ThreadOS Final Project Report

CSS430 Winter 2019: 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

Kernel Class

Kept the standard Kernel class with only slight modifications to the separate cases relating to the various file system interface calls. These include OPEN, CLOSE, SIZE, SEEK, FORMAT, DELETE, READ, and WRITE. These various cases call their corresponding FileSystem.class methods and return the values returned by those method calls to indicate failure or success. These Kernel calls are called by the SysLib interface.

SysLib Class

SysLib implemented methods include public static int… format(), open(), read(), write(), seek(), close(), delete(), and fsize(). These calls take in the appropriate parameters representing the parameters to the various system calls and return in the format… 'return Kernel.interrupt(Kernel.INTERRUPT_SOFTWARE, Kernel.<code>, params)'. An example call to the fsize(int fd) call would be:
'return Kernel.interrupt(Kernel.INTERRUPT_SOFTWARE, Kernel.SIZE, fd, null);'
representing the type of interrupt, the specific interrupt code, and the file descriptor to run the method on. The other SysLib interface calls relating to the file system map similarly to this and are linked to the FileSystem.class' corresponding methods.

TCB Class

Modifications to the TCB class include a file table entry array which represents the various open files in our file system. It's mapped to 32 total elements and have methods associated with this object such as public synchronized int getFd(FileTableEntry entry) {… } which scans through the 32-element array, comparing each index to the compared to parameter value and returns the index of it, -1 otherwise if not found. This index represents their file descriptor index so indexes 0, 1, and 2 are reserved to simulate standard in, standard out, and standard error.

public synchronized FileTableEntry returnFd (int fd) {… } will free up the FileTableEntry index but save it to a variable first. It'll then free up the old slot and return the old value, null if the index is invalid.

public synchronized FileTableEntry getFtEnt(int fd) {… } just returns the file table entry object at the given index represented by the file descriptor number 'int fd'.

These methods have been synchronized since various file system calls are being made that may cause race conditions such as attempting to delete a given entry as we're attempting to access a given object.

FileTableEntry Class

A given class to us for the project but wanted to talk about the structure so the later sections have more clarity.

int seekPtr represents the current file seek pointer of a given file. This is where the file will be reading and writing at in the file.

Inode inode is the object representing the file's meta data which allows us to be able to find it on disk, know its details, and read/write from/to the file because of this.

short iNumber is the inode's number that the FileTableEntry object represents.

int count is how many threads are pointing to this object. This is instantiated and incremented through the 'OPEN' file system call.

string mode is the current mode the file is opened in. This includes "r" (read), "w" (write), "w+" (write+ or also known as write+read), and "a" (append). Having a "a" mode will set the file seek pointer to the end of the file (since we're adding onto the end of the file).

Directory Class

int maxChars represents the maximum length of a given file.

int[] fsize is an array representing each file's name's length.

char fnames[][] is a two dimensional char array which'll coordinate with the fsize array to store the file's names as well as their lengths.

A directory in a given file system is, in itself, a file. Knowing this fact, we map the first file in our directory as the directory itself, assign it a file size of 1 to represent "/" root directory. It'll initialize the fsize array accordingly to the number of maximum inodes allowed in the system (which is equal to the maximum amount of files in the system).

bytes2directory(byte data[]) meant to populate the Directory object from disk. It assumes data[] contains valid directory data. It's the counterpart to directory2bytes() which'll be covered next. This essentially only saves the meta data of the directory which happens to be all the file names and their lengths associated with it. It'll iterate through the given fsize array and do byte to int conversions using SysLib.bytes2int(buffer, offset); and setting each fsize index accordingly. The offset is in terms of 4 since an integer (fsize is an integer array) is 4 bytes in Java. By doing this, we iterate correctly over the disk, getting the appropriate data. For the names, it's trickier. Since a given name has a maximum length of 30 characters, that's 60 bytes total per name since a char value is 2 bytes in Java. It'll repeat this same process except use the String.getChars() method to copy a given string to map into the char[][] 2-D array.

directory2bytes() will do the opposite of bytes2directory(). It'll write the fsize int values to a byte buffer in increments of 4 (an integer is 4 bytes) until all fsizes have been written. Writing the given file names to the buffer took more research. First we get the string representation by invoking the String constructor String(char[], startIndex, length)

to copy the character array from 0 to length-long to get the string representation. There isn't a String2bytes method which could've been an add-on into the SysLib methods but System.arraycopy mimicked the functionality perfectly. System.arraycopy(fromBuffer, 0, endBuffer, offset, lengthToCopy) is what allowed us to copy data from our temporary buffer from index 0 to our directory buffer at the given offset to the lengthToCopy. The offset is then incremented by 60 to represent the amount of bytes a file name takes up on disk. It'll then return this byte array representing the directory.

