Mustafa Majeed & Cody Rhee

6/9/19

CSS 370

P5 Report

**Final Project: File Systems**

**FileSystem.java Specification**

This class is written to be the File System of ThreadOS. It will be used to keep track the SuperBlock, Directory and all created files for this Volume. With this class, it provides user threads with system calls that allows them to format, to open, to close, to format, to write to, to read from, to update the seek pointer, to delete, and to get the size of the files. Below are the list of functions that are provided in this class:

* **void sync()**: this method is used to write the directory and the superblock to DISK
* **boolean format( int )**: this method is used to re format the FileSystem using the passed in int value to represent the max number of files that this Volume will contain.
* **boolean createInodes( int )**: this method is used to create and write the Inodes for this volume to the DISK based on the passed in int files that will signify the max number of files for this Volume
* **FileTableEntry open( String, String )**: this method is used to open a file and add it to the FileTable if does not already exist. It will return the correct pointer to the FileTableEntry if the file is already in the FileTable.
* **boolean close( FileTableEntry )**: this method is used to close a file in the passed in FileTableEntry. This is accomplished by decrementing the FileTableEntry count first and then checking if the count is 0 to determine if the entry needs to be removed from the FileTable or if there are other threads currently in this file and thus keeping it in the file table.
* **int read( FileTableEntry, byte[] )**: this method is used to read the contents of a file to the passed in byte[] buf. Will return the number of bytes read from the file.
* **int fsize( FileTableEntry )**: this method returns the file size
* **int write( FileTableEntry, byte[] )**: this method writes in the content in the byte[] buf passed in the argument to the ftEnt also passed in the argument. Method may overwrite or append on the existing data in the file, depending on where the seek pointer of file is. Method returns the number of bytes that have been written.
* **boolean deallocAllBlocks( FileTableEntry )**: this method will zero out all blocks associated with the FileTableEntry that is passed in and reset Inode values to default
* **boolean delete( String )**: this method deletes the file name passed in the argument. All blocks used by the file are freed. In the case that the file is currently open, instead of deleting the file, the method returns false, indicating the file was not deleted. If the file was deleted successfully, the method returns true.
* **int seek( FileTableEntry, int, int )**: this method updates the seekpointer of the passed in file with the given set from the beginning of the file, current seek pointer value, or the end of the file

**Description of Internal Design**

*Directory.java*

The purpose of this class is to maintain different files. The Directory.java file receives the total number of inodes (files) to be created and keeps track of which inode numbers are in use. Below are the functions that are present in the Directory.java file:

* **void bytes2directory( byte[] )**: this method assumes data[] received directory information from disk initializes the Directory instance with this data[]
* **byte[] directory2bytes()**: this method converts and return Directory information into a plain byte array this byte array will be written back to disk. Only meaningful directory information should be converted into bytes.
* **boolean ialloc( String, int)**: this method allocates a new inode number for the String filename that is passed in as a parameter
* **boolean ifree( short )**: this method deallocates the inode number passed in the parameter (inode number) and the corresponding file will be deleted.
* **short namei( String)**: this method returns the inumber corresponding to the String filename passed in the parameter
* **int freeSpot()**: this method finds the first index in the file size array that is free

*FileTable.java*

This class is written to represent the system wide open FileTable for this volume. The main job of this class is to create new file table entries, add them to the Vector in the class that holds file table entries, and remove any file table entries from the Vector when it is necessary. Below are the list of functions within the class:

* **FileTableEntry falloc( String, String )**: allocates a new file (structure) table entry for this file name. Method allocates/retrieves and registers the corresponding inode using dir. Inside the method, it also increment this inode's count and immediately writes back this inode to the disk. It returns a reference to this file (structure) table entry.
* **boolean freeEntry( FileTableEntry)**: this method will remove the passed in FileTableEntry from the FileTable and save the Inode to DISK
* **boolean fempty()**: this method checks if the Vector table that holds files is empty

*FileTableEntry.java*

This class is written to represent entries in a FileTable. Each entry will contain a seekPtr int, an Inode pointer, an iNumber short, a count int (# of threads sharing this entry), and a mode String.

* **int fileSize()**: will return the length of the Inode in this FileTableEntry. This number represents the size of the file

*Inode.java*

This class is written to represent Inodes for each file in the FileSystem. Each file gets 1 Inode that will track the size of the file and pointers to the disk blocks that the file contents reside in.

