

# CSS 430

## Final Project

### File System

Name: Hong Nguyen

Due Date: June 7<sup>th</sup>, 2024

## Contents

<b>Specification .....</b>	<b>3</b>
<b>Design .....</b>	<b>3</b>
<b>File System Big Picture .....</b>	<b>3</b>
<b>File System Components Implementation .....</b>	<b>4</b>
<b>SuperBlock.....</b>	<b>4</b>
<b>Inode .....</b>	<b>6</b>
<b>FileTableEntry .....</b>	<b>9</b>
<b>FileTable.....</b>	<b>10</b>
<b>FileSystem.....</b>	<b>11</b>
<b>SysLib .....</b>	<b>17</b>
<b>Kernel.....</b>	<b>17</b>
<b>How to Test.....</b>	<b>18</b>
<b>Test Output.....</b>	<b>18</b>

## Specification

Design a Single Level File System for the ThreadOS. The File System must support format, open, read, write, seek, close, delete, and fsize. Since Single Level File System is the most basic file system, it only has a root directory that manages all the files under it. This introduces an advantage for easy implementation. However, the limitation is that each file name must be unique and no sub-directory is allowed.

## Design

The design is broken down into two parts: the big picture overview of the File System, and the implementation of the individual components. The big picture covers how the file system works, and how each component interacts with the each other. The components implementation will go into more detail about the structure and operations of each class.

### File System Big Picture

Below is the overall picture of the File System. Starting with the System calls at the highest level. System calls generate an interrupt to the kernel. The kernel then looks into the TCB to get the file descriptor. After that, the kernel calls the respective method from the file system to perform the needed operation. The file system contains a root directory, the file table (the file descriptor table), and the superblock. The root directory contains inodes, which store the information of each file. Finally, the lowest level is the direct raw read and write to the disk.

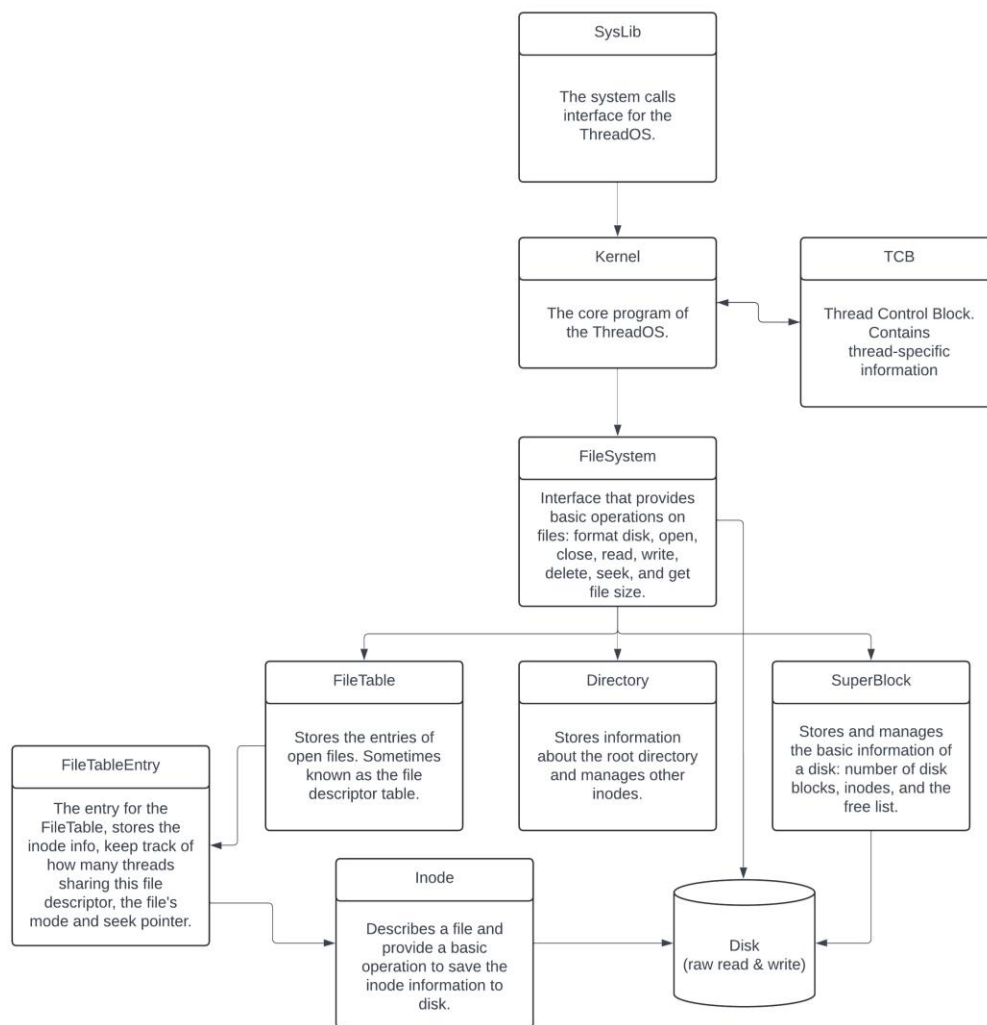


Figure 1: The Big Picture of the File System

## File System Components Implementation

### SuperBlock

This is disk block 0. It contains the number of disk blocks, inodes, and the block number of the head of the free list.

To initialize the superblock, read disk block 0 and start parsing information. The first 4 bytes contain the totalBlocks number, the next 4 bytes contain the total inode blocks, and the next 4 bytes contain the block number of the free list block. For the first time initialize the superblock, all the numbers will be 0, thus, we need to format the superblock.

I decided to use a Stack for the free list because it provides faster and easier insertion and retrieval (constant time).

