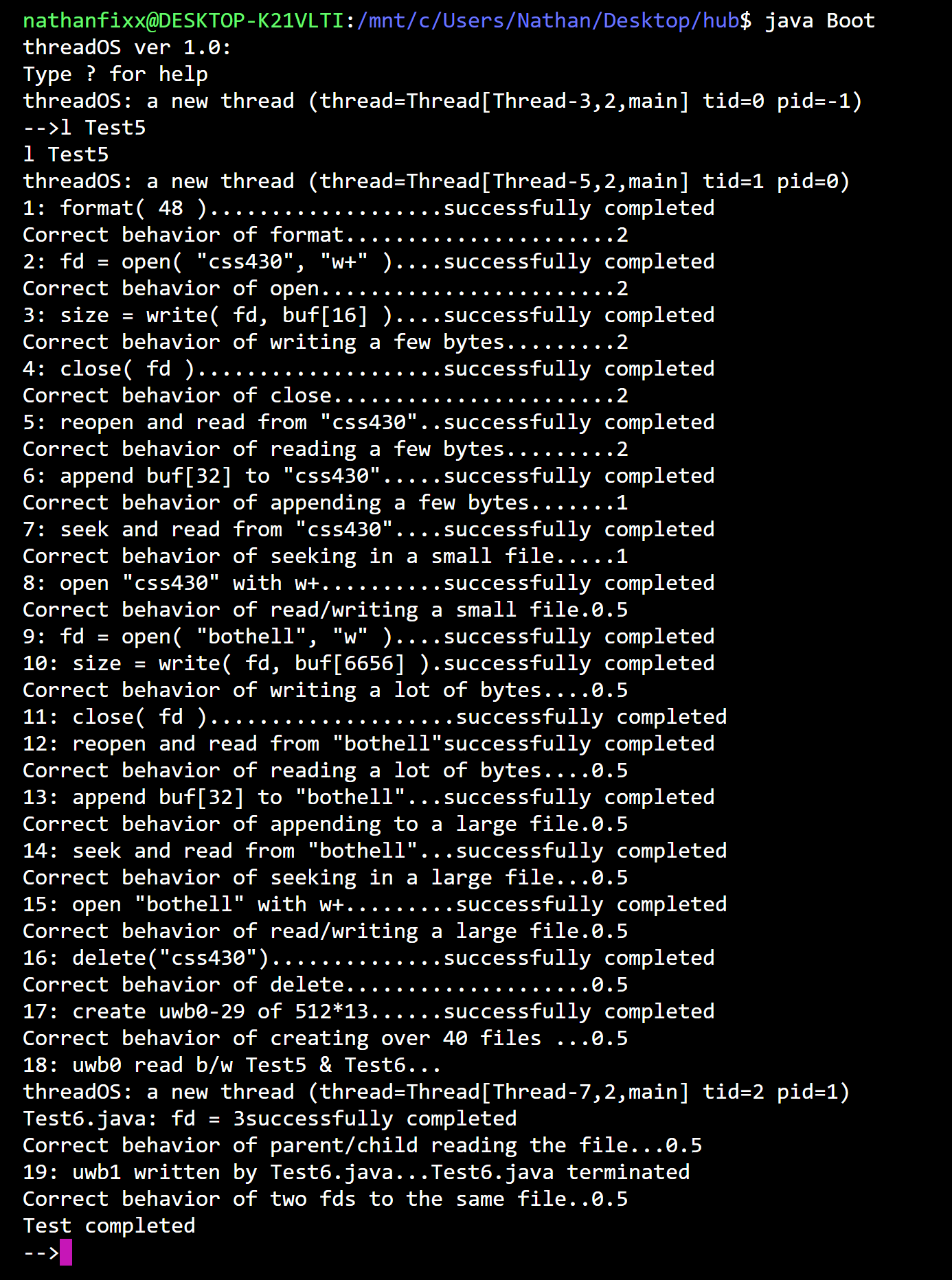
File System Documentation

By: Group 2 (Nathanel Fixx, Diego Guzman, and Misha Ward)

**Results:**

**Assumptions and Limitations  
Many of the assumptions that we made during this project and limitations that we faced were since we were writing this code for ThreadOS rather than a full-fledged operating system. Of course, we also made assumptions based off what the assignment description told us to do.  
  
Some of the assumptions that we made were:  
1. The file system would only be expected to handle the** format, open, close, read, write, seek, delete, fsize, and sync system calls.

