P5: ThreadOS File System Report

Daniel Yan

Emily Krasser

CSS 430

Spring 2019

BUILD CODE:

To properly run the File System within ThreadOS, the following steps must be performed:

1. Fresh copy of ThreadOS is obtained
2. All modified code files that are submitted overwrite present ThreadOS Files
3. Recompilation of all modified code files is performed as specified in Compile Command

Compile Command:

```
javac Kernel.java Scheduler.java SysLib.java TCB.java
Inode.java FileTable.java FileTableEntry.java
SuperBlock.java Directory.java FileSystem.java
```

The following command can be run in UWB Linux Labs to obtain a fresh copy of ThreadOS to the current local directory:

```
cp -r /usr/apps/CSS430/ThreadOS .
```

Table of Contents

BUILD CODE: 2

Compile Command: 2

Table of Contents 3

Discussion: 5

Specifications: 6

Assumptions: 6

Limitations: 6

Descriptions: 7

Inode: 7

TCB: 7

File Table: 7

File Table Entry: 7

Directory: 7

Superblock: 7

File System: 8

Class Internal Designs 9

Inode: 9

Attributes 9

Methods 10

TCB: 11

Attributes 11

Methods 11

File Table: 13

Attributes 13

Methods 13

File Table Entry: 14

Attributes 14

Methods 14

Directory: 14

Attributes 14

Methods 15

Superblock: 15

Attributes 15

Methods 16

File System: 17
Attributes 17
Methods 17
Results: 19
Future Implementation Considerations 20
Performance Estimation 20
Current Functionality 20
Possible Extended Functionality 21

Discussion:

Discussion answers and analysis for Program 5 are listed in this section.

1. Does the filesystem allow one to read the same data from a file that was previously written?

    - FileSystem does allow one to read the same data from a file that was previously written to by saving to the disk every time that an Inode is changed. This makes sure that when you read from a file that had been written to, you are reading the updated file and not the original version.

2. Are malicious operations handled appropriately and proper error codes returned? Those operations include file accesses with a negative seek pointer, too many files to be opened at a time per a thread and over the system, etc.

    - Malicious operations are handled via case handling and clamping code. Potential malicious operations such as negative seek pointers are handled in the seek and write function so that error outputs are produced in those cases. Synchronized blocks handle the potential of having too many files open at the same time/per thread. Case handling can be primarily seen in methods such as seek() or delete() in the FileSystem.

3. Does the file system synchronize all data on memory with disk before a shutdown and reload those data from the disk upon the next boot?

    - The synchronizing function in FileSystem makes sure to synchronize everything in memory to the disk. It then calls the synchronize function in the SuperBlock which gets all of the data in bytes and rawwrites it to the disk.

Specifications:

Assumptions and limitations for all classes that were modified or created for this Program 5 File System in ThreadOS are listed in this section.

Assumptions:

We made our assumptions according to the specifications that the assignment provided. This included the powerpoint slides, examples and provided files, and the FAQ. These assumptions included the idea that the specifications provided all of the functionality that the users would need. As well, we developed under the assumption that the commands generated by users to access files would be mostly legitimate, thus requiring limited validation.
Other than the few extra classes that we created, we assumed that the functions specified by the assignment were all that would be needed by users and/or tests.

Limitations:

There were a few limitations to our file system.

- One of them was that there was no usage of caching algorithms for keeping Inodes in memory and using cached methodologies, such as Enhanced Second Chance. Rather, the Inodes write back to disk each time they are changed, reducing efficiency.
- Another limitation was that there was a limit of up to 64 Inodes per disk (per the specifications). This is much unlike a real file system, where thousands upon thousands of files can be handled in a dynamic matter.
- A limitation that we encountered when working with ThreadOS itself, was that a file name can only be 30 characters long.
- An issue and limitation was that we encountered, was that there were only 11 direct and only 1 indirect pointers that were present in each Inode. This solution is reasonable if the files don’t exceed more that 11 blocks total. If the file does exceed 11 blocks, the indirect pointer is then used, causing access time to rise.
- The last limitation was that there is no protection and no permissions required to access/use the file system. Using more encapsulated programming principles would aid in this since there are many instances where other methods could directly access or change values in Inode.

