Jessica Nguyen, Audrey Nguyen

CSS 430, Final Project Report

## **FILE SYSTEM SPECIFICATIONS**

This project was to create a single level file system. The java files submitted were edited from the skeleton files provided in the assignment as well as pre-existing files from Thread OS.

**Figure 1:** Visual diagram of the structure of the file system

## 

### **Assumptions**

Our file system assumes the follow criteria:

* The file system is a single level file system, meaning there is only 1 directory.
* There are a max of 1000 file blocks on the disk
* Each block is 512 bytes
* There are 64 inodes in the superblock.

### **Limitations**

* Our program is under the following limitations:
* Because this is a single level file system, we will not be implementing any tree or graph structures of the directory
* All the users will have to share a common file directory. Therefore, their file access types will be the same
* There are max 1000 files, therefore, the page fault will be higher if we want to access different localities at the same time

## **INTERNAL DESIGN**

### **Super Block:** SuperBlock.java

The SuperBlock class represents the first disk block of the file system. It describes the number of disk blocks, the number of inodes and the block number of the head of the free list.

**Methods:**

* SuperBlock()
  + The constructor which reads the disk size from the input and initializes the number of totalBLocks, totalInodes and freeList which keeps the location of the free blocks.
  + Re-formats the disk if the contents reading in from the SysLib disk doesn’t line up with the disk format.
* format(int files)
  + Sets the total blocks (files) to 1000 and totalInodes to the input integer files.
  + It sets the index for the free list based on how many files there are, and initializes each block in the disk to 512 bytes.
  + For example when the parameter iNodes is 32, format() will yield 2 blocks of Inodes.
* sync()
  + Write back totalBlocks, totalInodes, and freeList to Disk to update the new specs of the Superblock with int2bytes() and rawwrite() from SysLib.
* returnBlock(short blockNumber)
  + Enqueues the given block blockNumber to the end of the free list.
* getFreeBlock()
  + Dequeues the top block from the free list and returns the location of the free block.

### **Inode:** Inode.java

This class represents an inode file in the file system. The inode includes 12 pointers of the index block. The first 11 of these pointers point to direct blocks. The last pointer points to an indirect block. Additionally, each inode has a length of the file, number of table entries that point to this inode, a flag to indicate the status of the inode. The flags are as follows: UNUSED, USED, READ, WRITE, and DELETE.

**Methods:**

* Inode()
  + A constructor which sets the length to 0, count of threads to 0, flag to unused, direct pointers to -1, and indirect pointers to -1.
* Inode( short iNumber )
  + A constructor that retrieves the inumber from the disk and initializes length, count, flag, direct pointer and indirect pointer based on the inumber.
* void toDisk(short iNumber)
  + Saves the iNode associated with the parameter iNumber to the disk.
* int getOffset(int iNumber)
  + Returns the current block's offset associated with the parameter iNumber.
* getIndexBlockNumber()
  + Returns indirect pointer to index block.
* setIndexBlock(short indexBlockNumber)
  + Sets indirect pointer to input indexBlockNumber.
* findTargetBlock(int offset)
  + Returns the block number based on the given input offset.
* int setTargetBlock(int iNumber, short fBlock)
  + Registers the block into one of the direct or indirect blocks.
* byte[] deleteIndexBlock()
  + Deletes one of the registered blocks in indirect blocks.

### **Directory:** Directory.java

The Directory class represents the single level root directory of the file system and serves as a place where information about each file in the directory is stored. The root directory is the only directory in the system: more directories cannot be created. Directory.java contains two arrays representing the sizes and names of the file entries: int fsize[] and char fnames[][] respectively. The array’s indexes represent the iNumber of each file. Each file name has a maximum of 30 characters and 60 bytes in Java.

**Methods:**

* Directory( int maxInumber )
  + Constructor that takes in a single integer parameter which represents the maximum number of files within the system.
* bytes2directory( byte data[] )
  + Initializes the directory by converting a byte array that was passed in into the data of the directory system.
  + The byte array’s contents represents the sizes of each file in the system and the names of each file.
* byte[] directory2bytes( )
  + Converts the directory’s file names and sizes into a single byte array.
* short ialloc( String filename )
  + Accepts a string that represents a filename of a new file that will be located in the directory.
  + The method allocates a new inode number for the file to be created. It returns 0 if it is successful and -1 if it is not.
* boolean ifree( short iNumber )
  + Frees the file in the system with the iNumber that is passed into the method.
  + It achieves this through deallocating the inode associated with the iNumber as well as deleting the associated file.
  + It returns true if it is successful and false if it is not.
* short namei( String filename )
  + Returns the iNumber associated with the name of the file passed in.
  + It returns 0 if it is successful and -1 if it is not.

