MP7: Simple File System

Joshua Clapp

UIN: 930004089

CSCE410: Operating System

Assigned Tasks

Main:

Completed.

Bonus Option 1: Disk Design

Completed

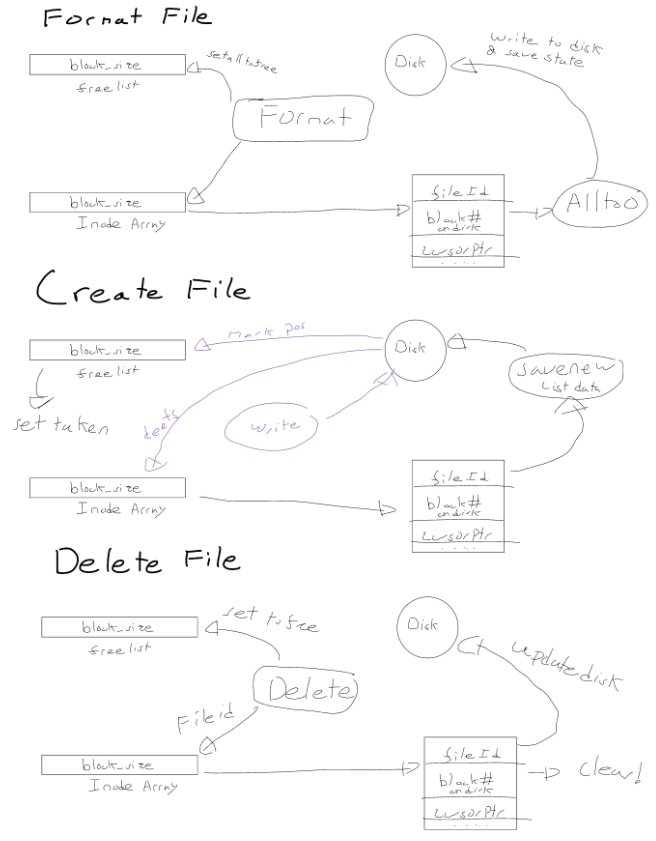
Bonus Option 2: Disk Design Implementation

Not attempted

System Design

**Main:**

The main system is to write sequentially and allocate free blocks on disk

****

**Bonus Option 1: Disk Design**

To expand the disk to hold files up to 64KB, I would need to allocate more blocks per write a file as each file block has 512 bytes on the simple disk drive. This would require a function that finds free blocks. I would need to then account for the cursor in the file write as changing blocks would require me to hold a cursor pointer to the buffer position. Then, I need to mark multiple allocated blocks related to the same inode and the order to read in the proper order, a queue would work well here. When deleting, instead of deleting one 512 block for a file, I would need to delete multiple blocks up to 64KB, order would not matter here, this would require the same data structure that links multiple blocks per inode.

Code Description

**Main:**

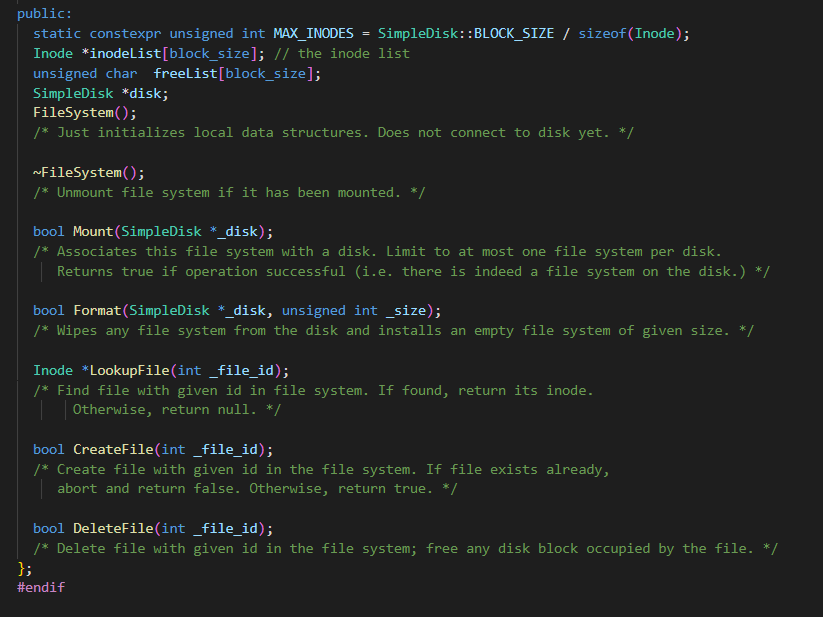
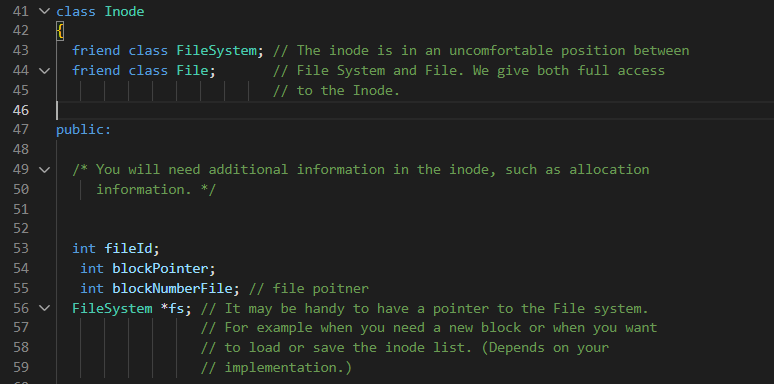
I changed File\_System.C, File\_System.H, File.C, File.H, and Kernel.C.