Below is the flow chart for the format method:

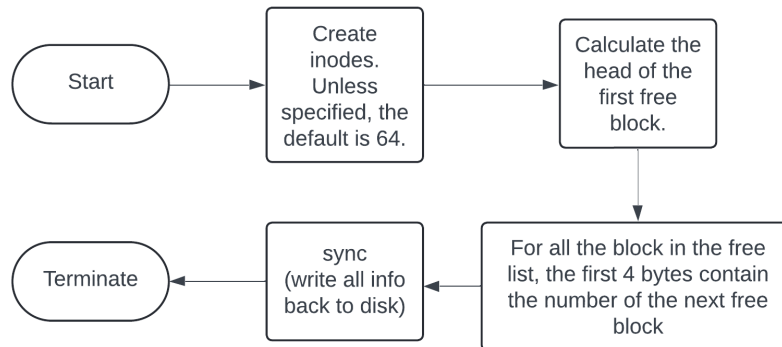


Figure 2: SuperBlock format Flowchart

Next is the flow chart for getting a free block from the free list:

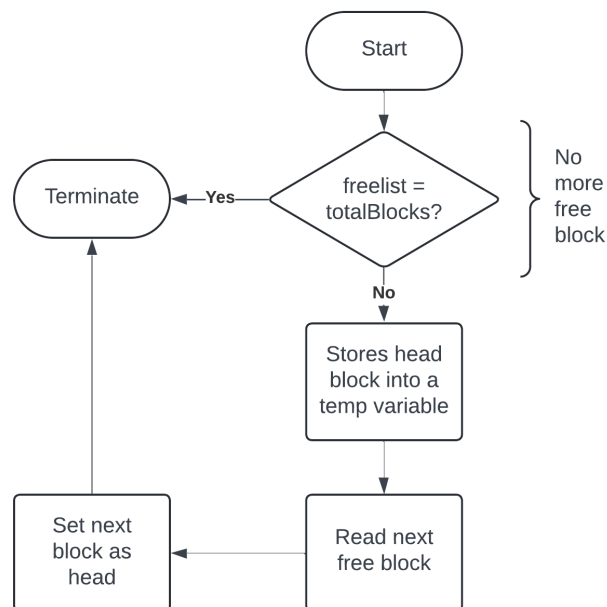


Figure 3: SuperBlock getFreeBlock Flowchart

Lastly, the flowchart for returning a free block to the free list:

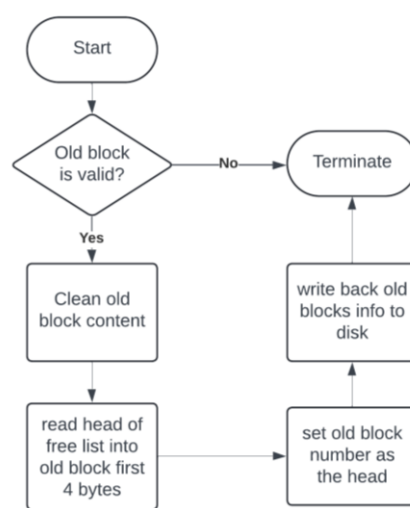


Figure 4: SuperBlock returnBlock Flowchart

## Inode

Each inode describes a file. The inode stores the length of the file, keeps track of how many file table entries are pointing to this, a flag to indicate what/how the file is being used, 11 direct pointers, and 1 indirect pointer. Figure 5 shows a visual of the Inode.

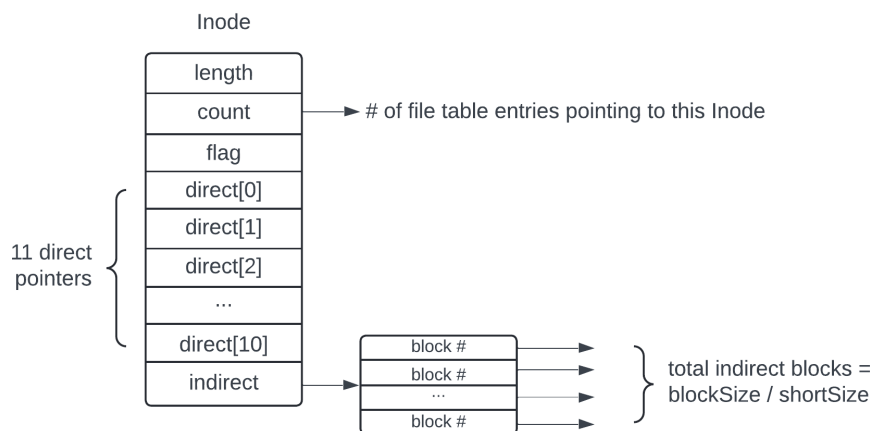


Figure 5: Inode Structure

A default Inode constructor initializes all the values to default: length and count to 0. Flag to 1 ( unused), direct and indirect pointers to -1.

The Inode constructor that takes in an inode number retrieves an existing inode from disk to memory. Figure 6 illustrates how this process works.

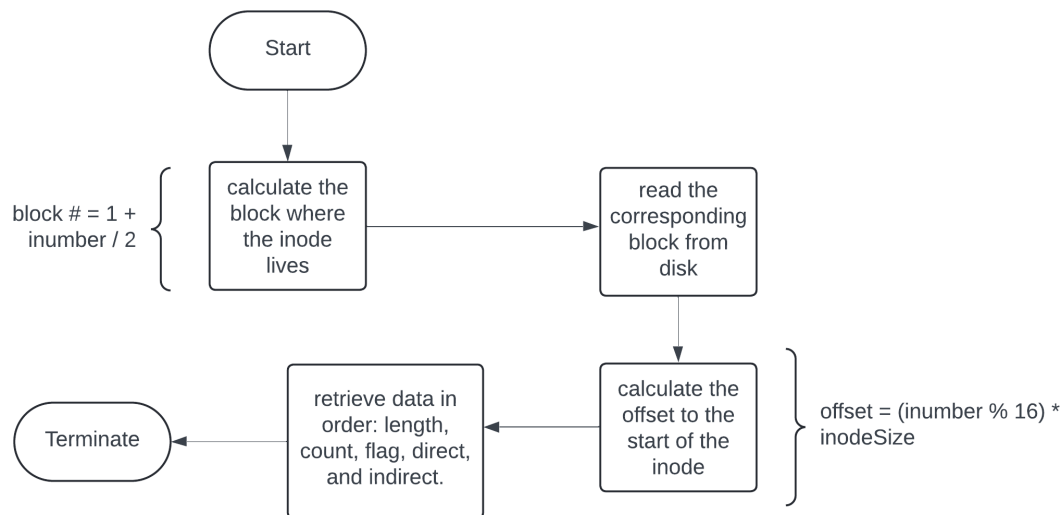


Figure 6: Flowchart of How to Retrieve an Inode from Disk to Memory

The toDisk method does the opposite. It saves the current inode information back to the disk. Figure 7 illustrates this process.