### **File Table:** FileTable.java

This class represents the file table, which is stored internally within the class as a vector. It also contains the root directory. The instance of FileTable in our system is shared by all the threads that are running. The class methods enables the users to allocate and delete files within the system to and from the table.

**Methods:**

* FileTable(Directory directory)
  + Constructor that instantiates a file (structure) table and creates a reference to the directory from the file system.
* synchronized FileTableEntry falloc(String fname, String m)
  + Allocates a file to the table with the name and mode passed in as strings.
* synchronized boolean ffree(FileTableEntry fte)
  + Frees the file table entry from the table.
* synchronized boolean fempty()
  + Check if the table is empty through checking if the vector containing the file entries is empty.

### **File Table Entry:** FileTableEntry.java

This class represents an entry or a single file within FileTable. It contains the global variables READONLY, WRITEONLY, READWRITE, and APPEND which represent the modes of each file entry. It was modified from the file provided by the One Note.

**Methods:**

* FileTableEntry(Inode i, short inumber, String m)
  + A constructor that sets the seek pointer to the file top
* static String getMode(String mode)
  + Retrieves the mode associated with the string passed in.

### **File System:** FileSystem.java

This class represents the whole file system. It allows users to format, open, read, write, close, and delete files as well as update the seek pointer and retrieve the size of files.

**Methods:**

* FileSystem(int blocks)
  + A constructor which creates superblock and formats disk with 64 inodes in default; creates directory, and registers "/" in directory entry.
  + Takes in an integer representing the number of blocks in the system
  + Creates the file table, constructs the root directory, and stores the directory in the file table.
* void sync()
  + Converts all necessary data to byte format and writes the data to the disk.
* boolean format(int files)
  + Formats the system to accommodate the amount of files the system will hold which is passed as the integer parameter files.
  + Returns true if successful and false if not.
* FileTableEntry open(String filename, String mode)
  + Opens the file with the user defined name, filename, in the user defined mode, mode.
* boolean close(FileTableEntry ftEnt)
  + Closes a thread in the ftEnt file table entry.
  + Commits all file transactions on this file, and unregisters fd from the user file descriptor table of the calling thread's TCB.
  + Returns true on success and false if failed.
* int fsize(FileTableEntry ftEnt)
  + Measures how much data ftEnt contains through returning the byte size of the entry passed in.
* int read(FileTableEntry fte, byte[] buffer)
  + Reads from file into the buffer.
  + Achieves
* write(FileTableEntry fte, byte[] buffer)
  + Writes from the buffer to the file.
* delete(String filename)
  + Deletes the file in the directory with the name filename
* seek(FileTableEntry ftEnt, int offset, int whence)
  + Moves the ftEnt's seek pointer to the appropriate position using the specified offset amount and starting position at whence.
* boolean deallocateBlocks(FileTableEntry fte)
  + Deallocates blocks associated with file table entry passed in.

### **Kernel:** Kernel.Java

The kernel was modified from Kernel\_org.java from ThreadOS. We made modifications to it so it could be compatible with the file system. We added the system calls to other classes like FileSystem to call our own methods that we implemented whenever an interrupt call is made. Such methods are sync, read, write, open, close, fsize, format, and delete depending on the input cmd parameter value.

### **Thread Control Block:** TCB.java

This file was modified from the existing java file in ThreadOS. We made minor modifications to it so it could be compatible with the file system.

**Methods:**

* synchronized int getFd(FileTableEntry entry)
  + Gets the file descriptor associated with the entry passed in
* synchronized FileTableEntry returnFd(int fd)
  + Gets the file entry associated with the file descriptor passed in

### **Test 5:** Test5.java

This class serves as the test driver for the file system. It has 18 tests that check the functionality of the file system. The results of Test 5 are included in the **Testing Output** section between **Figures 2** to **21** and are discussed in the **System Analysis** section. It is mostly unchanged from the code initially provided from ThreadOS with the exception of adding a clock to calculate performance times.

