Team (Group 6): Jacob J. Parkinson, Duke Dynda, Fuli Lan, Nicolas Koudsieh

UWB Su16 CSS430

Professor Michael Panitz

26 July 2016

File System Project Report

# Table of Contents

[Table of Contents 1](#_Toc458677793)

[General 2](#_Toc458677794)

[Debugging 3](#_Toc458677795)

[Output 4](#_Toc458677796)

[Detailed Class Specifications, Limitations, and Design Descriptions 4](#_Toc458677797)

[Directory 4](#_Toc458677798)

[Purpose 4](#_Toc458677799)

[Specifications 5](#_Toc458677800)

[FileTable 5](#_Toc458677801)

[Purpose 5](#_Toc458677802)

[Specifications 5](#_Toc458677803)

[FileTableEntry 6](#_Toc458677804)

[Purpose 6](#_Toc458677805)

[Specifications 6](#_Toc458677806)

[FileSystem 6](#_Toc458677807)

[Purpose 6](#_Toc458677808)

[Specifications 7](#_Toc458677809)

[Inode 8](#_Toc458677810)

[Purpose 8](#_Toc458677811)

[Specifications 8](#_Toc458677812)

[Kernel 8](#_Toc458677813)

[Purpose 8](#_Toc458677814)

[Superblock 9](#_Toc458677815)

[Purpose 9](#_Toc458677816)

[Specifications 9](#_Toc458677817)

[SysLib 9](#_Toc458677818)

[Purpose 9](#_Toc458677819)

[TCB 9](#_Toc458677820)

[Purpose 9](#_Toc458677821)

[Performance and Functionality 9](#_Toc458677822)

[Feedback for Instructor 10](#_Toc458677823)

[Jake Parkinson 10](#_Toc458677824)

[Duke Dynda 10](#_Toc458677825)

[Fuli Lan 10](#_Toc458677826)

[Nicolas Koudsieh 10](#_Toc458677827)

[Work Items 10](#_Toc458677828)

[References and Works consulted 12](#_Toc458677829)

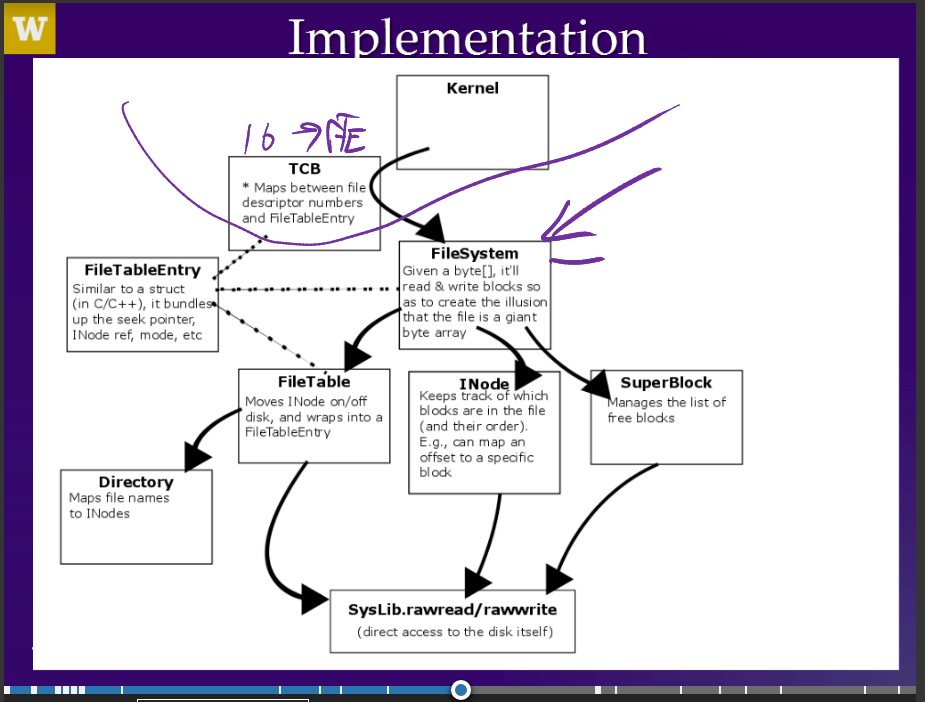
# 

# General

In order to write our code, some of us used Linux winSCP and some of us used IntelliJ. I (Jake) used Linux winSCP.

In order to ensure that we understood what we were implementing, we read about the various classes in the project, particularly the Inode, from the book and online, as well as utilized the videos.