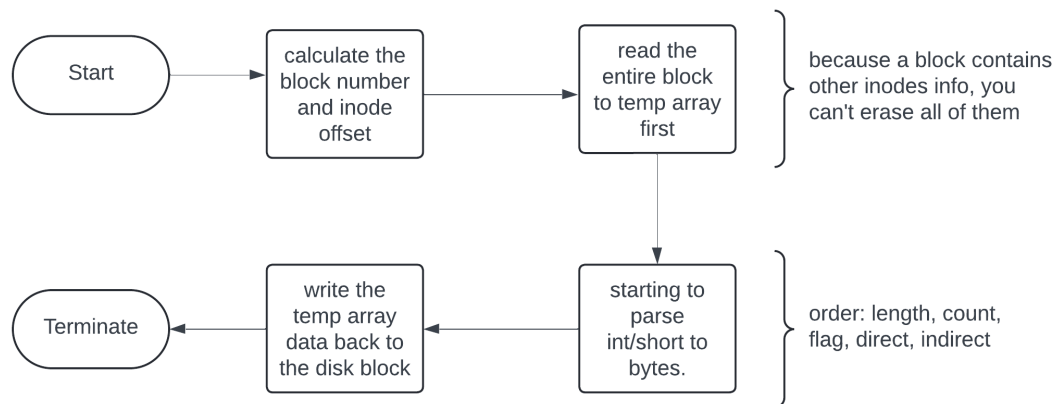


Figure 7: Inode toDisk Flowchart

**Directory:**

This is the root directory that maintains each file size and name. Since we know the maximum number of files the directory can have in advance, the fastest and simplest data structure is an array. The index of the array corresponds to the number inode number. Using an array has the advantage of random access.

There is a 1D array to store the file's size. Another 2D array stores the file names. The column index is the inode number, and the characters of the file name are stored in the rows corresponding to the appropriate column.

To initialize a Directory, we create both arrays' lengths to the maximum number of files. Then set all values to 0. The root directory is for inode 0. Which is the first index of the arrays.

Figure 8 shows the process of converting a bytes array retrieved from disk to the directory information.

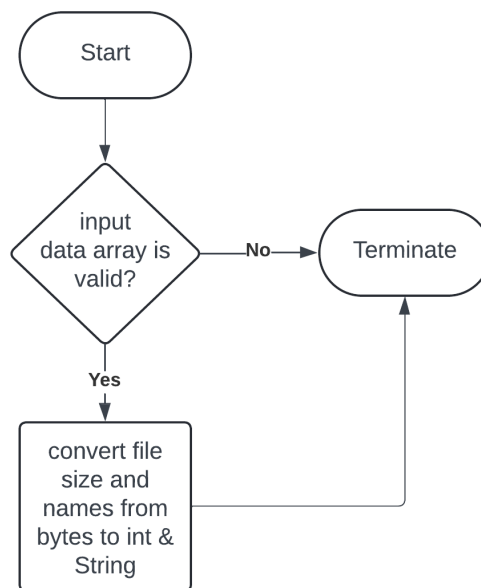


Figure 8: Bytes to Directory Process

The `directory2bytes()` method is the opposite. It converts the current directory information to a bytes array. The process is shown in Figure 9.



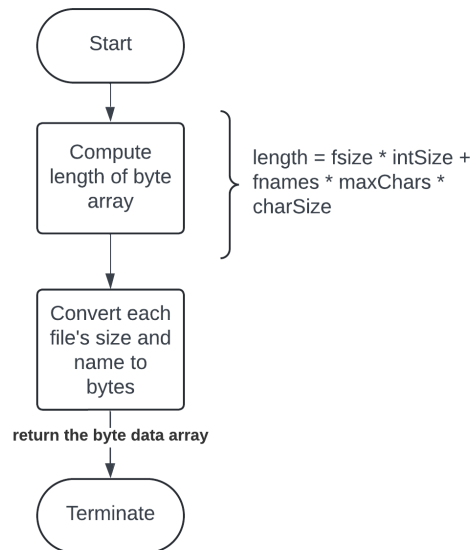


Figure 9: Directory to Bytes Process

To allocate a new inode for a given file name is fairly simple. We need to find an empty inode (whose file size value is 0), and put the new file name in it. This can be accomplished using a single for loop with linear time complexity  $O(n)$ .

To deallocate an inode, we first check if the input inode number is valid. Then, we take advantage of the random access of the array and use the input inode number as an index to reset the file length and name to the default values.

To find an inode number corresponding to the given file name (namei method), a simple for loop would suffice. Go over each entry and compare the file size and file name. Then, return the index (the inode number) when there's a match.

### FileTableEntry

This contains the seek pointer, inode, inode number, count of how many threads share this entry, and the current mode. This class is given and was already fully implemented, so I only added additional Javadoc and inline comments to help me understand the overall class.

## FileTable

This is the system-maintained file table (file descriptor table) that is shared among all user threads. Each entry contains the information mentioned above in the FileTableEntry section.

The FileTable contains a root directory and a vector that stores all the file table entries opened. Vector is used instead of ArrayList because Vector is thread-safe, while ArrayList is not. To initialize the FileTable, set the root directory to the input directory and create a new empty table.

To allocate a new FileTableEntry (falloc method) with the given file name and mode, follow the process in Figure 10.