**2. The file system maintains the file table shared among all user threads.**

**3. When a thread opens a file, it first allocates a new entry of the user file descriptor table in its thread control block. Then, it requests the file system to allocate a new entry of the system-maintained file table. Next, the file system locates the corresponding inode and records it in this file table entry before finally registering a reference to this file table entry in its file descriptor table entry of the TCB.**

**4. All commands a user would make to the file system were intentional and non-malicious.**

**5. The superblock will always be the first block in the disk.  
  
Some of the limitations that we had to deal with included:  
1. The maximum number of inodes (64 in 4 blocks). This limited the number of files we could handle.**

**2. The maximum number of blocks in each disk (1000)**

**3. Maximum character length of 30**

**4. No protection or warnings about unauthorized activity or actions.**

**Internal Design**

**Filesystem.java  
This class is the blueprint for the entire filesystem. It provides all the core functionality of a filesystem such as open, format, write, read, seek, delete, and close. It allows the user to make use of this functionality through specific commands. This is the class that is used by the kernel when any system calls are made.   
  
filesystem(): The constructor for filesystem.java. Sets the variables that will be used in this class like the superblock, filetable, and directory. Also includes code to read the specified file from the disk.  
  
sync(): The sync function, as the name would suggest, syncs up the filesystem metadata, superblock data, and directory data to the disk.**

**format(): This method formats the disk by erasing the disk and creating a new version of the directory and then a new version of the filetables for that directory.**

**open(): This method opens the file that was passed into it and then allocates it a file table entry in the file table. It also checks if the program is in write mode. If it is in write mode, it will delete all of the blocks.**

**close(): Boolean function that searches the file table and closes the appropriate file. If it is the final thread accessing the entry, then it will free it from the file table. Returns true id the operation was successful. Otherwise returns false.**

**read(): Reads the file from the file table entry and returns the amount of bytes that it read. Reads a buffer-size amount of data in each loop.**

**write(): Writes the buffer into the file that is supplied by entry. The seek pointer will tell the program where to start writing in the file.**

**seek(): Used to update the location of the seek pointer based off of the value that is passed into the method. Performs some error-checking in case the user tries to set the value to a negative number.**

**deallocAllBlocks(): Deallocates all of the blocks from the inode. On completion, will write the inodes back to the disk.**

**delete(): Deletes a file based off of the name that was passed into the function.**

**fsize(): Returns the size of the file (in bytes)**

**Kernel.java  
The Kernel method handles all of the system calls that are made through Syslib. Handles the following cases:**

**Open: Opens the file based off the file name that is passed in.**

**Close: Closes the file based off the file descriptor.**

**Size: Returns the file size based off the file descriptor.**

**Seek: Uses the file descriptor to update the seek pointer.**

**Format: Clears the disk for all files that are given.**

**Delete: Deletes the file based off of the string that is passed in.**

**Read: Reads the file from the user file descriptor table.**

**Write: Writes the file from the user file descriptor table.**

**Sync: Syncs the memory data to disk**

**Superblock.java  
The superblock is a block of metadata that is used to describe the file system that we’ve built in this project. It is managed by the operating system and manages the list of free blocks. The following functions are contained in superblock.java:**

**superblock(): Constructor for superblock. Is passed an int value that represents the number of blocks on the disk. The constructor will read the superblock from the disk and initialize the number of blocks and member variables.**

**Sync(): Syncs the physical superblock with the with the superblock class instance by writing totalBlocks, totalInodes, and the freeList back onto the Disk**

**getFreeBlock(): searches the freeList to find the next block ID of the next free block.**

**returnBlock(): Adds a block to the end of the freeList using the block ID that was given. This is done as soon as a block is freed.**

**format(): formats the disk by deleting all data and then reloading default variables.**

**Inode.java()  
Shows where file contents are located on the disk. Contains 12 pointers to the index block. 11 of these pointers are direct and one is indirect. The Inode.java class contains the following methods:**

**Inode(): Default constructor. Initializes variables and pointers.**

**Inode(short iNumber): Retrieves specific inode from the disk. First figures out the number of blocks that are needed and then reads that amount of blocks to the disk.**

**toDisk(): Saves the iNode data (count, ﬂag, direct[], and indirect) to the disk.**

**submitBlock(int pointer, short freeBlock): Iterates through all pointers on the disk and will read the data of all valid pointers.**

**setIndexBlock(): Writes data to the block that was passed into the method.**

**findBlock(): Returns the target block of the direct or indirect pointer (depending on the size of the target). Writes data to the specified blockspace.**