* **void toDisk( short )**: this method is used to write the calling Inode to DISK at the passed in iNumber position in Inode list.
* **int findIndirectBlock()**: this method is used to write the calling Inode to DISK at the passed in iNumber position in Inode list.
* **boolean registerIndexBlock( short )**: this method is used to allocate the passed in block number to the indirect pointer
* **int findTargetBlock( int )**: this method is used to find the DISK block number of a file using the passed in int value for the byte of data to find in that file.
* **int registerTargetBlock( int, short )**: this method is used to add the passed in bNum to the list of indirect block pointers for this Inode at the specified position in the file
* **byte[] getIndirectIndexBlock()**: this method will return the contents of the DISK block that is being pointed to by the indirect pointer for this Inode

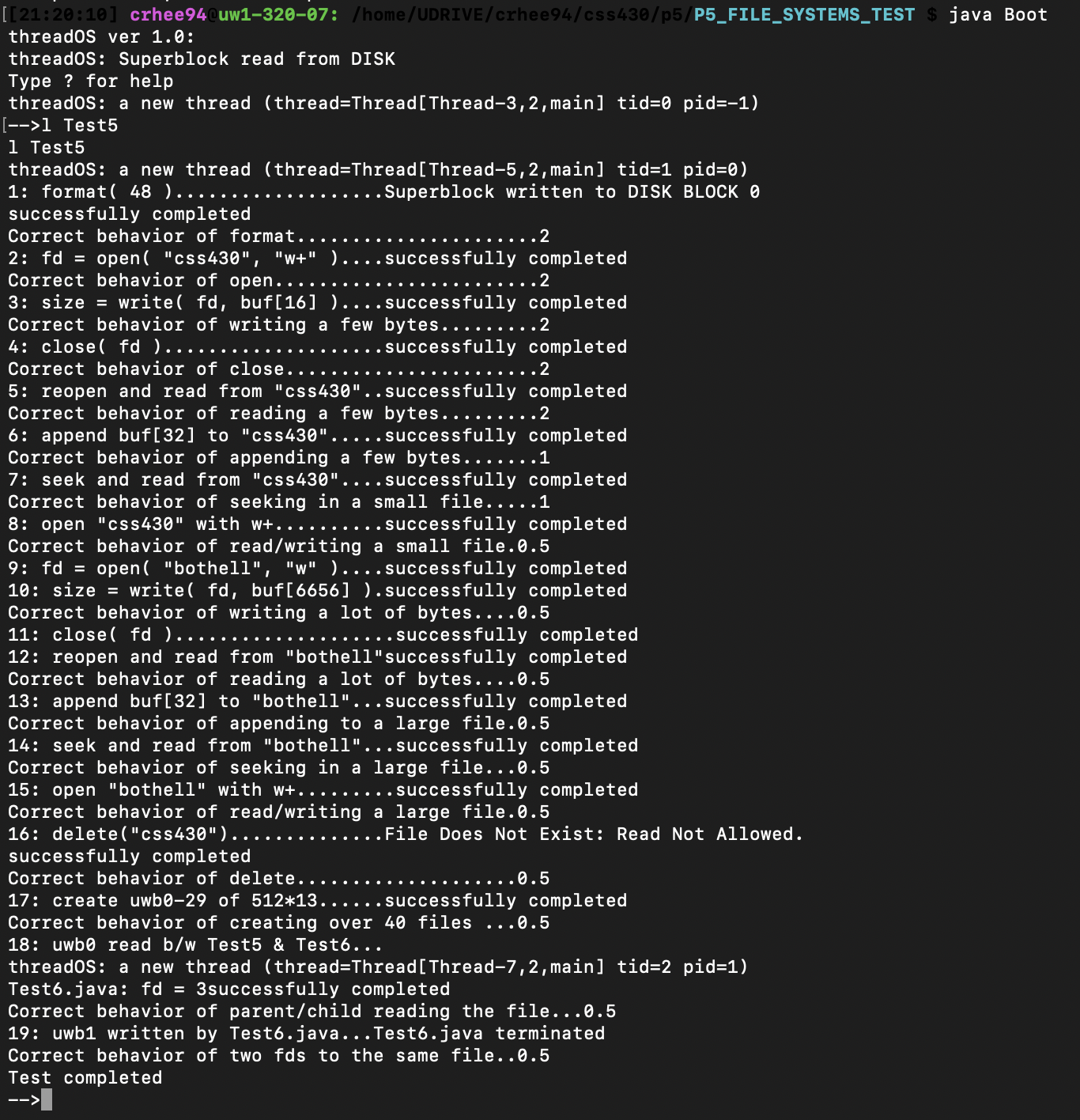
*SuperBlock.java*

This class is written to represent the SuperBlock for a File System on disk block 0. The SuperBlock stores the total number of blocks on this DISK, the total number of Inodes that will be used for this Volume and the head of the free block list.

* **boolean format( int )**: this method will re format the Superblock to keep track of the passed in max number of files. It will reset the totalInodes and freeList  data members to match the values needed for the passed in number of files.
* **void sync()**: this method will write the contents of this Superblock to the DISK
* **int getFreeBlock()**: will return the int value of a free block that is not being used by any file on the DISK.
* **boolean returnBlock()**: this method is used to return a block to the head of the freeList

**Test Results**

The following Figure is a screenshot of our final test using the provided Test5.class and Test6.class.