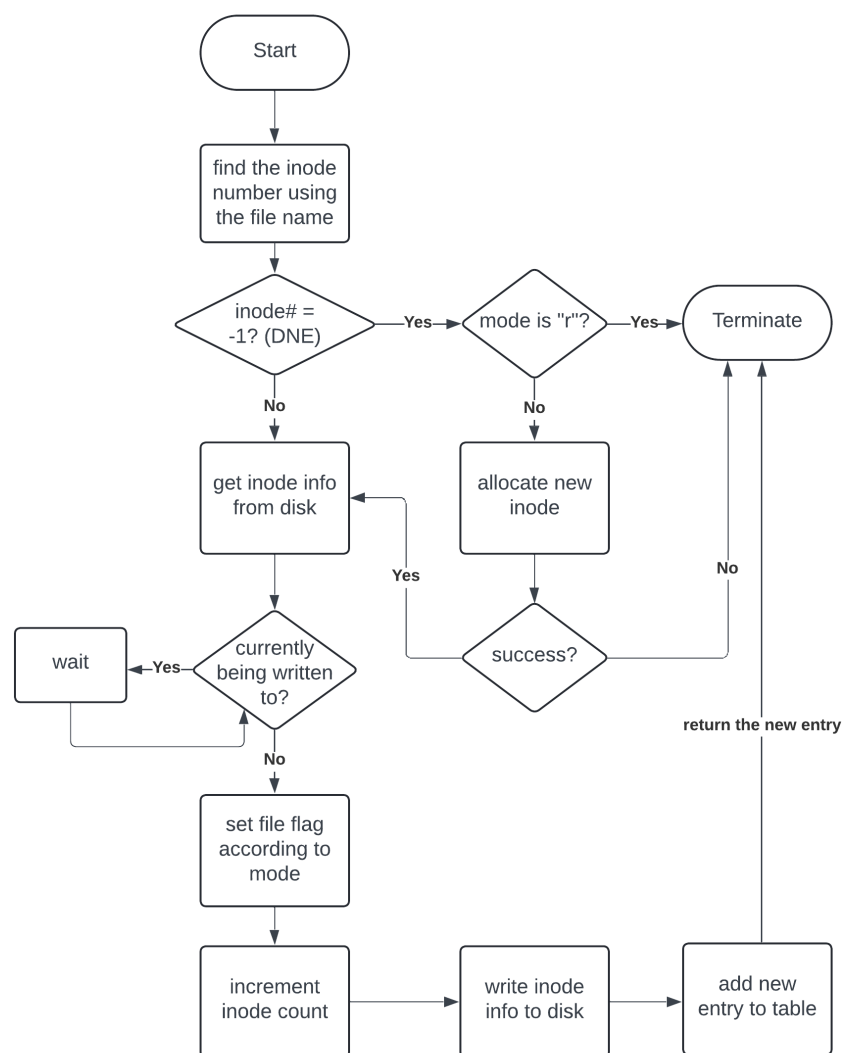


Figure 10: Allocate a New FileTableEntry Flowchart

To remove a FileTableEntry (ffree method) when given an entry, follow the process illustrated in Figure 11.

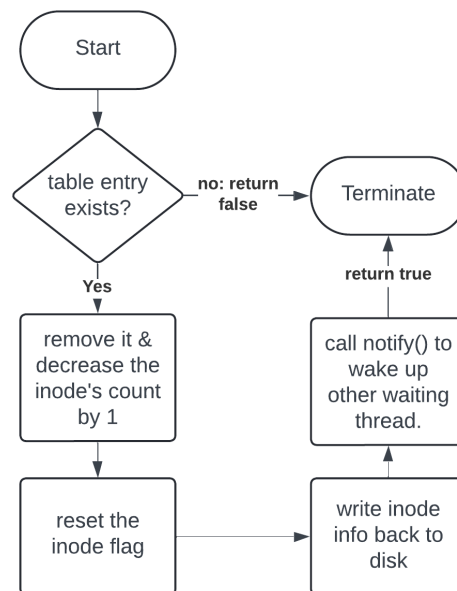


Figure 11: Process to Remove a FileTableEntry from the System-Wide File Table

To check if the file table is empty (fempty method), we just call the vector isEmpty method to check.

## FileSystem

This class is where all the components come together to form a single-level file system. The FileSystem contains the superblock, root directory, and the file table. As expected, the constructor receives the total disk blocks and initializes the superblock, root directory, and file table accordingly. To initialize the root directory, we open it (by a call to the open method), and read the directory data from the disk, then convert all that information to the root directory in memory.

Next, I'll go over each method in detail. Each method's order matches the order the method appears in the FileSystem.java file.

For the sync() method, Figure 12 illustrates how to synchronize the file system data to the disk.

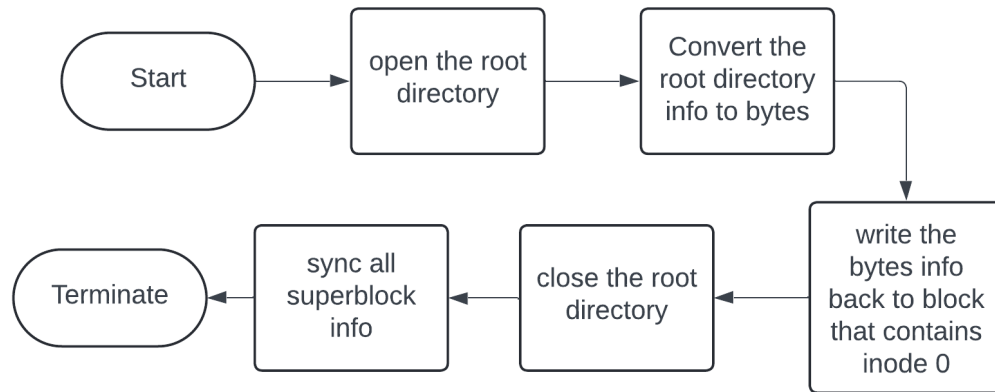


Figure 12: Sync Process Flowchart

To format the file system, follow the flowchart in Figure 13.

