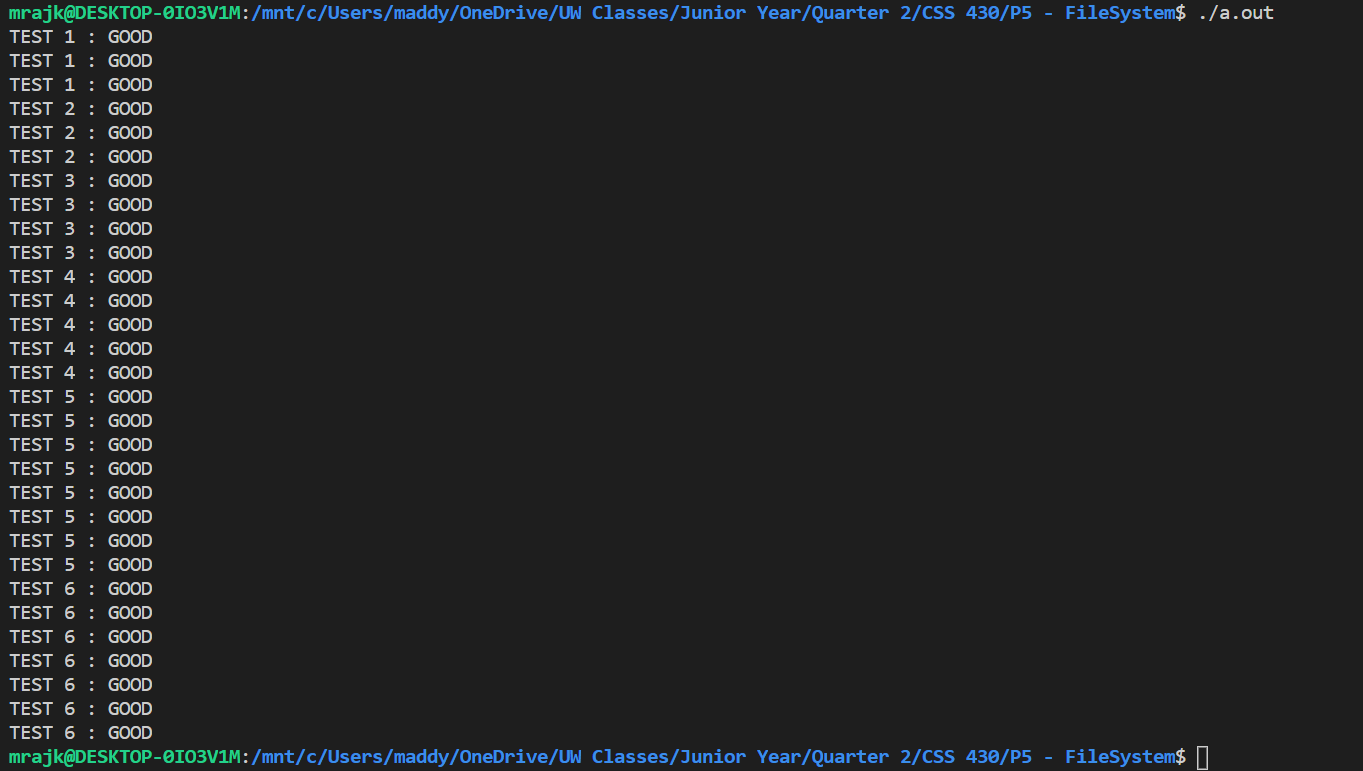
P5: FileSystem Report

Here is a screenshot of our results which shows that all the tests are passed:



For extensive testing of our program to ensure all edge cases are accounted for, both of us made another test case method where did multiple checks on the program. This way, we were able to ensure there are no errors when opening and closing a file, writing to a new disk, and ensuring that we are not corrupting the BFS disk.

We made sure the test cases worked by adding a new BFS disk in our directory and running all the test cases at the same time. If we got the response of “GOOD” then we had affirmation that the test cases have passed.