ialloc(String filename) allows us to find an empty slot in our directory to allocate a new inode essentially. It'll go through the fsize array and find a index whose value is 0 which indicates an empty slot for a given file. It'll then map accordingly to the fname[] char array and populate it with the filename. It'll return the index number to represent the inode number that'll populate it. It'll return -1 if there are no slots left for the given file.

ifree(short iNumber) will free up the given file slot. One of the design choices we went with is to free up element in the fsize array. Since the fsize array is how we determine if a file exists or not in ialloc, we don't need to clear up the fnames[] entry since it's going to take up 60 bytes regardless of if a file exists in that slot or not. Because of the space constraints, we didn't find a need to clear it up. This method'll set the given index to 0 and return true if successful and false otherwise.

namei(String filename) seems to be the most practical one in communicating between the user's experience and file system itself. It'll iterate through the fsize array and check if there's a file with the same length. If so, it'll check the fnames equivalent and check if the given file exists within the directory. If so, it'll return the inode number. To us, this seemed the most practical. It allows us to map the human-readable form of the file's identifier to the filesystem's representation of the given file. Since this is a single directory file-system, this is even more important.

Overall, based on the specifications the Directory is a helper class that helps the FileSystem and FileTable classes be able to map the physical files to their own representations of them. To be able to identify the files correctly, allocate them, and deallocate them accordingly. Because of the specifications mentioned in the video as well as the homework documentation, this is the direction we went with this class.

Inode Class

static int iNodeSize = 32; represents how many bytes a given inode takes up on disk. It has a int length (4 bytes), short count (2 bytes), short flag (2 bytes), short direct[11] (22 bytes), short indirect (2 bytes) which sum up to 32 bytes per Inode object.

static int directSize = 11; references how many direct data block pointers we have per Inode object.

static int iNodePerBlock = 16; references how many inodes can fit in a block of data. A block has 512 bytes, an inode is 32 bytes so 512 bytes / 32 bytes = 16 inodes total.

static int OK = 0, BLOCK_ERROR = -1, MISSING_ERROR = -2, REGISTER_ERROR = -3 represents the different states a given inode request can be in. If everything is fine a 'OK' is returned to indicate so. A 'BLOCK_ERROR' is returned when the given target block number is out of bounds of the data. A 'MISSING_ERROR' is given when the given target block we want to register is missing

or invalid. Lastly, a ‘REGISTER_ERROR’ is given when an indirect pointer hasn’t been assigned. We can’t allocate a block and have it floating around without it being assigned to a given inode’s indirect pointer.

int length represents how many bytes the file is.

short counter represents how many file table entries opened this file.

short flag represents the state of the file (unused, used, reading, writing, etc.)

short direct[] represents the 11 direct pointers we use to map data blocks to the file.

short indirect represents the pointer to an index block which then maps to 256 more blocks.

public Inode() initializes all the variables to default values (length = 0, count = 0, flag = 1 to indicate it’s being used, initializes all 11 direct pointers and the indirect pointer to -1 to signify that they’re not used).

public Inode(short iNumber) by receiving a iNumber, it’ll map to the block the inode is located in, the offset within the block and retrieve the inode’s instance variables. It obtains the block number by dividing the iNumber by iNodePerBlock then adding 1 to account for the SuperBlock. Getting this block number, it reads in that block from disk. From there, we get the offset size by taking the iNumber % iNodePerBlock then multiplying by the iNodeSize to get the byte index within that block. From there, we get the length, count, and flag doing simple offsets along with bytes2int and bytes2short calls. We then iterate through 11 more times to get the direct pointer values and once more for the indirect pointer values. Meant as means to populate the inodes’ data values when we’re reading back from disk.

toDisk (short iNumber) is our way and as very heavily hinted way to write the given inode to disk. By getting the iNumber, we can find the correct place to write the inode back into disk. Taking the iNumber and dividing by iNodePerBlock and + 1, we get the block the inode belongs in. We find the offset doing iNumber % iNodePerBlock and multiplying by iNodeSize to get the offset within the block. From there, we write the meta data of the inode much like we did in the public Inode(short iNumber) constructor but in reverse (utilizing the int2bytes and short2bytes methods provided in the SysLib class). This method was made synchronized since we’re modifying the disk contents.