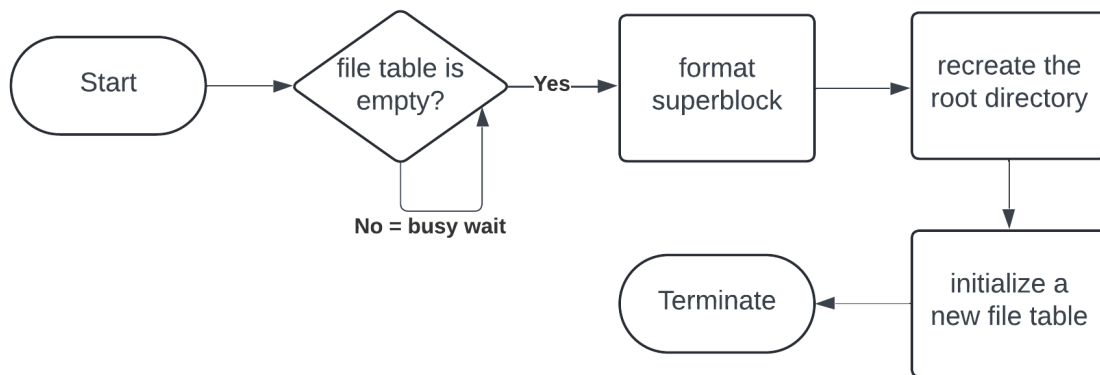


Figure 13: Process of Formatting the File System

To open a new file given the file name and mode, follow the process outlined in Figure 14.

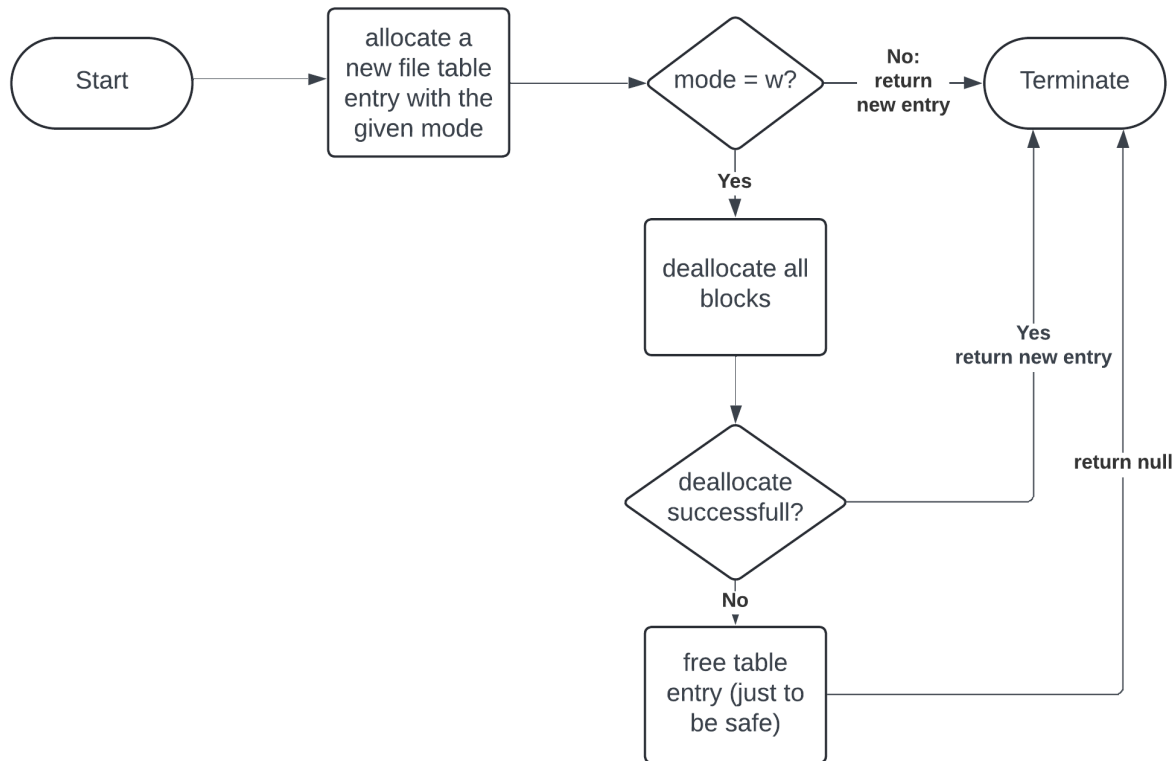


Figure 14: Process of Opening a File

To close an entry in the file table, decrement the count that keeps track of how many threads are sharing this file. If the count is greater than 0, this means that other threads are currently using it, we can just successfully return at this point. However, if the count is 0, meaning that no more threads are using this file, we called the `ffree` method to free (delete) the table entry from the file table.

To get the file size in bytes of a file stored in an entry, return the length of the file by accessing the inode and return its length. If the input entry is invalid (null for instance), then -1 is returned as an error code.

Read is one of the two major operations of the file system. This method is complicated, and I describe the detailed algorithm in the source code. Figure 15 illustrates the general approach (language independent) to read a file given a file table entry and the buffer that the information will be read into.