Design Document

A filesystem starts out with a raw disk that has millions of blocks and can read/write to any blocks. The Filesystem has the following abstractions:

* **Files:** this is a container of bytes that can grow or shrink. In our program, we were responsible for knowing where they are located on the disk, their specific size/protection/owner. The Read/Write can be for any arbitrary number of bytes and for a variable amount of length.
  + A file takes up 3 blocks, numbered 0,1,2, where each one is 512 bytes long. These blocks are located at physical Disk Block Numbers (DBN) 123, 124, and 127.
* **Directories:** they help map a path and a filename to the desired file. They are able to grow and shrink and have the ability to be renamed and deleted.
* **Freespace:** this is the space where we can see which blocks are free or unallocated to any of the files.

The BFS is a layered file system with 3 different layers from top to bottom. Here we will explain the 3 layers which include all the methods and the logic we implemented in the read and write methods:

1. **User-level filesystem:** includes several fs functions as follows:

This file contains all the fs functions. fsOpen opens the file with the appropriate parameter name. This returns a filedescriptor or fd to perform operations on file. If it fails, then return EFNF (file not found). For the fsRead, in our program, we first decided to make variables for the cursor at the start, the read size of the cursor and number of bytes of data from cursor opened in the file, an Inode with a direct array to map between FBN and DBN, and an indirect field that holds the DBN. We made a new array called bioBuf to store the bytes read. Then, we had a loop to go through all the bytes and inside it we made several checks like if the block is in indirect then break, else if the buffer is starting at 0 then keep reading in 0s. Then, we have another loop going through all the bytes per block where we make several checks like if we read 0 then break or if we reach end of file (eof) then break. After that, we update the cursor value, increment the block counter variable, and set the eof variable to 0. Finally, check if the sum we tracked is equal to the number of bytes read or if we are done then break. Now, we make another if statement to see if we are not done, then we need to follow a closely similar process as above so the logic will be the same here. On success, return the number of bytes read but on failure, abort the program.

For the fsWrite in our program, we made variables similar to fsRead but specific for this function. First we had an if/else statement to check if we are inside, outside, or overlap. Then, we created a buffer to check how many bytes left. Then, we checked if the inside and overlap are both true then only read first block, or final block with moving left over data respectively. If we have overlap then extend the file size appropriately. Then, go through all the blocks and biowrite followed by bfs conversion from FBN to DBN. Then set the cursor size. If we are outside, then very similar logic as above but we also write the Inode and dereference file with inum in the open file table. On success, return the number of bytes read but on failure, abort the program. There is fsSeekwhich adjusts the cursor to offset. SEEK\_SET for 0,1,2 decide where the offset starts whether at the start, current, or end of the file. On success, return 0 else on failure, abort the program. Finally, fsClosewill close the file currently open on fildescriptor fd. On success return 0 else abort the program.

1. **Functional internal to BFS:** includes internal Bothell file system functions:

This file contains all of the constants used throughout the program as well as the structures for the superblock, the directory block, the I node, as well as the open file table. There are Initialization methods to initialize the directory, the free block list, the Inodes, the superblock, as well as the open file table. There's also a method to reference and a method to dereference an entry in the open file table. There's also an extended method which will extend the file out to the file block number specified in the parameter. This method calls the find free block and allocate block functions in order to allocate more space for the file. There's also a method to create a file by taking in a file name And methods to convert from file block number to disk block number, file descriptor to Inode number, And inode number to file descriptor. There is also a method to get an entry out of the open file table with the inode number. There are methods to get instead the size of the file. There's a method to read a file block number into a buffer And a message to read and write to an Inode. The last 2 methods will get the cursor in the file descriptor or set the cursor using the inode number.

1. **Lowest level block IO functions:** includes lowest level block IO functions:

This file has bioreadwhich reads the block number dbn from the disk into the memory array buf and returns 0 else aborts if failed. There is also biowritewhich writes the contents of the memory array buf into block number dbn on disk.