findTargetBlock(int numBytes) is finding the block the current file seek pointer is pointing to. We find the block number (mapped accordingly to the inode object, not block numbers mapped to the disk → block numbers 0 to 10 are the direct data mapped blocks and so on not blocks 0 to 10 on disk). If the given block number is within that range, we can return the disk block number by indexing into the direct pointer array. Else, we check if the indirect pointer is pointing to an index block already. If so, we can find the block by reading in the current index block and finding the offset. Each 2 bytes in this block represents a short so we can map into it by multiplying the indexBlockLoc variable by 2 to represent the 2 byte-offsets of shorts and return that number.

registerTargetBlock(int offset, shortIndexBlockNum) is the method that checks and maps a block to a pointer. We can find the correct block number to map to by dividing the offset by Disk.blockSize. If it maps to one of the direct blocks and they’re populated, it’s a BLOCK_ERROR which represents a index error in this case. Else, we’ll populate that pointer with that block. This essentially updates the block information If and only if

it’s not currently in use. With the indirect pointer, we’ll read in the data block it points to, find the target block pointer in the index block, update its pointer, and write the information back to the disk using SysLib.rawwrite.

FindIndexBlock() just returns the indirect pointer’s value. May be -1 (not used) or a positive value (used).

registerIndexBlock(short indexBlockNum) method for mapping a block and formatting it to be 256 pointers to other blocks. It’ll check if the indirect pointer is in use already by comparing it with -1. If so, the method call was invalid since the indirect pointer is already in use. Otherwise, we’ll iterate through the direct pointers and check they’re already in use. If none of those failed then we map the indirect pointer value to indexBlockNum. We create a byte buffer of Disk.blockSize (512 bytes). Then we write do a series of SysLib.short2bytes((short) -1, data, forLoopVar) 256 times total. The -1 represents a pointer to another block but not in use, ‘data’ is the byte buffer and forLoopVar is the offset within the block which is incremented by 2 to represent the 2 bytes a short takes up. At the end of this, it’ll rawwrite this block back to disk.

unregisterIndexBlock() resets the indirect pointer. It’ll first check if the indirect pointer is in use at all by checking it’s a non-negative. Second, it’ll save the data into a byte array of size Disk.blockSize. Set indirect to -1 and return the byte array representing the data (pointers) on disk.

FileTable Class

final static int UNUSED = 0, USED = 1, READ = 2, WRITE = 3 are used to safe guard against each other. These mappings will be used to prevent malicious attempts at race conditions.

Vector table is a vector of FileTableEntry objects so it can grow and shrink dynamically as file table entries are added and removed from the file table.

Directory dir acts as a communicator between the FileTable and the various Inodes and FileTableEntries. As described previously.

Falloc(String filename, String mode) stands for file allocation. When a file is first opened, it needs a name and mode associated with it. This is one place where race conditions/dead locks may occur so we need inode mappings described earlier. Method is synchronized to protect against race conditions along with wait and notify calls embedded. We check if the file has been allocated before and what mode it may be in. For example, if the file is already opened in write mode and another file tries to open it in read, the process will call wait until the first process is done writing. This applies vice versa (attempting to open in write while another is already reading). It’ll increment the inode count by one and create another FileTableEntry with the updated value. The vector of FileTableEntry objects will add this and return the newly created FileTableEntry object.

ffree(FileTableEntry e) Will remove the given entry from the vector if found inside. It’ll set the inode flags to UNUSED to indicate they’re being closed essentially. It’ll decrement the inode count by 1, write the inode to disk and call notify(). It’ll return true if everything ran smoothy or false if there wasn’t the specified FileTableEntry in the first place.

fempty() will just return table.isEmpty() which does what you would anticipate it to do. Will return true if there are no FileTableEntry objects and false otherwise.

SuperBlock Class

final int DEFAULT_INODE_BLOCKS = 64; is the default number of inodes created for a disk of 1000 blocks.

Final int totalBlocksLoc = 0, totalInodeLoc = 4, freeListLoc = 8 represent the offset within a byte these values will be located.

totalBlocks is number of blocks in the disk.

inodeBlocks is the total number of inodes.

freeList points to the first free block in the disk.

