close was functioning properly so we closed our remaining files and attempted to write to the files even after they were closed by comparing the size returned (which is equivalent to bytes written). Any size above 0 is invalid.

**Read Testing**

Passed all of Test5’s test which were very intensive. Anywhere from reading a small amount of bytes to very intensive (6000+ bytes in comparison). Consistency checks seemed to pass regardless of how many bytes seemed to be passed in. Only further testing we did was to check the consistency of our previous writes which wrote 586 byes of increasing numbers (as far as a byte would allow by doing n % 128 so we don’t have overflow). If it passed these consistency checks, we assumed our SysLib.read() was working properly.

**Seek Testing**

Test5 seek’s have all passed. Test5 seemed more oriented around correct positioning within the file with the seek calls so we tested some edge cases that were mentioned in the document for the assignment. We attempted to set the seek pointer to a negative number and as a number passed the file’s size to ensure our correction cases were working properly. The tests showed the seek pointer were clamped at 0 for negative numbers and clamped at fsize(FileTableEntry) to ensure we have consistency and no fragmentation with our files.

**Fsize Testing**

Test5 didn’t necessarily go over testing over size other than the size of byes written to/read from a file. With our initial file that we wrote 586 bytes to, our fsize call should’ve returned the same number. We didn’t go into extreme cases like seen in Test5’s cases with write and read so perhaps there may be some limitations since our tests were limited for fsize. But overall, it seems to be functioning as intended otherwise. Just speculations at this point.

**Delete Testing**

Test5’s deletion tests were passed. Tested it further with just additional files in our test cases. We couldn’t really think of edge cases. We tried to delete a file that didn’t exist and our file system handled it by not really doing anything which is what seems to be intended. A command that can’t be executed should be ignored.

**Possible Limitations and Assumptions**

One of the things we didn’t necessarily test for is making the entire file system formatted with inodes. Excluding the superblock, there should be 999 blocks which is 511488 bytes total. This equates to 15984 inodes possible. The formatting would take a substantial amount of time and didn’t check what would actually happen if we tried this so this is one edge case we didn’t look for. We would assume the user actually wants to use the file system and be able to store files so this extreme edge case wouldn’t necessarily happen in a real use-case of the program.

We were able to successfully write the max file size of 267 blocks but cannot write more than that limit. We assume the user will never write a file past the max allotted size.

**Class Specifications**

**Directory Class**

**Inode Class**

**FileTableEntry Class**

**FileTable Class**

**SuperBlock Class**

**Kernel Class**

**SysLib Class**

**TCB Class**