# Final Project Report - File System

**Names (Group C):** Ibrahim Deria, Nour Ali
**Professor:** Erika Parsons
**Class:** CSS430

# Test 5 Results and Additional Testing:

We put our file system through many tests so we can make sure it handled everything we could throw at it. Our test program covered all the bases from the project guidelines, and here's what we found:

## Test 5 Program Results

When we fired up the system, the superblock loaded well.  We opened a test file against the disk to format it and then started creating and deleting some files. Everything ran smoothly and files appeared and disappeared just as they should. We checked out some basic reading and writing to files, and the data went back and forth correctly. Seek pointers were behaving correctly too. Size-related test cases also passed. Lastly, we tested what happened when we opened the same file multiple times – the system kept track of everything correctly.

Throughout our testing, we were super careful about how things work in the linux-like file system and making sure basic file stuff didn't just work, but worked the way you'd expect.

These tests make us feel good about the file system so far. But they also point to where we can make things even better.

```
[alinour@csslab9 ThreadOS]$ javac *.java
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Kernel.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[alinour@csslab9 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
-->
```

# Additional Testing

Here are the extra testing methods we added and their objectives, how they work, and why they're important:

**Format Test**

**Goal**: Make sure the file system can get set up correctly, with the right layout for storing files and tracking what's free.

**How to Test:**

1. Use the file system's "format" command.
2. Manually inspect the key areas of the disk:
3. Superblock: Is this where you expect it, and does it have the right info?
4. Inode blocks: Are these set up and empty, ready for files?
5. Free list: Does this accurately match the unformatted space on the disk?

**Why it Matters:** You can't use a file system until it's formatted, so this is the first step.

**Open-Write-Close-Read Test**

**Goal:** Check if you can do the basics: make a file, put stuff in it, close it, and get that stuff back.

**How to Test:**

1. Create a new file using the file system's commands.
2. Write some sample text (like "Hello, file!") into the file.
3. Close the file.
4. Re-open the file.
5. Read the contents and make sure they're exactly "Hello, file!"

**Why It Matters:** If the most basic file operations don't work, the whole system is broken.

**Append-Read Test**

**Goal:** Can we add more data to the end of a file, and does it read back correctly?

**How to Test:**

1. Open an existing file (or create a new one if needed).
2. Write "This is the first line" and close the file.
3. Re-open the file in append mode.
4. Write "This is the second line".
5. Close the file.
6. Open the file in read mode and make sure the whole file reads as expected.

**Why it Matters:** Lots of programs append to files, like writing to a log.

**Seek-Read Test**

**Goal:** Can you 'jump' around inside a file and read the right parts?

**How to Test:**

1. Write a longer sample text into a file (a few sentences).
2. Use the 'seek' command to move the file pointer to the middle of the file.

3. Read some data starting from that position.
4. Make sure the data you read matches what's actually supposed to be at that location in the file.

**Why it Matters:** Seeking lets you read data without starting at the beginning. Editing files depends on this!

**Delete Test**

**Goal:** Can the file system actually get rid of files, and does it prevent access to them afterward?

**How to Test:**

1. Create a file and write some stuff in it.
2. Use the "delete" command.
3. Try to re-open the file you just deleted. You should get an error message!

**Why it Matters:** You need to be able to free up space, and make sure stuff is really gone when it's deleted.

**Multiple Writes-Reads Test**

**Goal:** Can we write a bunch of stuff to a file over multiple steps, and read it all back correctly?

How to Test:

1. Create a new file.
2. Write "Part 1" to the file.
3. Write "Part 2" to the file.
4. Write "Part 3" to the file.
5. Close the file.
6. Re-open the file and read the whole thing. It should be "Part 1Part 2Part 3".

**Why it Matters:** Files often get changed over time, not all at once.

**Overwrite Test**

**Goal:** Can we change existing data in a file?

**How to Test:**

1. Write "This is old data" to a file.
2. Use the 'seek' command to go back to the beginning of the file.
3. Write "NEW DATA!" (make sure it's longer than the original).
4. Close the file, then re-open it.
5. Read the file back – it should say "NEW DATA!" followed by leftover bits of the old data.

**Why it Matters**: Text editors, image programs, everything that modifies files needs this to work.

**Concurrent Access Test**

**Goal:** If more than one thing tries to change a file at once, does it blow up or stay consistent?

**How to Test:**

1. This depends a lot on the system and weather you can run multiple programs at once:
2. Have one program open a file and start writing.
3. While it's in the middle of writing, have a SECOND program try to write something different to the same file
4. Check that either one or the other write wins, and that the file doesn't end up garbled with mixed data.