Descriptions:

Descriptions for all classes that were modified or created for this Program 5 File System in ThreadOS are listed in this section.

Inode:

The Inode class is used for keep track of items for FileSystem. It deals with file blocks and their management for their respective files and holds the file size, the number of file table entries pointed to, the Inode’s flags, as well as direct and indirect pointers. There is a limit of 11 direct pointers in an Inode. When all 11 are used, the indirect pointer is then used.

TCB:

TCB was modified for P5 in order to work with the FileSystem. Each user thread maintains a user file descriptor table in its own TCB

File Table:

Maintains the file table shared among all user threads.

File Table Entry:

FileTableEntries to use in FileTables.

Directory:

Implements Unix-like directory structure for ThreadOS File System. The root directory maintains each file in a different directory entry that contains its file name and the corresponding inode number. The directory receives the maximum number of inodes to be created and keeps track of which inode numbers are in use. Since the directory itself is considered a file, its contents are maintained by an inode, specifically inode 0. This can be located in the first 32 bytes of the disk block 1.

Superblock:

SuperBlock which refers to the 0th block of the DISK and is used to describe the number of disk blocks, the number of Inodes, and the block number of the head block of the free list. It also synchronizes to the disk.

File System:

A Unix like file system on ThreadOS that allows the user programs to access persistent data on disk by way of stream-oriented files rather than direct access to disk blocks with rawread and rawwrite.

Is responsible for performing all operations on the disk through implementations of all basic file system functionality including: open, write, read, delete, seek, format, and close. Some functionality is aliased to be similar to other more common names such as create and find which Windows File Systems primarily use. The File System initializes the other class objects which are utilized.

Class Internal Designs

Implementation details for all classes that were modified or created for this Program 5 File System in ThreadOS are listed in this section, primarily including their attributes and methods.

Inode:

Attributes

Names	Descriptions
INODE_SIZE : static final int	Size of inode at 32 bytes
DIRECT_SIZE : static final int	Number of direct pointers at 11
BUF_SIZE : static final int	Size of byte[] buffer for data blocks
SHORT_BYTE_SIZE : static final int	Size of short in bytes, at 2 bytes
INT_BYTE_SIZE : static final int	Size of int in bytes, at 4 bytes
OFFSET_DISK : static final int	Offset of 1 for disk
DISK_BLOCK : static final int	Size of DISK_BLOCK at 16 bytes
SUCCESS : static final int	Status code of 0 to indicate status
ERR_BLOCK_REG : static final int	Status code of -1 to indicate status
ERR_BLOCK_UNUSED : static final int	Status code of -2 to indicate status
ERR_INDIRECT_NULL : static final int	Status code of -3 to indicate status
length : int	File size in bytes
count : short	Number of file entries pointing to
flag : short	Flag status, 0 = unused, 1 = used
direct : short[DIRECT_SIZE]	Number of direct pointers
indirect : short	Single indirect pointer

Methods

Names	Descriptions
Inode()	Constructor that simply initializes the variable members and sets indirect and direct pointers
Inode(short)	Constructor that performs the following steps 1. Gets Inode number from input 2. Determine disk block from Inode number 3. Make a new byte[] buffer to read Disk block 4. Use raw read to read data block in buffer 5. Get the specific Inode from the buffer 6. Retrieve Inode information and initialize member values
toDisk(short) : void	Writes Inode to DISK based on an inputted index for Inode after finding the Inode using raw read and using raw write to save it to disk
registerIndexBlock(short) : boolean	Checks if a block number is valid to register based on an indirect pointer after doing validity checks on indirect and direct pointers
findTargetBlock(int) : int	Finds a specific Inode block based on the passed in index
registerTargetBlock(int, short) : int	Registers an Inode block based on a passed in index after doing validity checks on indirect and direct pointers
unregisterIndexBlock() : byte[]	Frees up an Inode block after checking indirect pointer
findIndexBlock() : int	Finds a Inode block based on indirect

TCB:

Attributes

Names	Description
thread : Thread	Object to hold a thread
tid : int	Int to hold the thread id
pid : int	Int to hold the process id
terminated : boolean	Status of the thread
sleepTime : int	Time to sleep for the thread
ftEnt : FileTableEntry[]	Array of FileTable Entries

Methods

Names	Description
TCB(Thread ,int, int)	Constructor to initialize the Thread object based on passed in ints, where the first is the current tid of the Thread, and the second is the tid of the parent Thread.
synchronized getThread() : Thread	Gets the current Thread
synchronized getTid() : int	Gets the current thread id of Thread
synchronized getPid() : int	Gets the current process id
synchronized getTerminated() : boolean	Gets the current terminated state of Thread
synchronized setTerminated() : boolean	Sets the current terminated state of Thread
synchronized getFd() : int	Gets a file descriptor based on a FileTableEntry if the entered entry is not null
synchronized returnFD(int) : FileTableEntry	Return a FileTableEntry and set specified file descriptor in the entry to null
synchronized getFtEnt(int) : FileTableEntry	Returns the specified FileTableEntry based

	on passed in file descriptor index

File Table:

Attributes

Names	Descriptions
FLAG_UNUSED : static final int	Inode status flag for unused state
FLAG_USED : static final int	Inode status flag for used state
FLAG_READ : static final int	Inode status flag for read state
FLAG_WRITE : static final int	Inode status flag for write state
FLAG_TO_DELETE : static final int	Inode status flag for pending delete state
table : Vector	Entity for file table
dir : Directory	Root directory

Methods

Names	Descriptions
FileTable(Directory)	Constructor that initializes a Directory value
synchronized falloc (String,String) : FileTableEntry	Allocates a new FileTableEntry based on a passed in filename. Has different actions based on mode of operation passed in ● Allocate/retrieve and register the corresponding inode using dir ● Increment this Inode's count ● Immediately write back this inode to the disk ● Return a reference to this file (structure) table entry
synchronized ffree(FileTableEntry) : boolean	Attempts to free a FileTableEntry with the following steps: ● Receive a file table entry reference ● Save the corresponding inode to the

	disk ● Rree this file table entry. ● Return true if this file table entry found in my table
synchronized fempty() : boolean	Returns true if the FileTable is empty, called before format()

File Table Entry:

Attributes

Name	Description
seekPtr : int	File seek pointer
inode : Inode	Inode reference
iNumber : short	Inode number reference
count : int	Number of threads sharing this entry
mode : String	Available modes for entry

Methods

Name	Description
FileTableEntry(Inode,short,String)	Constructor which instantiates a FileTableEntry with a reference to Inode, Inode number, and the String mode

Directory:

Attributes

Names	Descriptions
maxChars : static int	Max number of characters for files, at 30
ERROR : final static int	Error status code for return
fsizes : int[]	Directory file size entries
fnames : char[][]	Directory filename entries

Methods

Names	Descriptions
Directory(int)	Constructor that initializes the file names, file sizes, and root path member variables
bytes2directory (byte[]) : void	Assumes data[] received directory information from disk and initializes the Directory instance with this data[]
directory2bytes() : byte[]	Converts and return Directory information into a plain byte array which will be written back to disk. Only meaningful directory information is be converted into bytes.
ialloc(String) : short	Uses the inputted String filename to mark it as one of a file to be created. Allocates a new inode number for this filename
ifree(short) : boolean	Deallocates the iNumber that’s passed in (inode number) and the corresponding file will be deleted
namei(String) : short	Returns this file's iNumber based on inputted filename

Superblock:

Attributes

Names	Descriptions
defaultInodeBlocks : final int	Default number of inode blocks
INODESIZE : static final int	Fixed size of inode bytes at 32
BUF_SIZE : static final int	Size for buffer byte[] for disk
totalBlocks : int	Total number of disk blocks
inodeBlocks : int	Number of inode blocks
freeList : int	Block number of free list head