We made our focus in this project oriented around the fact that the entire FileSystem is based on layered architecture, as shown here:



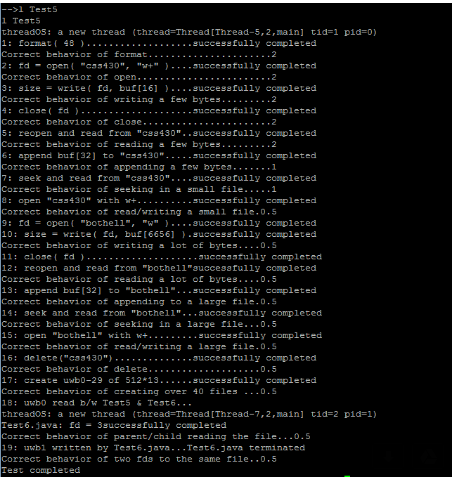
# Debugging

Our primary method of debugging depended on the individual person. I (Jake) used the provided .CLASS files to test individual .java files for bugs. When an individual file was working correctly with the provided .CLASS files from /usr/apps/CSS430/ThreadOS/, that meant that we had it working correctly, and could move on to the next. Because I used this manner of debugging our files, the smaller files like Directory.java, FileTable.java, and Superblock.java were completed first.

Our most difficult challenge was getting the read and write methods to correctly work. However, as we know, an important part of writing files is to modify the block map in the inode’s blocks (Rutgers). We accomplished this using helper methods modifyBlockMap, mapOffset (the primary function of the Inode according to the video), addIndex, and others.

We also encountered many smaller issues with individual methods such as the seek method not initially returning the seekPtr and forgetting the increment in the Inode(iNumber) constructor’s loop.

# Output



# Detailed Class Specifications, Limitations, and Design Descriptions

These specifications will be updated continuously as the project progresses. More detailed explanations in the .java files.

## Directory

### Purpose

* Map file names to inumber.
* Maintains a listing of each file that it exists in the root directory
* The directory can find a file if it exists in the directory and return the iNumber
* Keeps track of file size and name.
* Slot 0 -> inode 0 and so on.

### Specifications

* **Constructor:**

Gives a ‘blank’ directory object when called (**by thread**); provided by instructor

* **bytes2directory:**

**Thread calls this** to give the directory object some amount of bytes to be unpacked into a usable format (i.e., the format internal to directory object)

* **directory2bytes:**

**Thread calls this** if it needs to take the data internal to the directory object and pack it into a byte array to hand back as a return value

* **Ialloc:**

ask the directory to **create an inode** for that file and return the # of the inode created

* + If no inodes are available, return an error
* **Namei:**

Open a file which already exists, so call namei which checks if there’s a file with that name which has an inode.

* + If the file doesn’t exist, return an error (-1)
* **Ifree:**

Delete a file when done. Call and give it the inode to get rid of and return success Boolean.

* Max characters of filename must be 30 characters or less characters.

## FileTable

### Purpose

* The file system maintains the file (structure) table shared among all user threads.
* The user thread allocates a new entry of the user file descriptor table in its TCB. This entry number itself becomes a file descriptor number. The entry maintains a reference to a file (structure) table entry.
* The user thread then requests the file system to allocate a new entry of the system-maintained file (structure) table. This entry includes the seek pointer of this file, a reference to the inode corresponding to the file, the inode number, the count to maintain #threads sharing this file (structure) table, and the access mode. The seek pointer is set to the front or the tail of this file depending on the file access mode.
* The file system locates the corresponding *inode* and records it in this file (structure) table entry.
* The user thread finally registers a reference to this file (structure) table entry in its file descriptor table entry of the TCB.

### Specifications

* **FileTable(Directory directory**);

Instantiate a file table, receive a reference to the directory from file system.

* **void falloc(String filename, String mode);**

Allocate a new file (structure) table entry for this file name. 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.

* **boolean ffree(FileTableEntry e);**

Receive a file table entry reference. Save the corresponding inode to the disk. Free this file table entry. Return true if this file table entry found in my table.

* **boolean fempty();**

Return true if table is empty, false otherwise. Should be called before starting format.

## FileTableEntry

### Purpose

* See FileTable Purpose, specifically the file table entry portions.

### Specifications

* **FileTableEntry(Inode I, short inumber, String m);**

Seek pointer is set to the file top. At least one thread is using this entry. Once access mode is set, it never changes. If mode is append, seekPtr points to the end of file.