FileSystem(int diskBlocks) will initially read in the superblock to check if it was formatted before. It'll read in from the superBlock at offsets 0, 4, and 8 the totalBlocks, inodeblocks, and freeList. If the totalBlocks read from doesn't equal diskBlocks, inodeblocks isn't > 0, and freeList isn't 1 above the last block containing inodes, then our disk needs to be formatted. Else, it's fine. To format, the constructor sets totalBlocks to diskSize and calls format().

format(int numInodes) sets inodeBlocks to numInodes first. totalBlocks was already set in the constructor beforehand. It'll iterate through from 0 to numInodes, create the inode, and write to disk by calling inode.toDisk((short) i); Depending on how many inodes we have made, we need to offset the freeList accordingly. It'll get the block number by multiplying the number of inodes (numInodes) by 32 (32 bytes per inode) then dividing by Disk.blockSize to give us the block number they map to. We add at least 1 because of the superblock being the 0^{th} block. Then we add another if the inode number wasn't cleanly divisible by 16 (E.g numInodes % 16 != 0) which means one block is only partially filled with inodes but can't necessarily be used as a free block. Afterwards, we iterate through the entire disk via block-size chunks of 512 bytes in array form. Then we do a series of int2bytes(i + 1, data, 0); to link the freelist of one block to the one ahead of it. The last block is set to -1 to indicate it doesn't point to a freeblock and that it's essentially the end of the disk.

sync() essentially updates the superblock with the most recent data. We'll create a temporary byte array of size Disk.blockSize, write the totalBlocks, inodeblocks, and freeList using int2bytes with the appropriate offsets mentioned. Then write the byte array to the 0^{th} block, essentially replacing the superblock with a newer updated one.

getFreeBlock() will check if the freeList variable is -1 first (End of disk mentioned beforehand). If not, it'll save the freeList variable to be returned. Read in the block that the current freeList is pointing to, set the freeList to this block's freeList using bytes2int to get the new freeList, and returning it.

returnBlock(int blockNum) will validate the blockNum is valid (non-negative). It'll create a byte array of Disk.blockSize, write our current freeList to its freeList variable's value, write that array to disk then set our current freeList to point to that block. Then we'll return true indicating it was successful so the next getFreeBlock() call should return this block we just added to the front.

FileSystem Class

SuperBlock superblock holds data pertaining to the Disk as a whole

Directory directory holds data relating to on the file's human-readable mapping

FileTable filetable holds data based on how the file system will view the files.

FileSystem(int diskBlocks) will use this number of diskBlocks (1000 in our case) to create a new superblock, use the superblock to make a directory with the given inodeBlocks number, then use the directory to create a FileTable object. It'll open the root directory in read mode, get the number of files in the directory, and if it's shown that the directory already had files pre-existing, then we'll repopulate the directory's file names using directory.bytes2directory(numberOfFiles). Then we'll close the FileTableEntry relating to the root directory.

sync() is the higher level call of sync. We'll first save the directory's current data by opening it up in write mode, save it to a byte array by using directory.directory2bytes(), write the directory to the given FileTableEntry (should be the first one always), close the FileTableEntry object, then call superblock.sync() to write all remaining data to the disk.

format(int files) will first check if there's any files open by calling filetable.fempty(). If not, we'll format the superblock once more with superblock.format(files). Then using this newly formatted superblock, we'll recreate the directory and filetable objects and return true indicating it was successful.

open(String filename, String mode) we use filetable.falloc to create a new entry in the given mode. If the file is in write mode, we need to clear up existing blocks by calling deallocAllBlocks(FileTableEntry ftEnt). We return the newly created FileTableEntry object.

close(FileTableEntry ftEnt) will decrement the count. If the count is 0, we'll free it from the file table.

fsize(FileTableEntry ftEnt) returns ftEnt.inode.length which represents how many times this given file has been called open on.

read(FileTableEntry ftEnt, byte[] buffer) will check if the modes are "w" or "a" to ensure we can't write during a read (unless it's "w"). Else, we'll keep track of the total amount of bytes read with a totalRead variable (integer). Total amount to read is kept track by int toRead = buffer.length. We have a while loop that checks if the seekPtr is within the range of the file and that the amount toRead is > 0, otherwise, we read all that we could. We'll find the target block of the given FileTableEntry object by calling ftEnt.inode.findTargetBlock(ftEnt.seekPtr) which maps to the block we need to read in. We'll do a rawread into a byte array and check the amount we can read by getting the smallest value between Disk.blockSize – offset, the amount left to read, and the total amount of the file left from the seek pointer. This ensures we don't go past the file size boundary.

