Final Report
Elijah Shaw & Braxton Goss
CSS 430 - Dr. Fukuda
6/9/2022

1. Any assumptions and limitations

Assumptions:
We assumed that the other code given in ThreadOS files was accurate and functional, which appears to be the case based on our program's ability to pass all tests. We also assumed that when functions were shown as having different return types in the assignment specification files, it was okay to pick from one of the two and code other functions relying on them to match that (this really only happened with functions that could have returned integers or booleans).

NOTE TO GRADER: We tested our program by placing our implemented files in a folder with the currently available class files from the ThreadOS folder on the linux machines (using Scheduler_fil and Kernel_org as the assignment requested). See our output screenshot for details on how we compiled and ran the files.

Limitations:
We worked with the ThreadOS files given to us at this time, meaning we didn't change the implementations of Inode or TCB. Our code had to work within the constraints of all the current ThreadOS files, as changing any of them might mean our implementation didn't work as intended by the assignment specification.

2. Detailed internal designs of your super block, file table, and file system.

Superblock:
We followed the given implementation from the provided PDF and filled in the rest from the comments provided. The superblock can format and sync itself. It allows for other classes to call for a free block to use for their own functions, or to return a block no longer being used back to the superblock.

File Table:
FileTable is used by the FileSystem class to allow threads to open files by accessing the entry's place in the table. It is also used to allocate and free space in the table via falloc() and ffree(). It also contains a method that checks whether the table is empty or not which is then used by the file system to conduct a format() call on all the blocks.

File System:
This class manages a superblock, file table, and directory for a file system. It can open and close a specified file, as well as delete it (which calls close). It allows for the reading and writing of files through the use of buffers with no specified size, allowing a user or the system to adapt to the amount it needs to write. The user can also change where the current file's pointer is through the use of seek by providing an

offset and whence value that inform it of how to update the pointer. FileSystem does boundary checking in read, write, and seek, in order to ensure nothing goes out of bounds or gets accidentally overwritten/deleted. A file's size can also be determined through "fsize" by passing it a FileTableEntry. FileSystem can format and sync itself as well, formatting superblock in doing so.

3. Consideration on performance estimation, current functionality, and possible extensions of your file system.

Performance Estimation:
Runs slowly when creating over 40 files in test #17. Otherwise, it appears to run through the tests at a moderate pace without any obvious issues.

Current Functionality:
Our system is currently fully functional. It passes all 18 tests without any errors. One warning about deprecation is given when compiling, but it's from ThreadOS code that was provided to us, not any of the code we implemented ourselves.

Possible Extensions:
The first way we could extend this project would be to extend it beyond just having a root directory. We could also extend it by adding new functionality that would allow for additional file management tools that are present in current file systems such as the ones on Windows/Mac/Ubuntu like: rename(), copyFile(), zip(), createShortcut(), viewProperties(), viewFileLocation(), and many more. We also would want to test the file operations individually using timed testing (by fetching system time on start and end and subtracting them) and compare these to future iterations of the file system.