Figure 1: Test Results with our FileSystem

Based on the test results above, our FileSystem passed all of the provided test cases. This package also include an output.txt file that was generated from a script inside of the linux shell while running Test5.class and Test6.class on our FileSystem.

**Limitations and Assumptions**

*Key Limitations*

* Disk Size
* Block Size
* Only 1 indirect pointer
* Max File Size

The threadOS DISK block only contains 1000 blocks of 512B of data. Each Volume on a DISK requires a SuperBlock, Inode blocks and Directory blocks.

For threadOS, the max number of files is 64. This means that block 0 will be taken up the Superblock, blocks 1,2,3, and 4 will hold Inode information (Assuming that 1 block will hold 16 32B Indoes) and blocks 5,6,7,8,9,10,11, and 12 will be used to store Directory information.

This leaves only 987 DISK blocks for files to make use of.

Furthermore, each Inode used to track a file has 11 direct pointer and 1 indirect pointer to blocks on disk to be used by the file. The indirect pointer can house 256 block numbers. This leads to the following assumption about max file size in this FileSystem:

256 + 11 = 267 Max Blocks per file

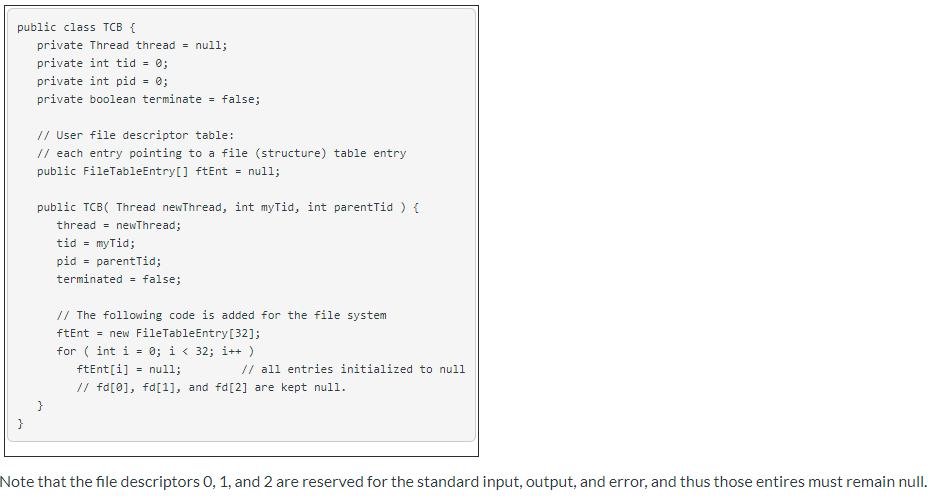
**267 \* 512B = 136.70400KB MAX FILE SIZE**

*For comparison*

The typical encoding standard for MP3 files is 128 kilobits per second (kbps, kb/s or kbit/s). This works out to about 1 megabyte (MB) per minute of sound. 128 kilobits is equal to 16KB. ~137KB/16KB = ~8.5. That means the maximum mp3 file size we can store in this FileSystem would only contain about 8.5 seconds of audio. This is not acceptable in any modern System. But it was a great way to demonstrate how FileSystems manage files.

Some assumptions that we made while coding our FileSystem is that the Disk object (used to simulate a memory DISK) was always created prior to the creation of our FileSystem. We also assumed that no cache system was included, so each read and write was done directly to or directly from DISK.

We also are under the assumption that the following code has been added to TCB.class



**Performance Estimation and Functionality**

For the performance of this program, we expected that the result would be like the result we saw when we tested the stock ThreadOS Test5.java. The performance was expected to be very slow because, in the FileSystem.java, the method to format the file system took exceptionally long because the method performed rawwrite links to each block on DISK. Another reason why expected the performance to be correct but slow is that in the read and write methods inside FileSystem.java, we were performing rawwrite to write to the DISK without using any things like buffer cache.

**Possible Extended Functionality**

*Functionalities*

* Increase block size or Disk size
* Buffer Cache
* Add L2 or L3 indirect pointers to Inodes

Because of the limitations mentioned in the *Limitations and Assumptions* section above, the first functionality that could be added to this System to for improved performance could be to increase the block size or Disk size. This would allow for larger files to be stored in the System.

Another possible functionality that can be added to improve performance is the inclusion of the buffer cache system that we implemented in P4. This can greatly increase reading from a file or writing to a file if we could bring in contiguous blocks of a file to a buffer cache in anticipation of contiguous user access. Cache access is much faster than DISK access and it would greatly improve performance.

Adding more indirect pointers can also improve the functionality of the FileSystem. Along with adding another level of indirect pointers comes the necessity for bigger disk blocks or a bigger DISK in general, but more indirect pointers allow for larger max file size. One of the biggest limitation to this System is the small max file size as it is currently constructed. This addition would help solve that issue.