We then do a System.arraycopy to the buffer and advance the seek pointer by readAmount, decrement the toRead amount, and increment the totalRead amount by the same increment. At the end we return totalRead.

write(FileTableEntry ftEnt, byte[] buffer) follows the read semantics in that buffer.length is the total amount we do operations on the disk with. We keep track of the total amount we've written with amountWritten = 0 initially. Int amountLeft = buffer.length initially. Then we loop while (amountLeft > 0). We find the target block like we did in the read method. If the given block number returned is -1, it means we

need another block to start writing in. We can find this by calling superblock.getFreeBlock then determining if this is a direct or indirect block. If direct, we can start writing right away. If indirect, we have to first register the index block by calling ftEnt.inode.registerIndexBlock((short) freeBlock) and registering it as a target block with ftEnt.inode.registerTargetBlock(ftEnt.seekPtr, (short) freeBlock)) then setting the block number to read from accordingly. After we determine the block we're writing to, we'll read in the current data at that block since we may overwrite pre-existing information. We need to keep track of two things: 1. How many bytes are left within the given block, and 2. how many bytes are left to write overall. We can find out 1. by doing seekPtr % Disk.blockSize and subtracting that value from Disk.blockSize. We then find out the minimum between the number of bytes left in the block vs number of bytes to write overall and take the minimum of the two (this ensures we don't write more than we intend to). We do another System.arraycopy to populate our byte array buffer then rawwrite this buffer back to the disk using the block number we've obtained earlier. We then increment the seekPtr by the amount we've written in that while-loop iteration, increment the amountWritten by the same amount, then decrement the amountLeft by the same amount. IF the seekPtr is greater than the current inode.length, we update the inode.length to the new seekPtr position to indicate the file size has expanded since then. Once the last write has finished, we write the inode back to disk with inode.toDisk(ftEnt.iNumber) then return the total amount of bytes written.

delete(String filename)

We call directory.ifree(ftEnt.iNumber) along with close(ftEnt) and if and only if both of these calls succeed have we deleted the file.

seek(FileTableEntry ftEnt, int offset, int whence). If whence = 0, then the seekPtr is at the offset. If whence = 1, then it's at the current location + the offset. If whence = 2, it's at the end of the file fsize(ftEnt) + offset. We then check if the seekPtr is < 0… if so, we set it to 0 (clamp at 0). If the seekPtr is past the file size, we clamp it at the file size (fsize(ftEnt)).

deallocAllBlocks(FileTableEntry ftEnt) is meant to reset all blocks belonging to this FileTableEnty. It'll first call superblock.returnBlock(ftEnt.inode.direct[i]) then iterate through the direct blocks, set the index to -1 again to indicate they're unused. This'll put all the freed blocks back to the front of the free list. Deallocating the indirect block is trickier. We need to first call byte[] indirectBlock = ftEnt.inode.unregisterIndexBlock(). Depending on if the index block was actually used, indirectBlock will be null if not used. If not null, we iterate in offsets of two. We get the block number by calling SysLib.bytes2short(indirectBlock, offset). Check if the block number returned is -1 or not. If not, we return that block number to be added back to the free list. We then write the inode back in this cleansed state back to disk using toDisk.

Process of development

We initially had a lot of trouble getting started on this project because of all the dependencies there seemed to exist between the various classes but began to clear up once we finished Directory.java. This class had no dependencies we needed to worry about and only needed to ensure we had the interface for this class in check with the others. We would do incremental development with this class, compile, and run with the original ThreadOS programs by running Test5. Once we had this class done we ran into more issues with FileTable.java and inode.java because we had to develop both at the same time. It was definitely difficult but we sort of reverse engineered what the inode class fully needed to do by doing development, compiling, and running with the original ThreadOS files. We would get things like “Exception in thread "Thread-5" java.lang.NoSuchMethodError: Inode.unregisterIndexBlock()[B”, “Exception in thread "Thread-5" java.lang.NoSuchMethodError: Inode.registerIndexBlock(S)Z”, “Exception in thread "Thread-5" java.lang.NoSuchMethodError: Inode.registerTargetBlock(IS)I”, “Exception in thread "main" java.lang.NoSuchMethodError: Inode.findTargetBlock(I)” which gave us the methods the inode class should have as well as their return and parameter arguments. From these error messages we began seeing which class was calling which of the methods which cleared up the dependencies that weren’t evident initially. This was our way of hashing through each class.

Breakdown of workload

Planning: Pair-effort

Directory.java: Pair-effort

Kernel.java, TCB.java, SysLib.java, and Scheduler.java modifications: Ji Kang

inode.java: Ji Kang

FileTable.java: William Eng

SuperBlock.java: Ji Kang

FileSystem.java: Pair-effort

Testing: Pair-effort

FSTest.java: William Eng

Report: Pair-effort