Methods

Names	Descriptions
SuperBlock(int)	Constructor that initializes a SuperBlock based on an inputted disk size
sync() : void	Syncs the disk by writing back the total number of blocks, Inode blocks, and free list to the disk
format(int) : void	Decides the number of created files through the inputted int for formatting
getFreeBlock() : int	Gets the index of the inputted block ID for the next free block in the free list
returnBlock(int) : boolean	Places the index of the inputted block Id into the free list

File System:

Attributes

Name	Description
superblock : SuperBlock	Object for SuperBlock
directory : Directory	Object for Directory
filetable : FileTable	Object for FileTable
SEEK_SET : final int	Seek status for set
SEEK_CUR : final int	Seek status for current
SEEK_END : final int	Seek status for end
BUF_SIZE : static final int	Buffer for byte[] on disk
DIRECT_SIZE : static final int	Number of direct pointers
ERROR : static final int	Error status code

Methods

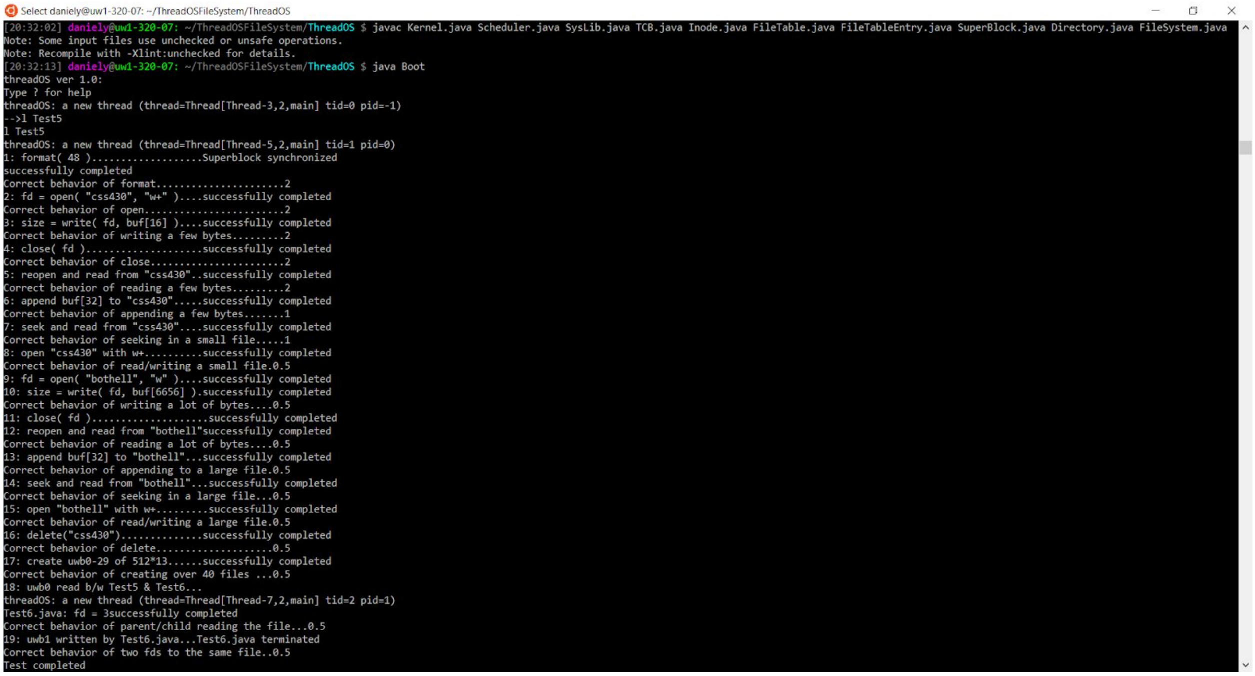
Name	Description
FileSystem(int)	Constructor which initializes the FileSystem member variables and sets the size of the disk blocks for Super Block
sync() : void	Syncs the file system back to the physical dis, by writing the directory information to the disk and ensuring the Super Block is synced
format(int) : boolean	Formats the physical disk by erasing all original content and regenerating the Super Block, Directory, and File Tables. The number of files created is used in the input
open(String, String) : FileEntryTable	Opens the file specified by the fileName string in the given mode (where "r" = ready only, "w" = write only, "w+" = read/write, "a" = append). The call allocates a new file descriptor, fd to this file. The file is created if it