## FileSystem

### Purpose

* The file system should provide user threads with the system calls that will allow them to format, to open, to read from , to write to, to update the seek pointer of, to close, to delete, and to get the size of their files.
* For simplicity, the file system being created will consiste of a single level. The "/" root directory is predefined by the file system and permanently available for user threads to store their files. No other directories are provided by the system and created by users dynamically.
* Each user thread needs to keep track of all files it has opened. For this purpose, it should maintain a table of those open files in its TCB. This table is called a user *file descriptor table*. It has 32 entries. Whenever a thread opens a file, it must allocate to this file a new table entry, termed a *file descriptor*. Each file descriptor includes the file access mode and the reference to the corresponding file (structure) table entry. The file access mode indicates "read only", "write only", "read/write", or "append". The file (structure) table is a system-maintained table shared among all user threads, each entry of which maintains the seek pointer and the inode number of a file. Depending on the access mode, the seek pointer is set to the first or the tail of the file, and keeps track of a next position to read from and to write to the file. It is entirely possible for one thread to open the same file many times, thus having several entries in the corresponding TCB's user file descriptor table. Although each of these user file descriptor table entries refer to a different file (structure) table entry with its own seek pointer, all of them eventually points to the same inode.

### Specifications

* **int format( int files );**  
  Formats the disk (*Disk.java*'s data contents). The parameter *files* specifies the maximum number of files to be created (the number of inodes to be allocated) in your file system. The return value is 0 on success, otherwise -1.
* **int open( String fileName, String mode );**  
  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".
* **int read( int fd, byte buffer[] );**  
  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*,*SysLib.read* reads as many bytes as possible, putting them into the beginning of buffer. It 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.
* **int write( int fd, byte buffer[] );**  
  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. *SysLib.write*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.
* **int seek( int fd, int offset, int whence );**  
  Updates the seek pointer corresponding to *fd* as follows:
  + If *whence* is SEEK\_SET (= 0), the file's seek pointer is set to *offset* bytes from the beginning of the file
  + If *whence* is SEEK\_CUR (= 1), the file's seek pointer is set to its current value plus the *offset*. The *offset* can be positive or negative.
  + If *whence* is SEEK\_END (= 2), the file's seek pointer is set to the size of the file plus the *offset*. The *offset*can be positive or negative.

If the user attempts to set the seek pointer to a negative number you must clamp it to zero. If the user attempts to set the pointer to beyond the file size, you must set the seek pointer to the end of the file. Return the seekPtr at the end.

* **int close( int fd );**  
  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.
* **int delete( String fileName );**  
  Destroys the file specified by *fileName*. If the file is currently open, it is not destroyed until the last open on it is closed, but new attempts to open it will fail.
* **int fsize( int fd );**  
  Returns the size in bytes of the file indicated by *fd*.

## Inode

### Purpose

* The Inode keeps track of the blocks that are in the file and maps between an offset in a file to a block
* Keeps track of eleven direct blocks and a single indirect block.
* Each Inode has a flag (status), a count (# of entries pointing to that Inode), and a length (file size)
* Used to modify the block map and save blocks to the disk before discarding them

### Specifications

* **Constructors**

Has a default constructor, a 3-argument constructor called in the format() method, and a 1-argument constructor that uses a block index to retrieve a block from the disk and modify it (the way to do this is shown in the Test5.java code)

* **Mutators**

Has several mutators to preserve layered architecture as much as possible when multiple people develop a project.

* **toDisk**()

A method that takes an Inode number (corresponding to a block number, as always) and saves it to the disk.

* **addIndex**()

Adds a new index or slot for use in the Inode’s block map

* **modifyBlockMap**()

Modifies the block map of an Inode. Takes an offset value for the Inode and a block number and then fetches that block from the disk if it’s not already in the Inode structure. Note that it doesn’t strictly modify the block map beyond moving something into the indirect block.

* **mapOffset**()

Maps the offset of a file / Inode to a particular block index and returns that value.

## Kernel

### Purpose

* Implements many of the SysLib calls which were not implemented by the final project. Was not modified for this project beyond fixing a race condition by using the instructor’s modified version.

## Superblock

### Purpose

* Keeps track of the blocks on the disk, the Inodes, and the free block list.
* Reformat the disk with a new diskSize and # of Inodes

### Specifications

* **Constructor**

Sets up the superblock and its class members (based on Test5.java)

* **Format(int fileCount)**

Formats the drive to be accustomed to the new fileCount (with a default 1000 blocks)

* **getFreeBlock**()

Dequeues the block on the top of the free list and returns its index if available

* **returnBlock**()

Adds a block to the end of the queue for later use

* **Sync**()

Helper method called by FileSystem.sync(), it basically redoes a portion of the constructor’s code without the chance to reformat anything, and then updates the disk with the new data.

## SysLib

### Purpose

* A class provided for use with the ThreadOS Library, SysLib is used to simulate parts of the OS that we can’t typically call in Java while using multithreaded calls.
* Used to communicate between the disk and the other classes and for some conversions.
* Not modified for the program

## TCB

### Purpose

* A class provided for use in with the ThreadOS Library. Essentially keeps track of our file descriptors for the project.
* Not modified for the program

# Performance and Functionality

The performance of our final version is unfortunately very slow, although it will complete correctly. As stated in the project description, there are ways (which are complicated) in which to more frequently update the disk in order to improve overall performance; however, we did not focus on this. Therefore, we would hypothesize that this was the reason for our slow performance.