**Why it Matters:** Especially with programs running on networks, multiple things might access the same file, the system has to handle this.

**Large File Test**

**Goal:** Files can be bigger than the storage blocks on the disk, does the file system handle that?

**How to Test:**

1. Figure out how big a single block is on your file system.
2. Write enough data to a file to be at least double the block size.
3. Close, reopen, and read the whole thing back, making sure it's right.

**Why it Matters:** Most real files (images, videos, etc.) are going to be much bigger than the underlying storage blocks.

```
[alinour@csslab9 ThreadOS]$ javac *.java
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Kernel.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
[alinour@csslab9 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 MyTests
l MyTests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
Superblock synchronized
Disk formatted.
Format test: Success
Write and read test passed.
Open-Write-Close-Read test: Success
Append and read test passed.
Append-Read test: Success
Seek and read test passed.
Seek-Read test: Success
Delete test passed.
Delete test: Success
Multiple writes-reads test passed.
Multiple Writes-Reads test: Success
Overwrite test passed.
Overwrite test: Success
Concurrent access test passed.
Concurrent File Access test: Success
Large file test passed.
Large File test: Success
All tests completed
-->
```

# File System Specifications (Assumptions, Limitations):

## Overview

The design and implementation of our file system in the ThreadOS environment are aimed at providing a Linux-like file management capability. This system supports basic operations such as creating, reading, writing, deleting files, and manipulating file pointers. It is structured around multiple components: the filesystem, superblock, and the Inodes, which collectively manage the disk's data in a cohesive and efficient manner.

## Assumptions

The following insights were assumptions that we made before and during our implement:

- **Fixed Disk Size:** The file system worked with a predetermined disk size. This simplified design, as it established a clear limit on how much data could be stored.
- **Single-Level Directory Structure:** The file system supported only a single root directory. This simplified lookup but sacrificed the flexibility of nested folders.
- **Limited Number of Open Files:** A restriction existed on how many files a process or thread could have open at the same time. This was a practical constraint for resource management.
- **File Access Modes:** The system supported standard access modes like read-only, write-only, read/write, and potentially append.
- **Synchronization Mechanism:** A basic synchronization mechanism was implemented to prevent multiple processes/threads from corrupting shared file system structures during updates.

## Limitations

As for limitation we had the following limitations in this program:

- **Lack of Hierarchical Directories:** The single-level directory structure meant the file system did not support subdirectories. This limited its ability to effectively organize complex sets of files.
- **Concurrent Access Management:** Instead of using complex management system, basic synchronization was implemented instead. This leads to inefficiencies when multiple threads or processes try to access the same files simultaneously.
- **Storage Efficiency:** The static allocation of inode blocks and the file allocation methods used might have resulted in suboptimal disk space usage. This was especially noticeable when handling many small files.
- **Performance:** We prioritized simplicity and correctness over raw performance. Operations like file allocation, deletion, and seeking may have involved multiple layers of abstraction, introducing some performance overhead.
- **Dynamic Resizing:** The system likely did not support changing the size of the disk after its initial configuration. This limited its flexibility and made it difficult to adapt to changing storage needs.

# Design/Implementation Description:

In designing our file system, we implemented an internal structure that works similar to a file system. The below describes the key components of our design:

**File System:**
This class is the master control for your operating system's file system. It handles the big-picture organization (like the superblock and directories) and the details of individual files through the file table. It has the tools to format your disk, open, close, read, write, and delete files, and it keeps everything synced between the disk and memory. You can even use it to jump around within a file using that seek pointer.

**Inode:**
The Inode class functions as the file's ID card within the file system. It holds all the important details: how big the file is, how often it's used, what permissions it has, and where to find its data on the disk. This class is the key to grabbing data blocks whenever you need them, getting more disk space for the file, and handling those extra structures that really big files need.

**Superblock:**
The Superblock class holds the blueprints for the entire file system. Inside, you'll find stuff like how many blocks the disk has, which blocks are set aside for special purposes, and where the list of available blocks lives. It's the toolbox for setting up the superblock in the first place, formatting the disk, keeping any changes saved, grabbing free blocks when you need them, and putting blocks back on the "available" list.