**To run this logic**

Make and run it

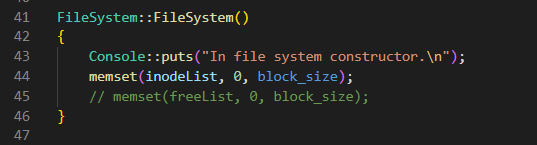
File\_System.H

The basic implementation is inodes containing file Id, the cursor pointer on the disk, and the respective block number that the disk contains the written file on.



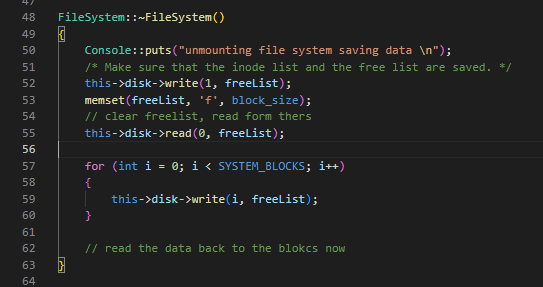
File\_System.C:FileSystem()

This initializes our inodeList and sets all data to 0



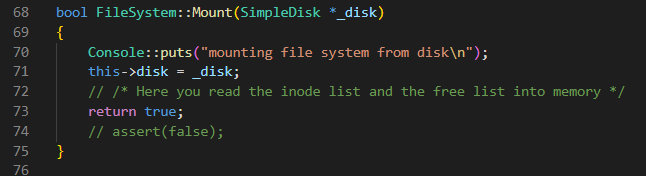
File\_System.C: ~FileSystem();

Write the free list data to the disk, set the memory and then copy all the inode data back into the file system.



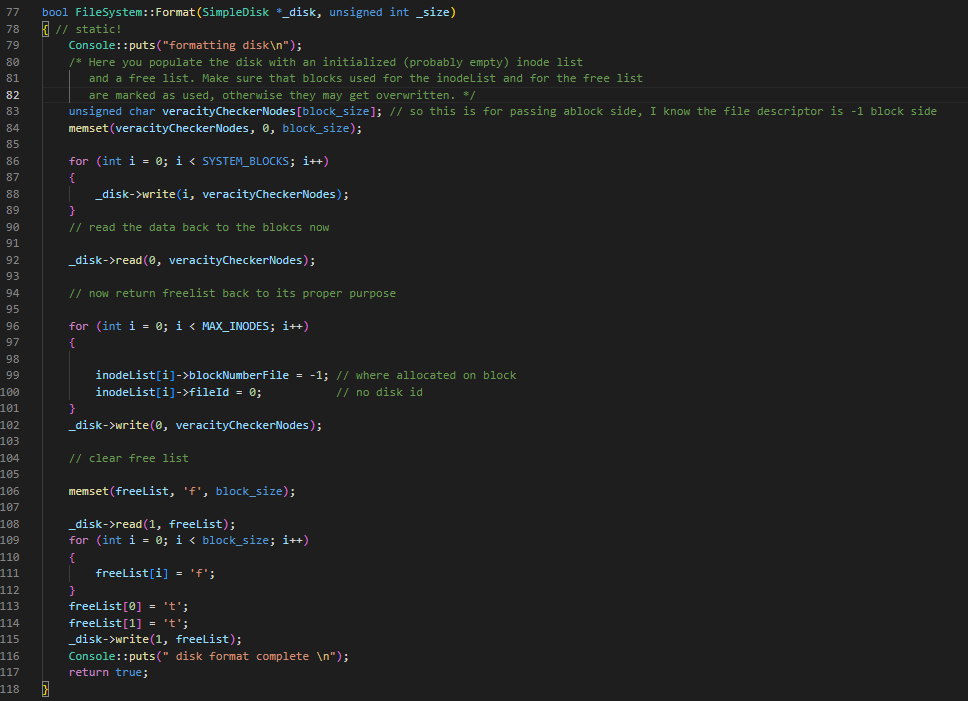
File\_System.C: Mount(SimpleDisk \*\_disk);