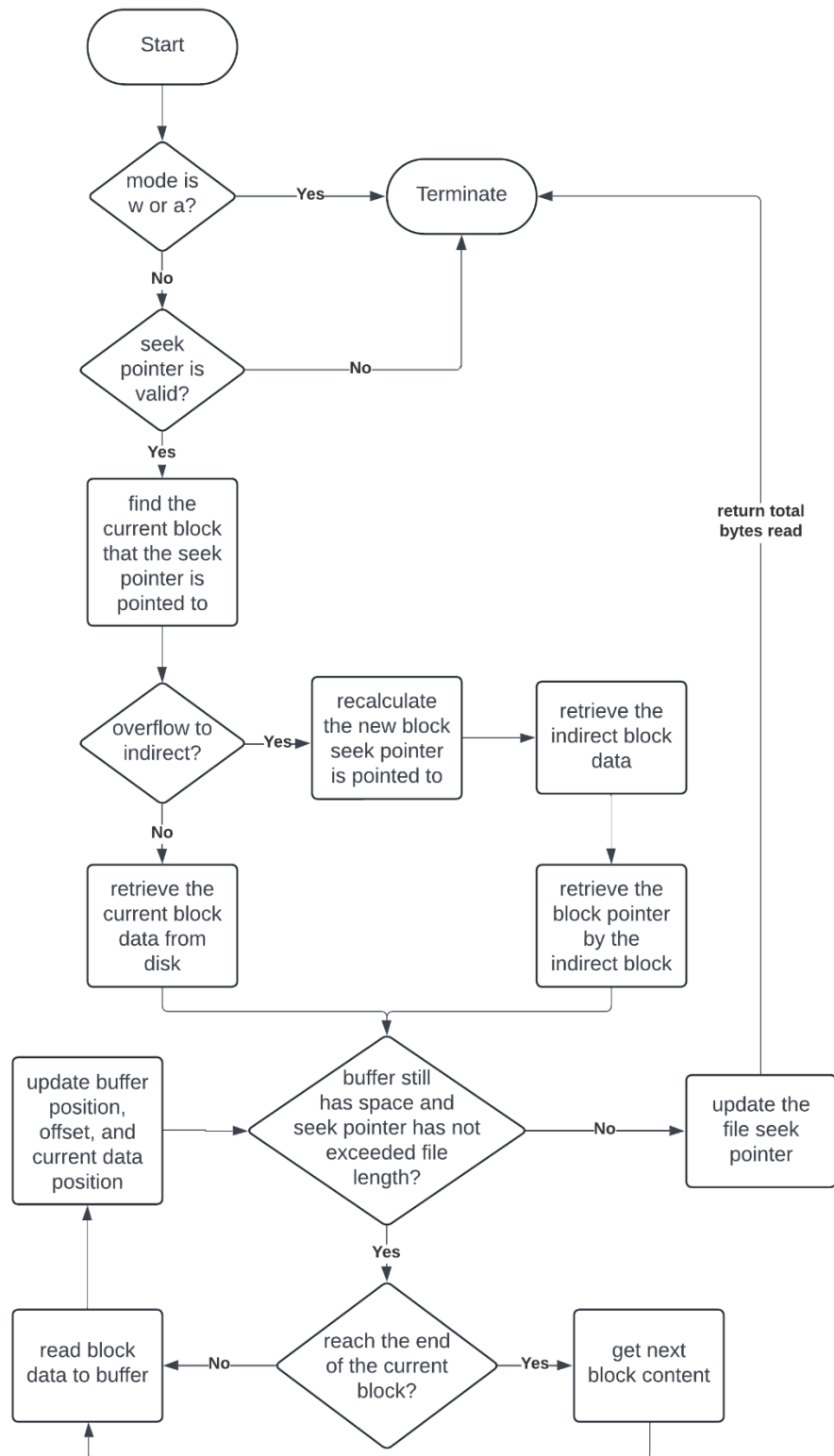
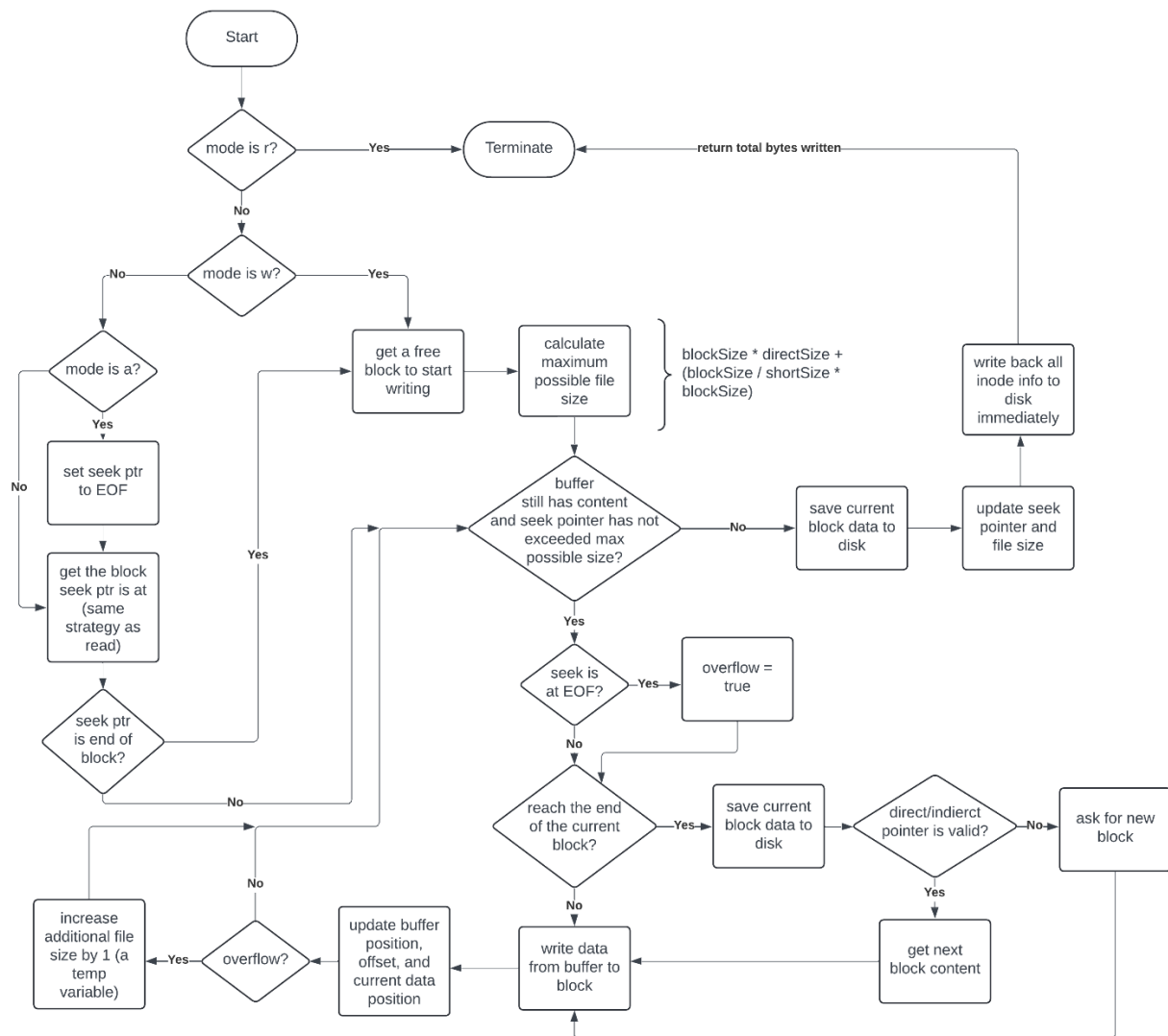


Figure 15: Process of Reading a File

Write is the second major operation of the file system. Part of the general approach is similar to reading, however, writing is more complicated than reading in which we need to allocate new blocks if the information we have written is longer than the file's length. Figure 16 illustrates this process.