**Directory:**
The Directory class is the address book for your file system. It's where you find the link between those friendly file names you use and the behind-the-scenes inode numbers that the system works with. To keep things organized, it usually stores file names and sizes in neat little lists. This class also handles the back-and-forth translation between how you see a directory and how it's actually stored on the disk.

**File Table:**
The File Table class acts like a secretary for all the files currently in use. It keeps a detailed list, and each entry is like a file's "open" status report. You need this class to make new entries (when a file is opened) and remove them (when closed). It also lets you check up on a file's status. Basically, this class is all about making sure multiple programs can use files without stepping on each other's toes.

**File Table Entry:**
The File Table Entry class is like a file's progress report. Each entry is a little snapshot of where things stand: where you are in the file (the seek pointer), the file's ID number (the inode stuff), how many programs are using the file, and whether they have permission to read, write, or both. Every time you open a file, one of these gets created to keep track of everything.

**Kernel:**

The Kernel class functions as the operating system's engine room.  It handles all the behind-the-scenes work:  routing requests from programs, juggling processes and threads, managing disk operations, keeping frequently used data handy, and making sure the file system plays nicely with the rest of the system. It's the foundation everything else is built on.

# Performance, Functionality, Future Work:

We spent a good chunk of time thinking about how fast our file system is, what it can currently do, and what kind of cool things we could add to make it even better.  Here's where we think we hit the mark, and where we might need a bit more work.

## Performance

The way we set things up definitely affects how fast our file system is. For example, those inodes that always stay the same size make things simple, but sometimes they waste space. And even though having just one directory works, finding stuff gets slower the more files you have.

The bitmap idea for managing disk space is pretty smart as it's easy and works well. Down the road though, if the disk gets really full, the way files get broken into pieces might start to slow things down. Plus, the way we make sure only one thing changes the files at a time is great for learning, but real-world systems would need something better.

## Features

Here's the core set of commands you can use to interact with our file system:

- **SysLib.format(int files):** This is the "reset" button. It wipes the disk clean and sets up space for a certain number of files (determined by the "files" parameter). Returns 0 if everything goes smoothly, -1 if there's an issue.

- **SysLib.open(String fileName, String mode):** The key to working with files. This does a few things:
    - Opens the file you ask for ("fileName") and chooses how you can access it ("mode"):
    - "r": Read only, like opening a book
    - "w": Write only, erases what was there before
    - "w+": Read and write, also erases what was there
    - "a": Append, lets you add new info to the end
    - If the file doesn't exist, and you used "w", "w+", or "a", it'll create one for you.

    - Gives you a handy file descriptor ("fd"), which is like a bookmark that helps with later commands.
- **SysLib.read(int fd, byte buffer[]):** Copies data from the file (marked by your "fd" bookmark) into your "buffer". It tries to fill the buffer as much as possible, and then tells you how many bytes it actually grabbed. Errors get you a negative number.

- **SysLib.write(int fd, byte buffer[]):** Takes info from your "buffer" and puts it in the file marked by "fd". Like the 'read' command, it tells you how many bytes it successfully wrote. Errors get you a negative number.

- **SysLib.seek(int fd, int offset, int whence):** This one's about your moving that "fd" bookmark around:
    - SEEK_SET (0): Jumps directly to a specific 'offset'.
    - SEEK_CUR (1): Moves relative to where you are now (forwards or backwards) depending on the 'offset'.
    - SEEK_END (2): Skips all the way to the end of the file, plus or minus the 'offset'.
    - It'll even stop you from placing the bookmark before the start of the file or way past the end, and returns where the bookmark ended up.
- **SysLib.close(int fd):** Tidies up. Finishes any pending file changes, removes your "fd" bookmark, and gives a 0 if things went well, a -1 if something wasn't right.

- **SysLib.delete(String fileName):** Deletes a file if the file is closed. If the file's still open, the deletion waits until the file is truly closed then deletes it

- **SysLib.fsize(int fd):** Tells you how big a file is based on the "fd". Is useful for making sure you don't try to put too much stuff in it.

# Future

Looking ahead there are a few ways we could improve the file system. Firstly we can add the functionality of a multi-level directory structure which would make it easier for users to keep their files organized. This change can also help us search for files faster by using better indexing methods.

Next changing the inode structure to so it works with dynamic allocation would use storage space more efficiently especially when dealing with files of different sizes. It would also let us handle larger files which is becoming more and more needed and important.

Finally we could improve performance and reliability by using a better way to manage concurrency. This would be helpful as multi-threaded applications are becoming more common.