Set the given disk into the Inode drive



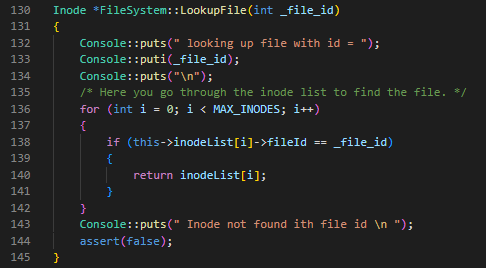
File\_System.C: Format(SimpleDisk \*\_disk, unsigned int \_size);

This uses the free list as a placeholder and writes an empty inode array, then reads it back into the inode list, and writes into the block zero of the disk. It then sets up the inode array to the empty parameters. I then clear out the freelist and then read from the freelist block into the freelist array, and mark the inode, and freelist block as taken and lastly writing this data back to the disk.



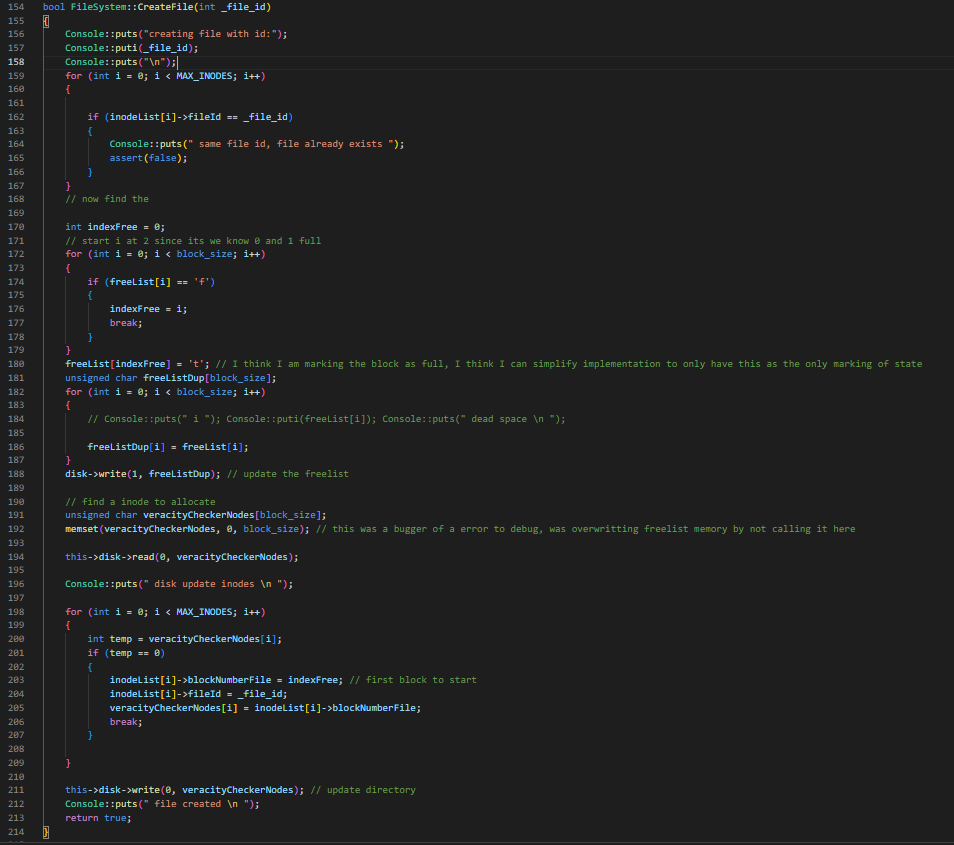
File\_System.C: \*LookupFile(int \_file\_id);

This runs through the inode array, finds a file id that matches and returns it.