	does not exist in the mode "w", "w+" or "a". SysLib.open must return a negative number as an error value if the file does not exist in the mode "r". Note that the file descriptors 0, 1, and 2 are reserved as the standard input, output, and error, and therefore a newly opened file must receive a new descriptor numbered in the range between 3 and 31. If the calling thread's user file descriptor table is full, SysLib.open should return an error value. The seek pointer is initialized to zero in the mode "r", "w", and "w+", whereas initialized at the end of the file in the mode "a".
close(FileEntryTable) : boolean	Closes the file corresponding to fd, commits all file transactions on this file, and unregisters fd from the user file descriptor table of the calling thread's TCB. The return value is 0 in success, otherwise -1.
fsize(FileEntryTable) : int	Returns the size of the file as indicated by the file descriptor in the inputted FileEntryTable
read(FileEntryTable, byte[]) : int	Reads up to buffer.length bytes from the file indicated by fd, starting at the position currently pointed to by the seek pointer. If bytes remaining between the current seek pointer and the end of file are less than buffer length, a SysLib.read is then performed to read as many bytes as possible, putting them into the beginning of buffer. It then increments the seek pointer by the number of bytes to have been read. The return value is the number of bytes that have been read, or a negative value upon an error.
write(FileEntryTable, byte[]) : int	Writes the contents of buffer to the file indicated by fd, starting at the position indicated by the seek pointer. The operation may overwrite existing data in the file and/or append to the end of the file. A SysLib.write operation increments the seek pointer by the number of bytes to have been written. The return value is the number of bytes that have been written, or a negative value upon an error.

deallocAllBlocks(FileTableEntry) : boolean	Deallocates all blocks by checking if Inode blocks are valid or not. Then goes through direct pointers and Super Block for validity checks. A final write to disk is then performed
delete(String) : boolean	Deletes the file as specified by the inputted String. Checks if the file is currently open or not, which if so, will fail until the last open file is closed.
seek(FileTableEntry, int , int) : int	Updates the seek pointer corresponding to the FileTableEntry that's inputted. The offset and whence input values are used to check cases for seek pointer validity. If the seek pointer is found to be below 0, it's clamped to be 0. If the seek pointer is found to be above file size, it's clamped to the end of the file

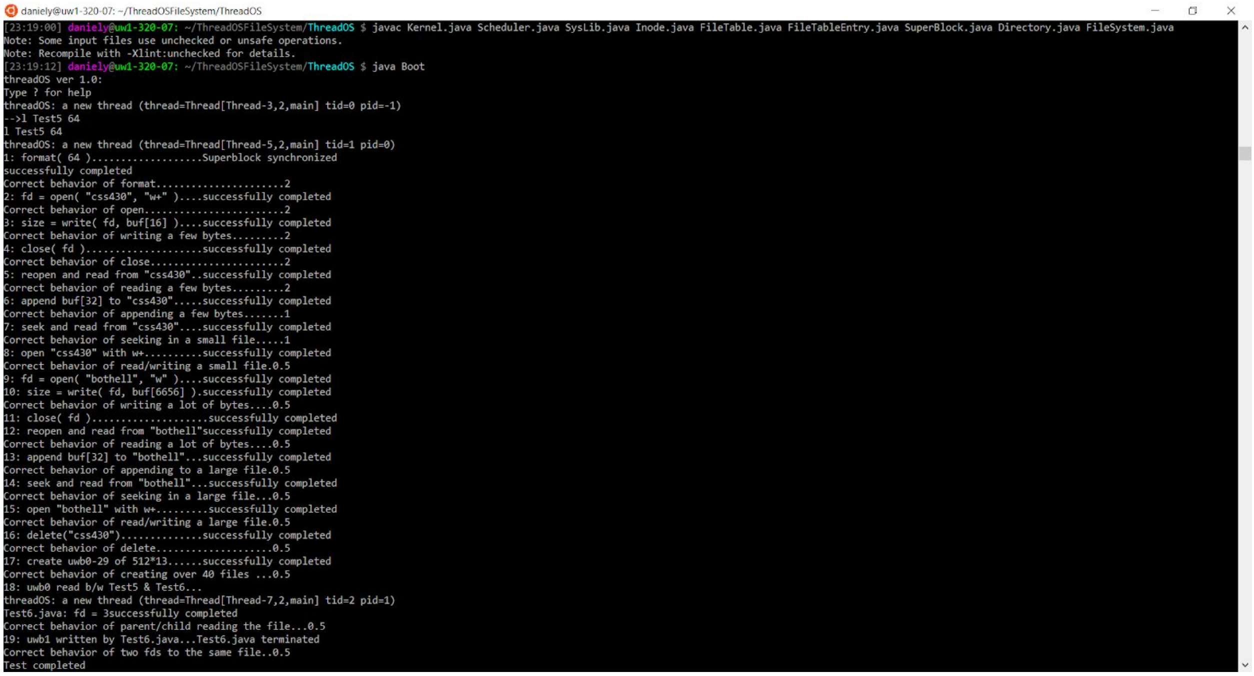
Results:

This section demonstrates successful performance testing based on Test5.java as detailed in the specifications of Program 5.



```
[20:32:02] daniely@uw1-320-07: ~/ThreadOSFileSystem/ThreadOS $ javac Kernel.java Scheduler.java SysLib.java TCB.java Inode.java FileTable.java FileTableEntry.java SuperBlock.java Directory.java FileSystem.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[20:32:13] daniely@uw1-320-07: ~/ThreadOSFileSystem/ThreadOS $ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test5
l 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")..............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
```

Figure 1: Test5.java using a format(48) for DISK on Linux Labs uw1-320-07.uwb.edu



```
[23:19:00] daniely@uw1-320-07: ~/ThreadOSFileSystem/ThreadOS $ javac Kernel.java Scheduler.java SysLib.java Inode.java FileTable.java FileTableEntry.java SuperBlock.java Directory.java FileSystem.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[23:19:12] daniely@uw1-320-07: ~/ThreadOSFileSystem/ThreadOS $ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test5 64
l Test5 64
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
1: format( 64 )...................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")..............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
```

Figure 1.1: Test5.java using a format(64) for DISK on Linux Labs uw1-320-07.uwb.edu

Future Implementation Considerations

This section includes future considerations as detailed in the specifications of Program 5.

Performance Estimation

Performance tests and estimations can all be quantitatively calculated based on the level of functionalities present within the system calls. The primarily test level was doing testing on Test5.java in the original provided ThreadOS for differing format numbers, primarily 48 (default) and 64 (max number). Our group felt that the provided Test5.java and all the underlying tests contained within was more than sufficient to test functionalities of all classes within The File System and majority of error case handling.

If more time was allocated, a performance comparison can be made by introducing the Cache.java using Caches to reduce I/O operation durations upon the disk. Comparisons of the time to perform all file operations could be done then using time calculations in microseconds inside of Test5.java to see speed differences in integrating a cache and without a cache.

Current Functionality

The current File System supports the following functionalities:

1. Support Multiple Files based on Disk Size Limits
    a. The DISK can be re-formatted to support multiple files
2. Save on Exit
    a. Upon exit, the File System can save files
3. Reload Data on Boot
    a. Upon boot, files can be loaded
4. Formatting Different Disks or Sizes
    a. A new DISK can be formatted based on DISK.java

Possible Extended Functionality

Additional functionality which could be added would include:

1. Extending the DISK and DISK block sizes
    a. DISK.java would be modified
2. Extending filename character limit
    a. Directory.java would be modified
3. Integrated Caching with the File System
    a. Cache.java would be added

4. Having multiple level of directories
    a. Directory.java would be modified
5. Having user permission protections
    a. File creations and access would be changed with ‘chmod’
6. Having additional file sizes available through additional pointers
    a. Inode.java would be modified
7. Having a better user interface or GUI to aid users
    a. A GUI.java would be added or FileSystem.java be modified
8. Having additional system calls be allowed for additional features
    a. SysLib.java, Kernel.java, FileSystem.java would be modified