Specification/design explanation on Cache.java

Cache.java is a program that is meant to act as a disk caching program that would allow for increased reads and writes due to localized data. This program was based on the Second Chance Algorithm to allow for the cache to read, write, flush, and sync cache blocks as needed. The Second chance algorithm itself was used to find victims to be written back the disk and replaced by new data/cache blocks. As these caches were modified, their reference bits and dirty bits were changed to reflect usages and if it would need to be written to the disk if it became a victim.

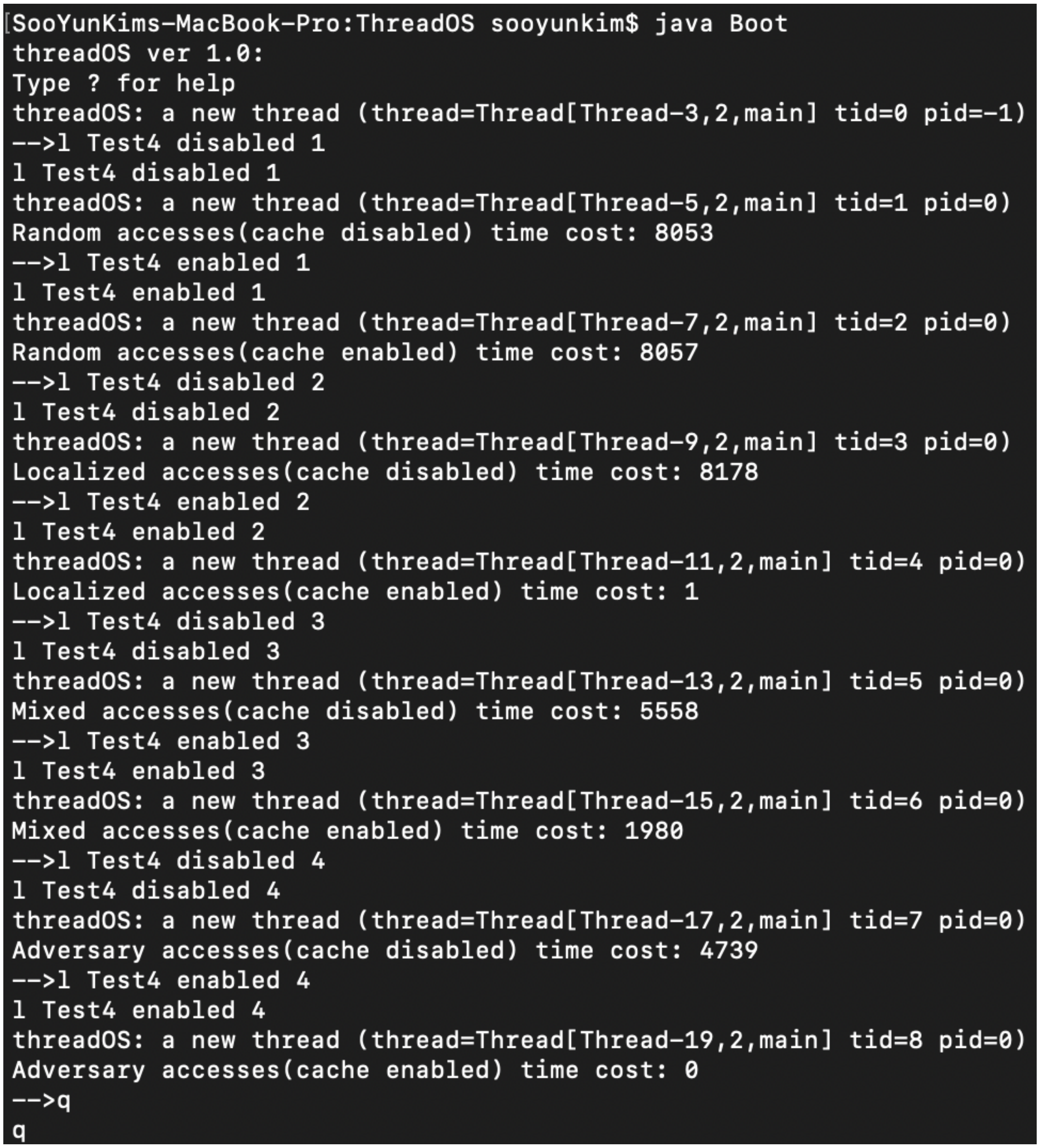
The second chance algorithm would loop over the cache trying to find a cache block of reference bit = 0 and dirty bit = 0. If no cache block met this requirement upon the start of the second loop through the data, it would choose the first reference bit = 0 and dirty bit = 1. It was implemented this way for two reasons, the first being that it would be costly to loop through the cache again to check the bits and that dimpsey said we could implement it this way if we pleased.
Due note that it could be implemented to have a second pass through by changing the "loopCount"} variable to -1 or changing the respected if statement to loopCount == 2 in the findVictim method.