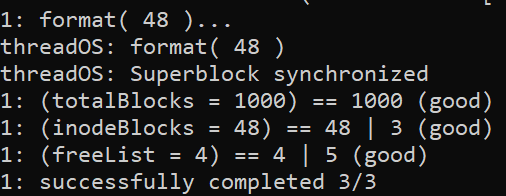
## 

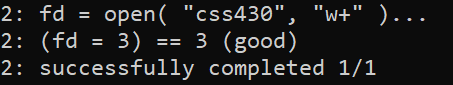
## **TESTING INSTRUCTIONS**

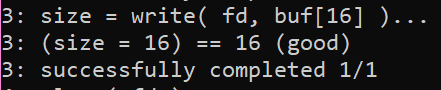
In order to test our file system, all java files submitted must be placed in the same directory as ThreadOS’s class files. Through command line, the java files may all be compiled with the line “javac \*.java”. Afterwards, start up ThreadOS with the command “java Boot”. When the command prompt “-->” is displayed, enter the command “l Test5.java” to test the program.

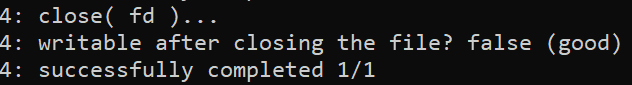
Test5 can be compiled on both Linux and Windows. In order to test the program properly, please use the class files from the Linux lab when compiling and running the file system.