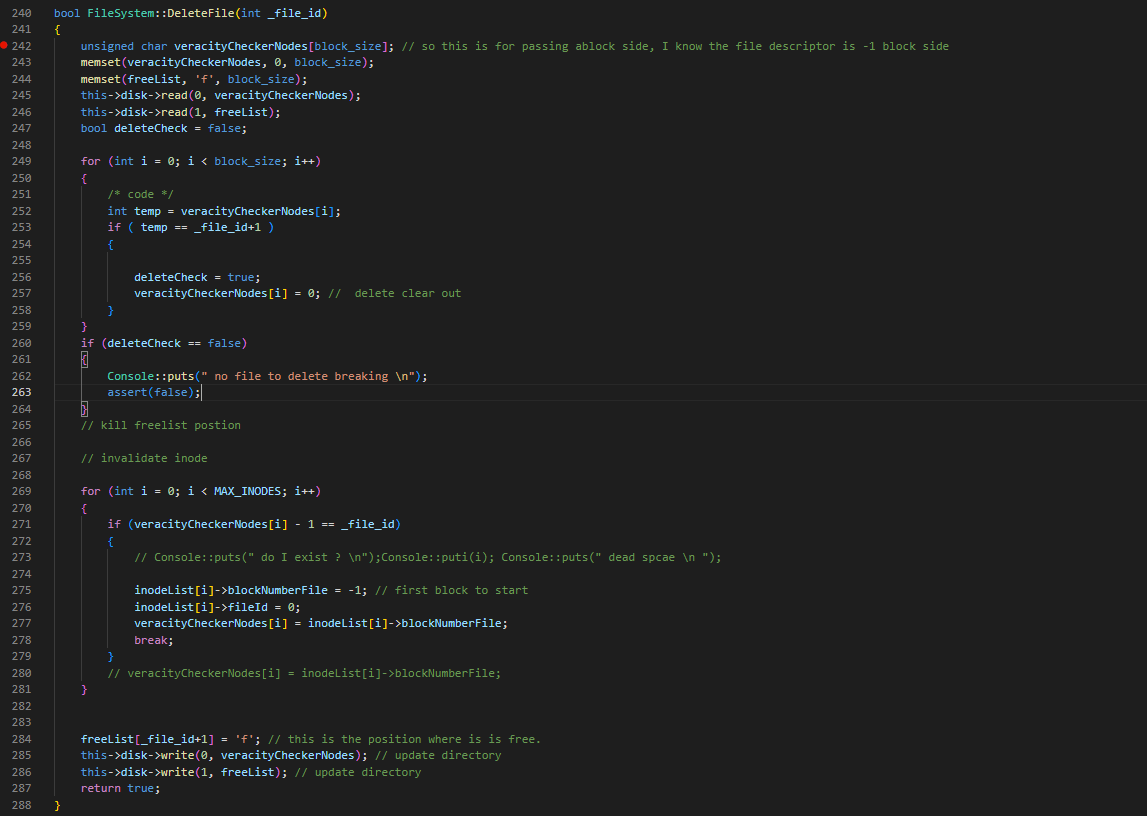
File\_System.C: CreateFile(int \_file\_id);

First error check to see if the file id does not exist before. Then check the free list and find the first free block and break sequentially. Mark the freelist as taken, and then I make a copy of the freelist to call from the disk block zero, I take a deep copy of the freelist and read from the disk. Using this data from the disk I find the first unallocated inode and assign it the proper block number, and file id. Then I write the data to the disk and return.



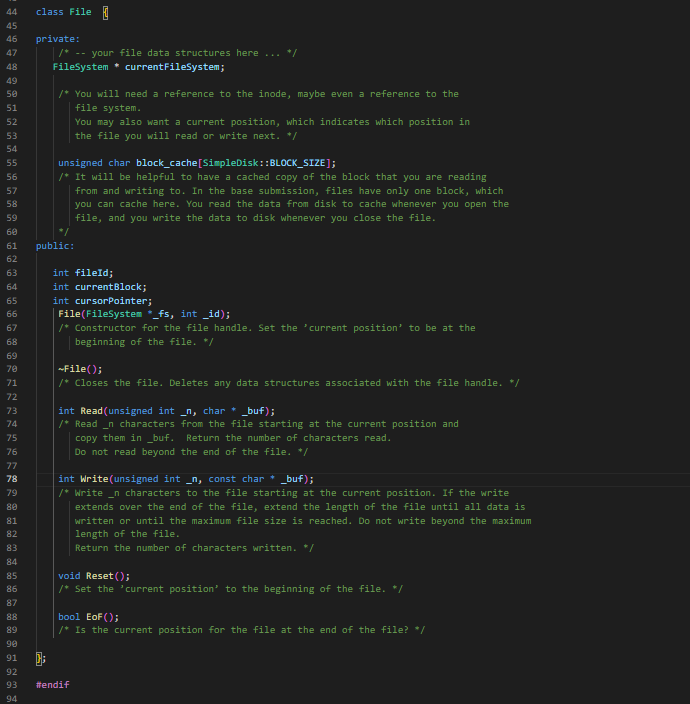
File\_System.C: DeleteFile(int \_file\_id);