### Figure 16: Process of Writing to a File

For the `deallocAllBlocks()` methods, follow the diagram in Figure 17. Note that this method calls a helper `clearAndFreeBlock()` that takes in a block number, and wipes all the disk block content.

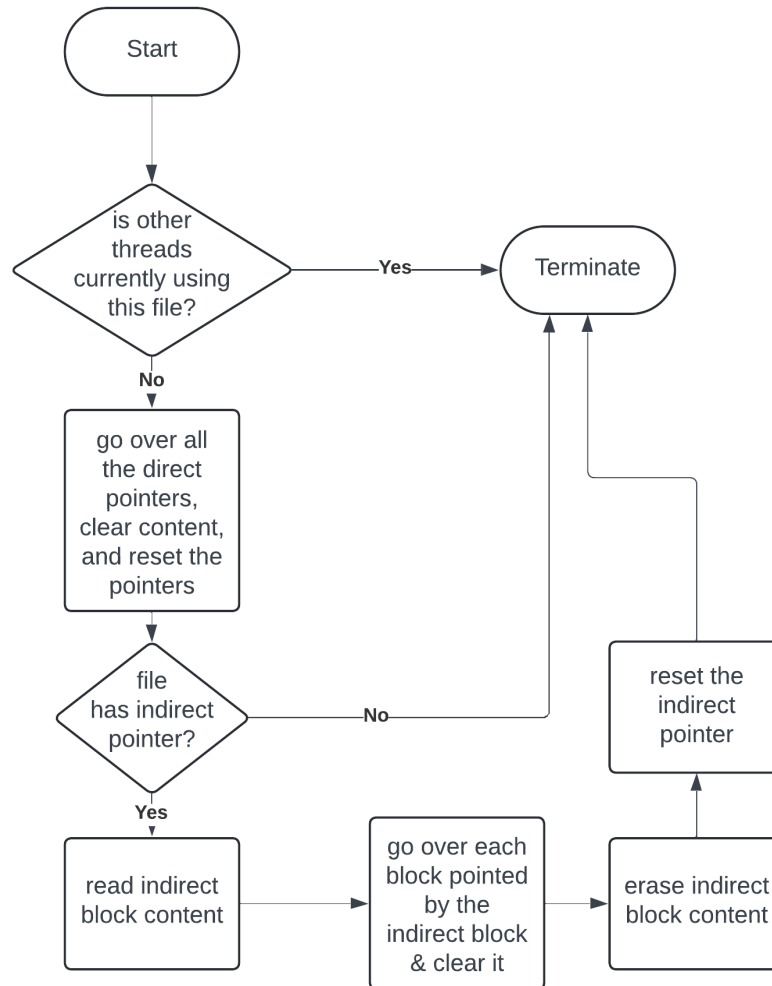


Figure 17: Process of Deallocating All Blocks a File is Using

Deleting a file is quite simpler than the other methods. We first open the file in write mode, this effectively clears all blocks's content (triggered by the open in write mode operation). All blocks are cleared only when no more threads use this file. If that operation is successful, then we close the entry and free the file from the root directory. If there are still threads pointing to this file, only the free from root directory operation is successful, which means that a new attempt to open this file will fail. The close operation will result in false until the last thread closes it, which will clear the inode content and save to disk.



Finally, the seek method updates the seek pointer. The process is shown in Figure 18.