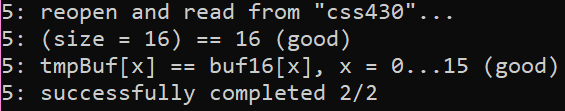
## **TESTING OUTPUT**

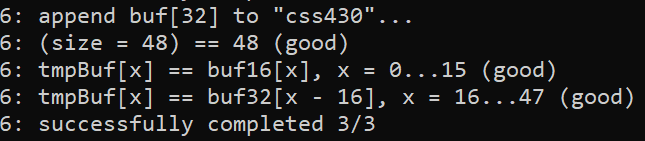
**Figure 2:** Output from Test #1

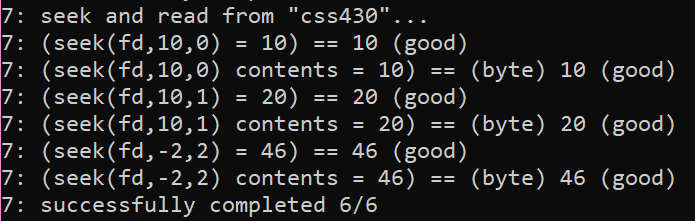
**Figure 3:** Output from Test #2

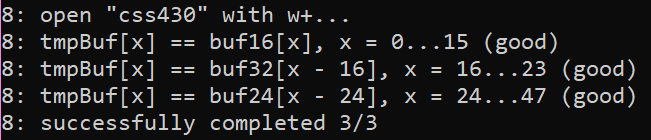
**Figure 4:** Output from Test #3

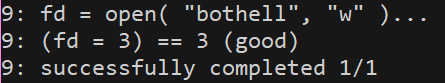
**Figure 5:** Output from Test #4

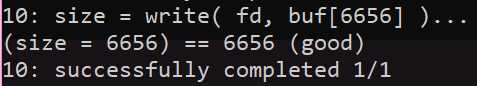
**Figure 6:** Output from Test #5

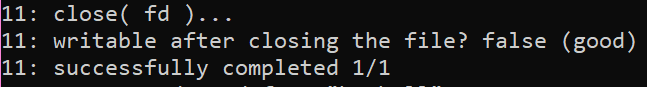
**Figure 7:** Output from Test #6

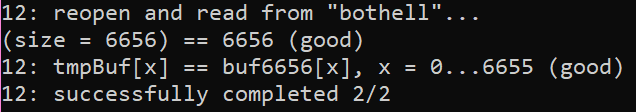
**Figure 8:** Output from Test #7 

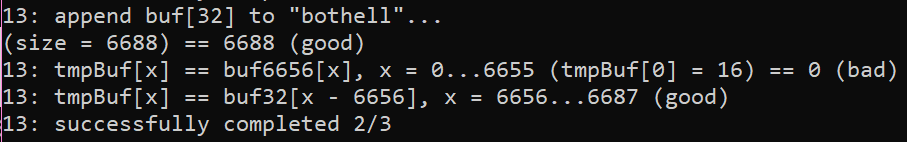
**Figure 9:** Output from Test #8

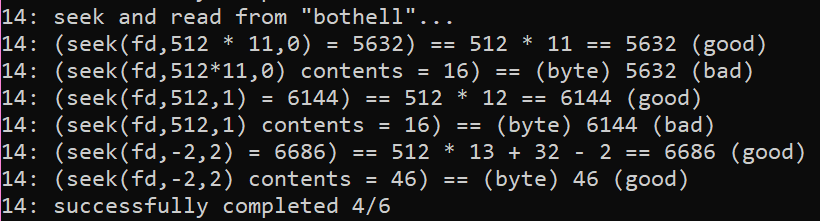
**Figure 10:** Output from Test #9

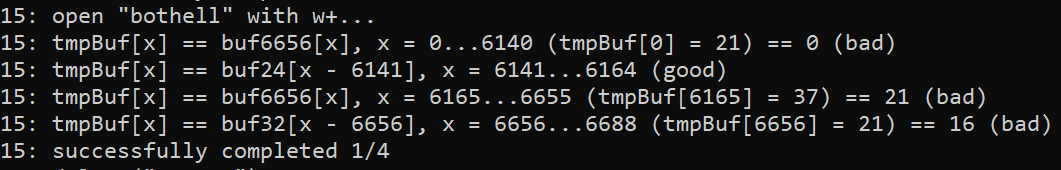
**Figure 11:** Output from Test #10

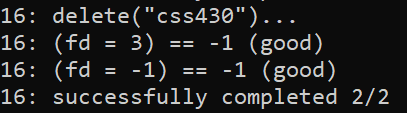
**Figure 12:** Output from Test #11

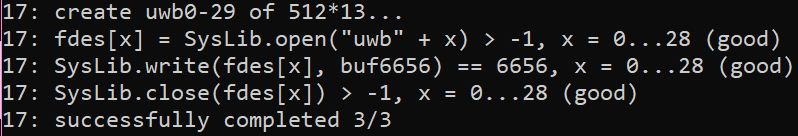
**Figure 13:** Output from Test #12

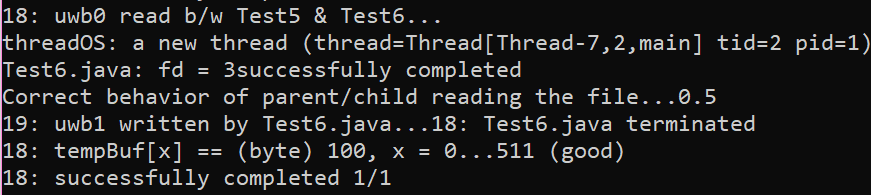
**Figure 14:** Output from Test #13

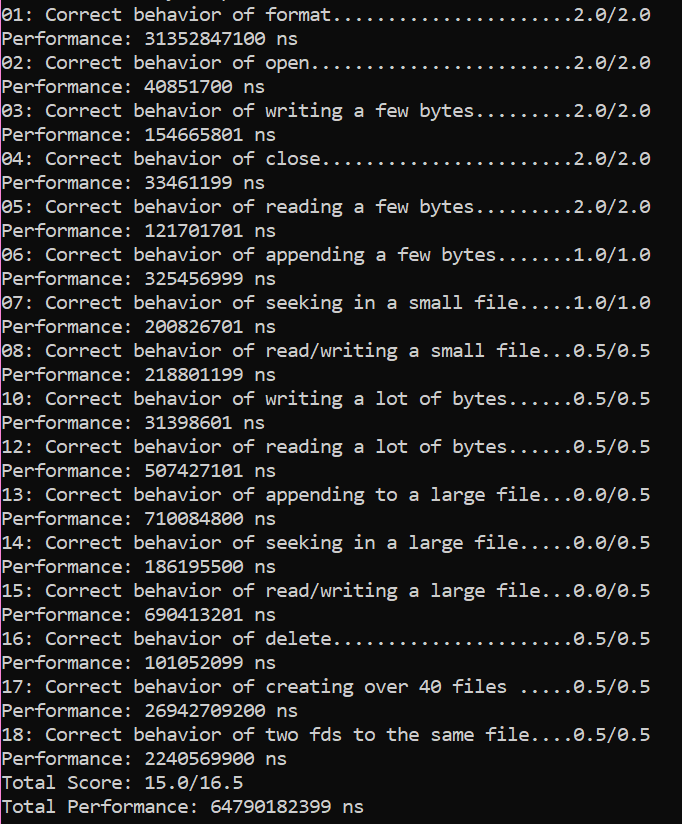
**Figure 15:** Output from Test #14

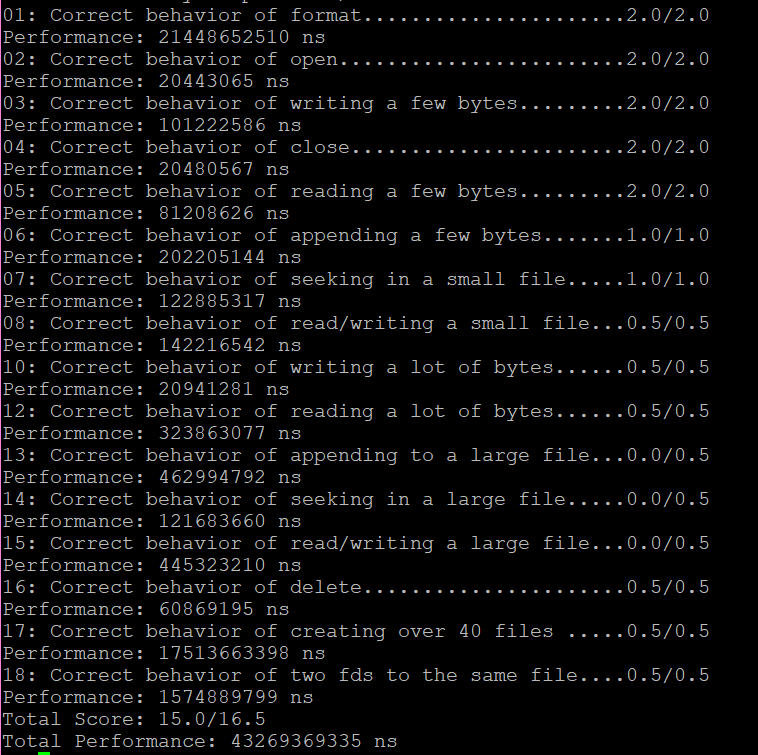
**Figure 16:** Output from Test #15

**Figure 17:** Output from Test #16

**Figure 18:** Output from Test #17

**Figure 19:** Output from Test #18 and #19

**Figure 20:** Output summary of test results (Windows)

**Figure 21:** Output summary of test results (Linux)****

## 

## **SYSTEM ANALYSIS**

### **Performance**

Our current file system works decently on a small scale as can be seen in **Figures 20** and **21**. On Windows, when we ran Test 5, all 18 tests took a time of 64790182399 ns (about 65 seconds) to run. On Linux it took 43269369335 ns (about 43 seconds). Retrieving files when we open, read, and write from them takes fairly a short amount of time. Test 2 which opened a file took 40851700 ns on Windows and 20443065 ns on Linux. Test 8 which read and wrote to a small file took 218801199 ns on Windows and 142216542 ns on Linux. Overall, test performance results were much quicker on Linux, probably due to there being less files on that system.

