CSS430  
Martin Metke  
2017/08/07

Program 4 Report

# Part 1

Part 1 details the results of running four different tests, each with caching disabled and enabled. The four tests are:

1. Random Access writes and reads
2. Localized Access (all writes and reads accessing the same 10 blocks)
3. Mixed Access with 90% localized writes/reads, and 10% randomized writes/reads
4. Adversarial Access, with each access purposely hitting a different track from the previous access in order to maximize seek time.

## Random Access (Caching Disabled/Enabled)

-->l Test4 disabled 1

l Test4 disabled 1

threadOS: a new thread (thread=Thread[Thread-25,5,main] tid=11 pid=0)

SUITE 1: Random Read/Write Test

All operations completed successfully: true

Total runtime: 22291 ms

Average time per read/write cycle: 111.455000 ms

-->l Test4 enabled 1

l Test4 enabled 1

threadOS: a new thread (thread=Thread[Thread-27,5,main] tid=12 pid=0)

SUITE 1: Random Read/Write Test

All operations completed successfully: true

Total runtime: 21204 ms

Average time per read/write cycle: 106.020000 ms

Based on the FAQ, I used 200 read/write cycles for the Random Access test. In this case, enabling the cache led to a very slight (4.9%) speed-up over un-cached random access.

## Localized Access (Caching Disabled/Enabled)

-->l Test4 disabled 2

l Test4 disabled 2

threadOS: a new thread (thread=Thread[Thread-29,5,main] tid=13 pid=0)

SUITE 2: localized writes + reads in blocks of 10 writes + 10 reads

All operations completed successfully: true

Total runtime: 82792 ms

Average time per 10x write + 10x read cycle: 413.960000 ms

-->l Test4 enabled 2

l Test4 enabled 2

threadOS: a new thread (thread=Thread[Thread-31,5,main] tid=14 pid=0)

SUITE 2: localized writes + reads in blocks of 10 writes + 10 reads

All operations completed successfully: true

Total runtime: 1 ms

Average time per 10x write + 10x read cycle: 0.005000 ms

I chose to make each test 200 passes of 10 writes and 10 reads each, because completely localized cached access is so incredibly fast that Java was reporting 0 ms completion time for smaller numbers of accesses. In this case, the cache performed 82,792 times faster than raw disk access (roughly on the order of magnitude expected when comparing memory access latency to disk access latency).

## Mixed Access (Caching Disabled/Enabled)

-->l Test4 disabled 3

l Test4 disabled 3

threadOS: a new thread (thread=Thread[Thread-33,5,main] tid=15 pid=0)

SUITE 3: Mixed Access: 90% localized, 10% random

All operations completed successfully: true

Total runtime: 82507 ms

Average time per 10x write + 10x read cycle: 412.535000 ms

-->l Test4 enabled 3

l Test4 enabled 3

threadOS: a new thread (thread=Thread[Thread-35,5,main] tid=16 pid=0)

SUITE 3: Mixed Access: 90% localized, 10% random

All operations completed successfully: true

Total runtime: 2 ms

Average time per 10x write + 10x read cycle: 0.010000 ms

I again chose to make each test 200 passes of 10 writes and 10 reads each, for easier comparison to the Localized Access test case. In this case, 10% randomized access caused the cache performance to be halved, down to only 41,253.5 x faster than raw disk access. Mixed disk access performed identically to localized disk access, however.

## Adversarial Access (Caching Disabled/Enabled)

-->l Test4 disabled 4

l Test4 disabled 4

threadOS: a new thread (thread=Thread[Thread-37,5,main] tid=17 pid=0)

SUITE 4: Adversarial Access

All operations completed successfully: true

Total runtime: 23373 ms

Average time per write/read pair: 116.865000 ms

-->l Test4 enabled 4

l Test4 enabled 4

threadOS: a new thread (thread=Thread[Thread-39,5,main] tid=18 pid=0)

SUITE 4: Adversarial Access

All operations completed successfully: true

Total runtime: 22530 ms

Average time per write/read pair: 112.650000 ms

Like Test 1 (Random Access), this test consists of 200 pairs of read/write accesses. However, each accessed block is further restricted to selection from a pre-created map of all tracks, ensuring that head seeking time will be increased (if not maximized). This test approach saw a 10% decrease in performance for both un-cached and cached access (versus Random Access), showing that while the randomness of Random Access was good, it was not pessimal.

# Part 2

Part 2 covers the specification and design of the cache and its block replacement algorithm.

## Specification

Although the Program 4 assignment, notes, and FAQ were in some ways unclear and at odds with each other, the following specification details were solid:

* Cache size of 10 blocks, each block 512 bytes (identical to disk blocks)
* Cached data accesses first seek existing cached blocks, then “invalid” (free) blocks, and finally victims selected via Enhanced Second Chance algorithm.
* Cache must provide interface including the following public methods:  
  constructor, read(), write(), sync(), flush()
* The methods read() and write() must be synchronized

Using these specifications, I drew up a class framework which I filled in based on information from the text, class slides, assignment sheet, and FAQ.

## Algorithm

Overall, my Cache class is fairly simple. It contains an array of CacheEntry “structs”, an inner class type which contains all of the information pertaining to a cached data block. Because the assignment was unclear about how cached data was to be stored, I have each CacheEntry holding a byte[512] array for its associated data; this is a little less performant than using a linear array or matrix to store the cached data within the Cache instance itself, but, I felt, cleaner and more modular. It means that the size of the cached data store and the size of the cache entry table are fully disassociated.

In addition to the required interface, I added a few helper functions to perform common actions, in order to make the public methods cleaner and easier to read through. These include:

* findFreePage(): searches through cache table and returns first “invalid” block index
* nextVictim(): executes Enhanced Second Chance algorithm on existing cached blocks
* selectVictim(): method that uses the above methods to decide which index to read/write into
* writeBackPage(): determines if a victim block needs to be written to disk and executes write
* bufferFromDisk(): tries to read a block from disk; if successful, caches it and fills the read buffer

All of these methods simply encapsulate required code to make read(), write(), sync(), and flush() as simple as possible.

The basic algorithm for Enhanced Second Chance block replacement was clearly delineated in both the assignment and the text, so I won’t cover its details. I did make one change to the listed steps in my nextVictim() method, however: I reduced the number of passes by 1, so that I begin unsetting the Referenced bits immediately rather than on the second pass over the cache table.

My reasoning for this change is thus: the filesystem and kernel for Program4 do not allow for disk blocks to be unallocated as “programs” are unloaded. Therefore there will only ever be exactly 10 “invalid” blocks in the cache, immediately after the Cache is instantiated or flushed, and these blocks will all be used following the first 10 accesses to un-cached data. My Cache also checks to see if all of the free blocks have been used up when searching for a replacement victim. So using the first pass over the cached blocks only to check for invalid blocks therefore represents a 50% useless overhead in the vast majority of cases. Therefore I combined the search for “invalid” blocks with the pass which unsets the Referenced bit, and included a break-out as soon as an “invalid” block is found. This should result in a slightly faster Enhanced Second Chance block replacement algorithm, with the understanding that it might lead to undesirable cache invalidation if this version of ThreadOS supported unloading blocks from memory after “programs” finished execution.

# 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