**freeIndirect (): If the indirect block exists, it will read data from it and return an array of the data. Otherwise will return NULL.**

**Directory.java  
The Directory.java class is responsible for managing the files that are used in this file system. Each file in the system has its own entry in the directory and an inode number. Because of this we can keep track of which files are currently being used by the file system. The directory is initialized with an array of a certain size. This is the maximum number of files that the system can contain since the array does not grow dynamically. The Directory.java class contains the following functions.**

**Directory(): The constructor for the Directory class. A value is passed in which is used as the size of the array of file names.**

**Bytes2directory(): Converts all of the byte array data to directory data.**

**Directory2bytes(): Uses the int2bytes method contained in the SysLib class to convert directory data into a byte array**

**Ialloc(): Allocates an inode for a file. To do this the method finds an empty file and then iterates through the directory to see if there are any files with a size of 0. Once it has been found it will be added to the filenames array.**

**Ifree(): This method frees up an inode by finding the iNumber in the file size array and then setting it to 0.**

**Namei(): The namei() method will search through the file sizes array until it finds a match. Once a match has been located, it will return the index value for that file.**

**FileTable.java   
FileTable.java is a class that is shared among all user threads. It is used when a user thread tried to open a new file. First the user thread will create a new entry in the file descriptor table in its thread control block. Then, the user thread tells the file system to create a new entry in the file table. Next, the file system saves the inode from the directory. Finally, the user thread will register a reference to the newly created file table entry in the file descriptor table entry of the thread control block.**

**FileTable(): This method will create the FileTable entry and see the directory to the directory that was passed in.**

**Falloc(): The falloc method creates a file table entry for the file that was passed into the function. Also creates files when a file that does not exist is opened for write mode.**

**Ffree(): Saves the inode to the disk and releases the filetable entry. Returns false if no entry was found.**

**Fempty(): Checks if the table is empty. Returns true if it is. This method should be used before any formats take place.**

**FileTableEntry.java  
FileTableEntry.java keeps track of entry data for the file structure table. This data includes seekPtr, inode, iNumber, count, and mode.**

**FileTableEntry(): Constructor. Sets the inode, inumber, and mode based off of the values that were passed in. SeekPtr is set to 0 and count is set to 1.**

**TCB.java  
Thread control block. Maintains a user file descriptor table for all the files have been opened. Every time it opens a file, it allocates a new entry table including a reference to the corresponding file (structure) table entry. Whenever a thread spawns a new child thread, it passes a copy of its TCB to this child which thus has a copy of its parent's user file descriptor table.**

**TCB(): Constructor. Sets the thread, tid, pid, and terminate values.**

**getThread(): Returns the thread.**

**getTid(): Returns the tid.**

**getPid(): Returns the pid.**

**setTerminated(): sets terminated value to true.**

**getTerminated(): returns the terminated Boolean value.**

**getFD(): Adds entry to the file descriptor table and returns the index of that entry.**

**returnFD(): Returns the file table entry, and then sets its index to NULL.**

**getFtEnt(): Returns the file table entry (does not set index to NULL)**

**Performance Estimation & Functionality  
For this project, we used the .class files that were provided in ThreadOS as a benchmark for what our files should be able to achieve performance-wise. After testing we found that both our files and the .class files were similar in terms of performance. We noticed that formatting the disk and creating large amounts of files would generally take the longest amount of time. This is because both the disk raw read necessary for formatting and the amount of Disk I/O required to create new files both take a while to complete.   
  
A possible solution for the disk I/O performance problem would be to use a cache. By doing this, the filesystem will be able to store frequently used files and would not have to look for every file off the disk. This will also save time since the inode will not have to write to the disk every time a change has been made.  
  
One limitation in the way we calculated performance is that we only calculated performance by running the programs and observing which sections took the longest to complete. We neglected to put any timing mechanisms into the code, so it is impossible to get an exact representation of the performance of this file system.  
  
The current functionality of this file system is fine for the small scale that is being tested with, but if it was to be used as a real file system, there would be many problems that would need to be solved. The limitations discussed above, the number of inodes allowed on the disk, the amount of direct and indirect pointers, the maximum number of blocks on each disk, and many other issues with implementation would have to be drastically changed to have this file system be functional in a typical operating system.**

**Contribution  
Diego Guzman: 33%**Misha Ward: 34%  
Nathanael Fixx: 33%