John Hodson
PoCSD HW2 Design
09/16/2019

I implemented the functionality of the copy, write, and read functions in this assignment. Read and write functionality was very similar to the functionality that I implemented in HW1, so I used HW1 as a skeleton for this assignment. I wrote common functionality that is shared between the write and the read functions, namely a calculate_blk_range function that when given the offset and the length of a data block to perform operations on, it calculates the block index range (inclusive) that we should be reading and writing to.

Write (I think my thoughts are more easily expressed in pseudocode)

- First, we validate the inode type and ensure that it is a file.
- Then, we calculate the block range that we are operating on using the offset and the length of the data.
- If the first block index that we just calculated is out of the range of our block numbers, we raise an exception because the offset is larger than the amount of data we have available for this inode.
- Then we loop over the range of blocks that need to be written to
    - Read the old data, as we are going to be updating this data.
    - Iterate over the old data, concatenating onto a new_data string as we go. Choose a character from the new data if it is within the range of byte numbers that we want to overwrite and taking the old data if it is not.
    - Once we have finished creating the new data to write, update the data block with the new data.
- Update the time modified and time accessed in the inode and return the inode.

Read is very similar to the write functionality, but simpler. The same calculations of block indexes and checks for offset out of bounds are performed. We are still going to loop over all of the data blocks that our data addresses are in, but this time we are going to concatenate the data of interest onto a string so that we can return it at the end of the function. Before we return the inode and the data that has been read, update the time_accessed attribute on the inode.

Copy is slightly different than the previous two functions and was the easiest to implement. First, validate that we are performing the operation on a file inode. Create a new inode that is a file. We must copy all of the old data into this new inode. That means, we must iterate over each of the blk_numbers present in the inode that we want to copy. If the blk_number that we are currently on in the inode we want to copy is equal to -1, then do nothing. If it is equal to an actual blk_number, then we need to make a call to the interface to allocate a new block number. Then, we read the old inode’s blk number and write this data to the new inode’s blk_number. This loop ensures all of the data that is referenced by the old inode has a corresponding copy referenced by the new inode. Additionally, copy all of the other attributes in the old inode to the appropriate attribute in the new inode (name, etc.). Finally, return the new inode.

To test the functionality of InodeLayer.py, I created a new python file called InodeLayerTest.py that enumerated several test cases (~6). Each of these tested an aspect of the code where I felt something could go wrong. I have included this platform in my submission.