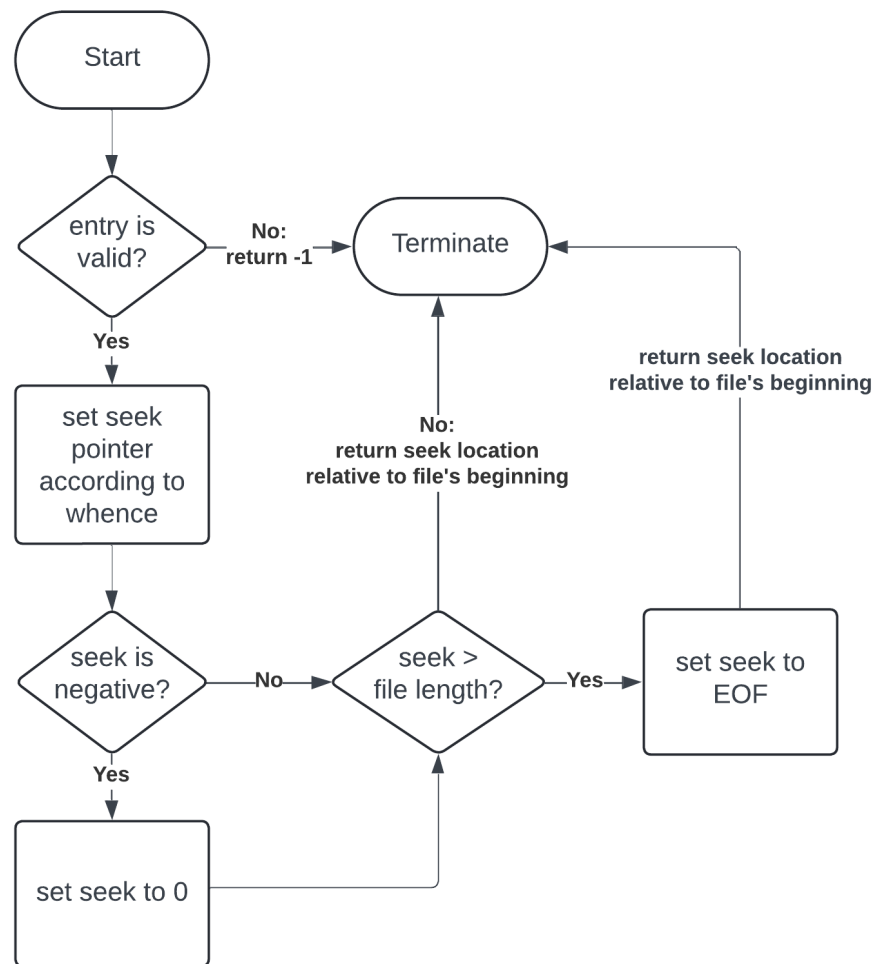


Figure 18: Process of Setting the Seek Pointer

## SysLib

I added a set of required methods from the Final Project specification document. I only added Javadoc on the methods I added so that it is distinguishable from the other given SysLib methods. Similar to other methods, each system call generates a software interrupt to the Kernel and passes in the appropriate arguments.

## Kernel

The following cases in Kernel interrupt are modified: FORMAT, OPEN, READ, WRITE, SEEK, CLOSE, DELETE, and SIZE. These implementations were given by Dr. Parsons on the Final

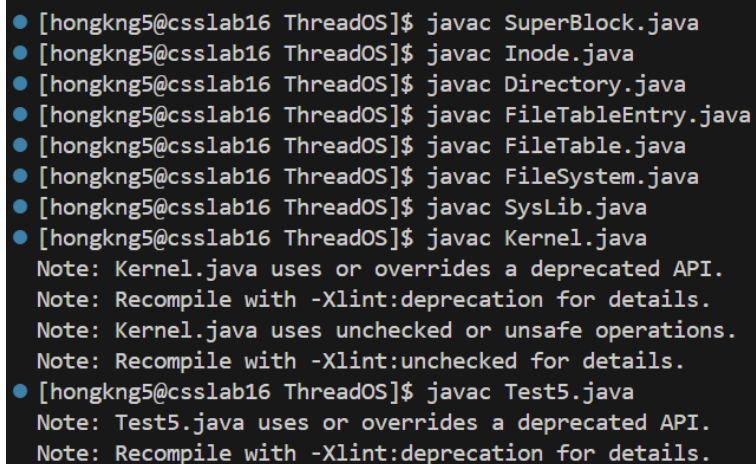
Project Notes slide. Most implementations are similar, in which we either call the method directly from the file system and pass in the arguments, or go through the TCB to get the File Table Entry and call the file system method to pass in arguments.

## How to Test

1. Compiled the following classes:
  - a. Superblock
  - b. Inode
  - c. Directory
  - d. FileTableEntry
  - e. FileTable
  - f. FileSystem
  - g. SysLib
  - h. Kernel
2. Assuming that all these files are in the same folder as the other files that are necessary to run ThreadOS.
3. Then run the ThreadOS using: `java Boot`
4. Run: `1 Test5`

## Test Output

The following images show the result when compiling and running Test5.java:



```
• [hongkng5@csslab16 ThreadOS]$ javac SuperBlock.java
• [hongkng5@csslab16 ThreadOS]$ javac Inode.java
• [hongkng5@csslab16 ThreadOS]$ javac Directory.java
• [hongkng5@csslab16 ThreadOS]$ javac FileTableEntry.java
• [hongkng5@csslab16 ThreadOS]$ javac FileTable.java
• [hongkng5@csslab16 ThreadOS]$ javac FileSystem.java
• [hongkng5@csslab16 ThreadOS]$ javac SysLib.java
• [hongkng5@csslab16 ThreadOS]$ javac Kernel.java
Note: Kernel.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Kernel.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
• [hongkng5@csslab16 ThreadOS]$ javac Test5.java
Note: Test5.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

```

○ [hongkng5@csslab16 ThreadOS]$ java Boot
threadOS ver 1.0:
threadOS: DISK created
Superblock synchronized
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->1 Test5
1 Test5
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
1: format( 48 ).....Superblock synchronized
successfully completed
Correct behavior of format.....2
2: fd = open( "css430", "w+" )....successfully completed
Correct behavior of open.....2
3: size = write( fd, buf[16] )....successfully completed
Correct behavior of writing a few bytes.....2
4: close( fd ).....successfully completed
Correct behavior of close.....2
5: reopen and read from "css430"..successfully completed
Correct behavior of reading a few bytes.....2

```

```

6: append buf[32] to "css430".....successfully completed
Correct behavior of appending a few bytes.....1
7: seek and read from "css430"....successfully completed
Correct behavior of seeking in a small file.....1
8: open "css430" with w+.....successfully completed
Correct behavior of read/writing a small file.0.5
9: fd = open( "bothell", "w" )....successfully completed
10: size = write( fd, buf[6656] ).successfully completed
Correct behavior of writing a lot of bytes....0.5
11: close( fd ).....successfully completed
12: reopen and read from "bothell"successfully completed
Correct behavior of reading a lot of bytes....0.5
13: append buf[32] to "bothell"...successfully completed
Correct behavior of appending to a large file.0.5
14: seek and read from "bothell"...successfully completed
Correct behavior of seeking in a large file...0.5
15: open "bothell" with w+.....successfully completed
Correct behavior of read/writing a large file.0.5
16: delete("css430").....File css430 doesn't exist for read
successfully completed
Correct behavior of delete.....0.5
17: create uwb0-29 of 512*13.....successfully completed
Correct behavior of creating over 40 files ...0.5
18: uwb0 read b/w Test5 & Test6...
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test6.java: fd = 3successfully completed
Correct behavior of parent/child reading the file...0.5
19: uwb1 written by Test6.java...Test6.java terminated
Correct behavior of two fds to the same file..0.5
Test completed
-->

```