Make sure the file id exists by taking a copy of the inode disk data from block zero, and error checking it for the file id using the fact that with our sequential allocation, the block number is always file id pulse one. Then go through the Inode array and at the file id clear the inode out, reset the freelist and write the data back to the disk for both freelist and inode.



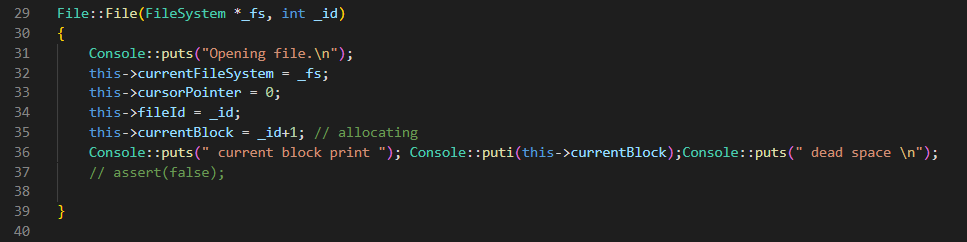
File.H

I keep track of file id, current block on disk from the inode, and the cursor of the buffer reading to or writing frome.



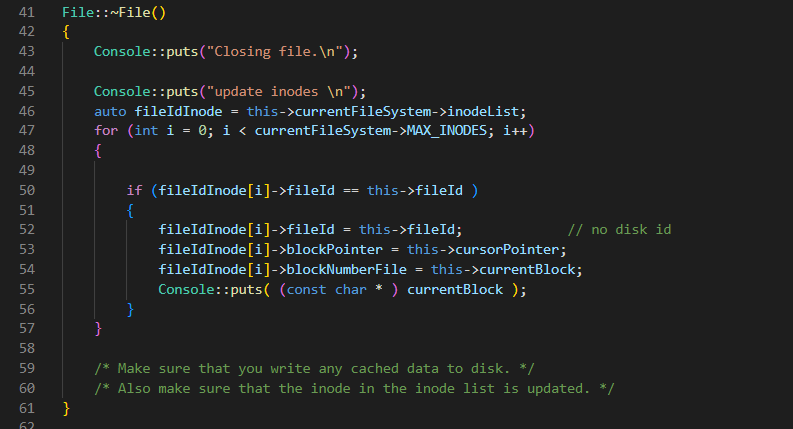
File.C: File(FileSystem \*\_fs, int \_id);

Save vital information like current file system, set the cursor pointer marking the buffer to zero, save the file id, and use the knowledge that a current block allocation is one more than the file id.



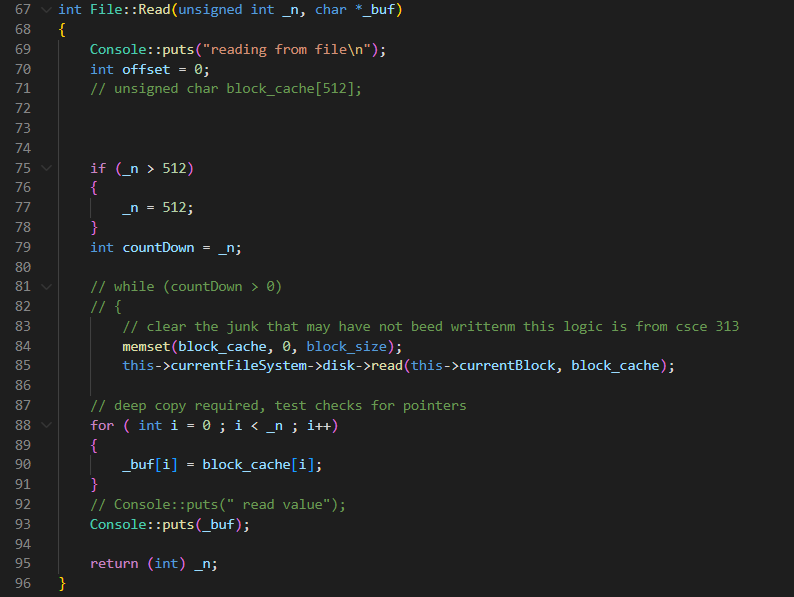
File.C: ~File();

Search through the inodes and match the file id saving the fileId, cursor pointer, and the current block.