Our file system’s performance is not optimal but is satisfactory when handling a system that contains small amounts of files. When running Test5 on both Windows and Linux systems, the file system runs quickly. While the file system is not particularly slow, because of the changes we have made to Kernel.java and SysLib.java, ThreadOS runs slower than in previous projects we have worked on. For a larger system with more files and larger files, our current file system may experience difficulty in performance due to the fact it is a single level and due to these changes.

### **Current Functionality**

As seen in the output in **Figure 20 and 21**, our current file system passes 15 out of 18 tests with a score of 15.0 out of 16.5. Based on our results, our file system can be assumed to achieve the basic functionality of a file system. It overall successfully enables users to format, open, read, write, close, and delete files, update the seek pointer, and retrieve the size of files. Our file system appears to work best with small files. However, it can also seem to read and write large amounts of bytes successfully.

Despite passing tests about writing and reading large amounts of bytes, our system struggles with large file sizes. As seen in **Figures 14, 15, 16, 20**, and **21**, the functionality that our file system does not properly complete is appending, seeking, reading, and writing to a large file. This suggests that there may be some issue with our read and write methods. As of this report, we are not sure how to amend this and improve our methods to ensure that the system could handle large file sizes but it is something we would like to reflect on in the future.

### **Extended Functionality**

Our program is a single level directory. In considerations of potential extended functionality we could consider altering the structure of the file system to be a graph or a tree. This would make it possible to have files with identical names and make the organization of the file system cleaner. It would also make the structure better for a larger file system that contains many files since it would be easier to navigate through the file system and sort and retrieve files on a larger scale.