CSS430  
Martin Metke  
2017/08/17

Final Project Report

# Part 1

Part 1 details design approach I took.

For this project I decided to make the FileSystem class the focal point of much of the design and development. It’s the entry point for all of our required calls (via SysLib and Kernel) and has handles to almost all of the other new classes, whether directly or indirectly. While it might have been possible to push more functionality (such as initializing the superblock, directory, filetable, etc. during formatting) down to the lower classes, I decided to implement the bare minimum in the classes that would have the most instances running, and kept a large part of the logic in FileSystem.java.

My usual approach to large program design is to work on each lower-level component separately, either before or after writing tests to exercise that specific class or method. However, given the time constraints of this project I took the following approach:

1. Code to spec, making each class fully compilable (but with stub methods)
2. Fill in helper / private / code-compacting methods as #1 requires
3. Use Test5/Test6 files as ad-hoc TDD test suites, and get each test passing before moving on.

This approach was not highly efficient but did get the job done.

## Testing results

l Test5

1: format( 48 )...................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

# Part 2

Part 2 covers the specification and design of the filesystem and its components.

## Specification

The documentation for this project was more complete than in previous assignments, so fleshing out the main classes was not difficult. I made the FileSystem class a heavyweight class, since there is only ever one instance and it has access to almost every other class that supports the filesystem. I made the iNode, Superblock, Directory, and FileTableEntry as light-weight as possible, even going so far as to put a generic field-to-disk/disk-to-fields utility method pair in SysLib, to keep code in the Superblock, Directory, and iNode as compact as possible. I feel this kept those classes simpler and let me focus on the FileSystem as a whole.

On the other hand, the FileSystem class is a monolith by comparison to the other classes, double the LOC of even the Kernel. But this is because a lot of utility methods that simplify large-file reads and writes, file deletion, formatting, and synching are part of this class. I was able to re-use a significant amount of code, or at least compact it into modular methods, by having almost every major function live in the FileSystem. By comparison, the FileTable, Directory, and Superblock are almost entirely concerned with bookkeeping, and the Inodes are entirely block-agnostic. The methods that combine an Inode’s full set of blocks into a contiguous array (for buffered reading or writing) live in the FileSystem class.

## Assumptions

1. I assumed that there could be multiple threads accessing the filesystem simultaneously, so all functions that can manipulate the underlying state of

## Limitations and Performance Concerns

There is a ton of uncached disk access going on here! I have not tested the code with my Cache from Program4 due to concerns about emergent errors. Because I wanted to stick as closely to the provided code, I keep the free block list on disk, with the first free block the only one cached in memory. On the plus side, this means that the free block list is always saved to disk and will survive powerdown without any additional work. The drawback is that there are a lot of times where the FileSystem has to either traverse a set of blocks to find the next free one, or load a whole block just to get at a couple of indirectly-referenced blocks. I feel that going simple was the correct choice given the limited time afforded us on the project, however.

# Part 3

Part 3 covers performance considerations of the various tests.

## Random Access Performance Considerations

Depending on the randomness of Java’s Random Number Generator, I would expect this test to produce performance close to the worst possible. However, true randomness allows for the occasional accidental cache hit; I believe this is what accounts for the ~5% better performance of cached random access over un-cached. That said, my Random Access performance was within 10% of that of the Adversarial Access test, indicating that purely random access is almost as bad as intentionally pessimal access patterns.

## Localized Access Performance Considerations

Using the FAQ as a guide, I wrote the localized access test to only write to, and read from, the same 10 blocks for 200 passes. Mindful of the need to stymie the JVM’s in-built caching, I did set the test up to rotate through a series of byte patterns, but the test is still the optimal use case for cached disk access, so cached performance is almost immeasurably fast compared to un-cached access. So I had to increase the number of tests by a factor of 10 compared to the Random Access and Adversarial access in order to allow Java to actually return a measurable elapsed time for the cached version of this test.

## Mixed Access Performance Considerations

As this test is a modified version of the Localized Access test, I kept the same “200 x 10 x write + read” approach – I expected cached performance to be not far off from the Localized Access test, so there needed to be a sufficient number of accesses for Java to return a measurable elapsed time. I also wanted the test to \*mainly\* access the same 10 blocks on disk, keeping the cache hit rate high. So my Mixed Access test simply adds a check that generates a random number, and chooses a random block off the disk 10% of the time. Since the un-cached performance for this test was identical to the Localized Access test, and cached performance was worse than the fully Localized Access test, I believe this approach is correct and has generated valid data.

## Adversarial Access Performance Considerations

The specification for this test called for “moving the disk head”, that is, accessing data from different tracks of 100 blocks within the 1000-block disk. I have not independently verified that the Disk.class file actually simulates disk seeking, but my approach (forcing each write or read to access a different track than the previous 9 accesses, and randomizing blocks within those tracks) gave a 10% performance decrease over sheer random access, so this too appears to be working correctly.

# Appendix A: Latencies

Latency Comparison Numbers (from <https://gist.github.com/jboner/2841832> )  
--------------------------  
L1 cache reference 0.5 ns  
Branch mispredict 5 ns  
L2 cache reference 7 ns 14x L1 cache  
Mutex lock/unlock 25 ns  
Main memory reference 100 ns 20x L2 cache, 200x L1 cache  
Compress 1K bytes with Zippy 3,000 ns 3 us  
Send 1K bytes over 1 Gbps network 10,000 ns 10 us  
Read 4K randomly from SSD\* 150,000 ns 150 us ~1GB/sec SSD  
Read 1 MB sequentially from memory 250,000 ns 250 us  
Round trip within same datacenter 500,000 ns 500 us  
Read 1 MB sequentially from SSD\* 1,000,000 ns 1,000 us 1 ms ~1GB/sec SSD, 4X memory  
Disk seek 10,000,000 ns 10,000 us 10 ms 20x datacenter roundtrip  
Read 1 MB sequentially from disk 20,000,000 ns 20,000 us 20 ms 80x memory, 20X SSD  
Send packet CA->Netherlands->CA 150,000,000 ns 150,000 us 150 ms  
  
Notes  
-----  
1 ns = 10^-9 seconds  
1 us = 10^-6 seconds = 1,000 ns  
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns  
  
Credit  
------  
By Jeff Dean: http://research.google.com/people/jeff/  
Originally by Peter Norvig: http://norvig.com/21-days.html#answers  
  
Contributions  
-------------  
Some updates from: https://gist.github.com/2843375  
'Humanized' comparison: https://gist.github.com/2843375  
Visual comparison chart: http://i.imgur.com/k0t1e.png  
Animated presentation: http://prezi.com/pdkvgys-r0y6/latency-numbers-for-programmers-web-development/latency.txt