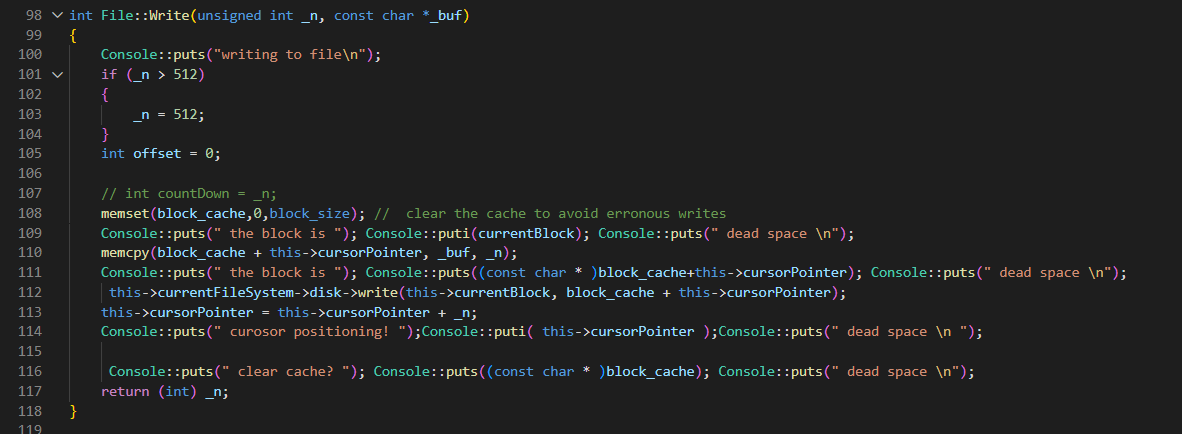
File.C: Read(unsigned int \_n, char \* \_buf);

Error check the \_n data make sure it is 1 block, and then clear the memory in block\_cache, reading from the disk into \_buf.

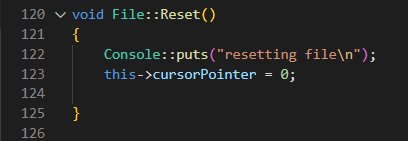


File.C: Write(unsigned int \_n, const char \* \_buf);

Error check the \_n block sign to make sure it is one block sign, clear the memory in the data, copy the data into block\_cache and add the pointer from \_buf, write the data to disk and adjust the pointer.

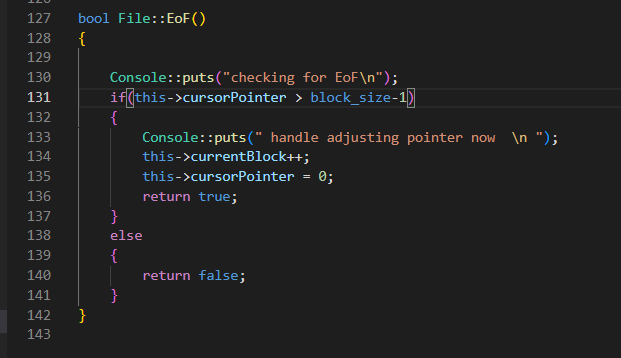
File.C: Reset();

Reset the cursorPointer to zero



File.C: EoF();

Check if the pointer is at the end of the file block and allocate another block, resetting pointer. This would be used in the design of the MP2 bonus.



Testing

I added no additional tests and the coverage is quite small, for the actual block size testing, there are no files that allocate to multiple blocks or multi directory files. I relied on the test cases implemented which ensures that my system can dynamically allocate to the disk, delete and reallocate without issue. I am ignoring eviction policies of the data and other important file system operations.

**Main Task:** Device Driver Delete, Create, and Allocate files.   
THIS RUNS FOREVER, I ADDED A STOP CONDITION OF 20 ITERATIONS SO 40 FILES ARE CREATED AND DELETED.  