On the other hand, our code works together without any noticeable issues, therefore ensuring complete functionality. In conclusion, this was a difficult project and we should have started about a week earlier. Visualizing the project and having to interpret the purpose of the functions from the Syslib class so our main priority was just ensuring complete functionality and not focusing too much on extended functionality.

# Feedback for Instructor

## Jake Parkinson

Next time, don’t give students the Test5.java method to test our code, and make them write their own test file. This would give people a better feel for what the entire system is doing. It would be especially useful if it were possible to tie in some kind of visual representation of what’s going on (but that would probably not be feasible).

## Duke Dynda

Covering the file system in greater detail in class would be great. Explaining code snippets from the final project description and final project pdf etcetera would benefit. Understanding how each class of the file system is interrelated was a challenge. Drawing out the class hierarchy in class and explaining this in class would help clarify.

## Fuli Lan

I think for the future, the explaining should be more clearly and give more idea about how to write this program. For example, write some code in the beginning of each class. Also, the relational graph between each class provide will also help students to understand this project.

## Nicolas Koudsieh

For the future, I definitely recommend maybe creating more visuals of how the classes are interrelated (i.e. the file structure table, file table entries, and processes). Additionally, maybe some more visuals regarding allocation of blocks of data . One of the most difficult aspects of this project was simply trying to visualize everything and if students in the future had a better perspective of how the blocks of data were allocated and how all the classes are interrelated, they may find the project to be a little more feasible.

# Work Items

|  |  |  |
| --- | --- | --- |
| Item | Estimated Completion (%) | Contributors |
| Project Report | 100% | Jake |
|  |  |  |
| Directory.Directory | 100% | Provided |
| Directory.bytes2directory | 100% | Jake |
| Directory.directory2bytes | 100% | Fuli |
| Directory.ialloc | 100% | Fuli |
| Directory.ifree | 100% | Fuli |
| Directory.namei | 100% | Fuil |
|  |  |  |
| FileSystem.FileSystem | 100% | Provided |
| FileSystem.sync | 100% | Jake |
| FileSystem.format | 100% | Jake, Fuli, Duke |
| FileSystem.open | 100% | Duke |
| FileSystem.close | 100% | Provided |
| FileSystem.fsize | 100% | Duke |
| FileSystem.read | 100% | Jake, Nick, Fuli |
| FileSystem.write | 100% | Jake, Nick |
| FileSystem.deallocAllBlocks | 100% | Nick, Fuli |
| FileSystem.seek | 100% | Nick, Jake |
|  |  |  |
| FileTable.FileTable | 100% | Provided |
| FileTable.falloc | 100% | Duke, Nick |
| FileTable.ffree | 100% | Duke, Nick |
| FileTable.fempty | 100% | Provided |
| FileTableEntry.FileTableEntry | 100% | Provided |
|  |  |  |
| Kernel.java | 100% | Provided |
| TCB.java | 100% | Provided |
| Inode.Inode(short iNumber) | 100% | Nick, Duke, Fuli, Jake |
| Inode.toDisk | 100% | Jake, Duke, Nick |
| Inode.mapOffset | 100% | Jake, Test5.java |
| Inode.addIndex | 100% | Nick |
| Inode.modifyBlockMap | 100% | Nick, Duke, Jake |
|  |  |  |
| Superblock.Superblock | 100% | Provided |
| Superblock.enqueueBlock | 100% | Duke |
| Superblock.dequeueBlock | 100% | Duke |
| Superblock.getIndexOfFree… | 100% | Duke |
| Superblock.getNumDiskBlocks | 100% | Provided |
| Superblock.getNumInodes | 100% | Provided |
| Superblock.format | 100% | Jake, Fuli, Duke |
|  |  |  |
| Individual files compile | 100% | All |
| Java Boot able to run | 100% | All |
| Test5 able to run | 100% | All tests run successfully |

# References and Works consulted

<https://mix.office.com/watch/1wi2kogvp9zhn> – instructor video for File System implementation.

<http://courses.washington.edu/css430/prog/2016_Summer/project.html> – Current File System overview

<http://courses.washington.edu/css430/prog/project_faq.html> – Current File System overview FAQ

<http://courses.washington.edu/css430/prog/project.html> – Previous File System overview

<http://stackoverflow.com/questions/11696472/seek-function> - Extra explanation for how to utilize seek

<https://www.cs.rutgers.edu/~pxk/416/notes/12-filesystems.html> - Detailed explanation of each class

<http://stackoverflow.com/questions/2755006/understanding-the-concept-of-inodes> - Detailed explanation of the Inode math

ThreadOS files