The read method is rather simple. The user provides a blockId and a buffer for the data they want to retrieve. If the blockId is found in the cache, the data is copied from the cache to the buffer. If the cache does not have the data, it will be copied from the disk to the cache then into the buffer. If there is no room in the cache, it will find a victim and write to the disk as needed before overwriting that block with new data from the disk. The second chance algorithm is used to find the victim. If it does as need, it returns true, otherwise false.

The write method will first try to find the blockId provided in the cache. If found, it will write onto that location making that block dirty. If not found, it will write the data to an empty location in the block. If there is no room in the cache, the SCA will find a victim, write as needed to the disk, and then write to the cache. If it does as need, it returns true, otherwise false.

Sync will write all dirty blocks to the cache. Flush will invalidate all blocks in the cache.

Performance results



```
[SooYunKims-MacBook-Pro:ThreadOS sooyunkim$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test4 disabled 1
l Test4 disabled 1
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
Random accesses(cache disabled) time cost: 8053
-->l Test4 enabled 1
l Test4 enabled 1
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=0)
Random accesses(cache enabled) time cost: 8057
-->l Test4 disabled 2
l Test4 disabled 2
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=0)
Localized accesses(cache disabled) time cost: 8178
-->l Test4 enabled 2
l Test4 enabled 2
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=0)
Localized accesses(cache enabled) time cost: 1
-->l Test4 disabled 3
l Test4 disabled 3
threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=0)
Mixed accesses(cache disabled) time cost: 5558
-->l Test4 enabled 3
l Test4 enabled 3
threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=0)
Mixed accesses(cache enabled) time cost: 1980
-->l Test4 disabled 4
l Test4 disabled 4
threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=0)
Adversary accesses(cache disabled) time cost: 4739
-->l Test4 enabled 4
l Test4 enabled 4
threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=0)
Adversary accesses(cache enabled) time cost: 0
-->q
q
```

Performance consideration on random accesses

When looking at the random-access results for both cache and non-cache disk access, we notice that they are strikingly similar. This is because the chance that a random assortment of disk locations/blocks between 0 and 512 is going to be very diverse. The chances that the same number is going to be in the same ten block set is very slim. Because of this, the cache and non-cache perform similarly because both are accessing the disk significantly making act similar to the adverse accesses.

Performance consideration on localized accesses

The cached version of the disk performs significantly better than the non-cached version. The cached version performs almost 0 in both read and write because they are being read/written to main memory rather than the disk itself. When we write/read to the disk, we get a large overhead of I/Os causing it to slow down significantly.

Performance consideration on mixed accesses

This is the most representative of an actual OS. In this test, the cached disk performed better than the non-cached version by about 100%. This is because the cached version still had to do some disk accesses causing it to slow down. But it did not have to do constant disk accesses like the non-cached version. This can be seen with the random accesses vs the localized accesses of the previous two tests.

Performance consideration on adversary accesses

Because this test was to make the worst use of a cached system, we wanted to increase the disk accesses as much as possible. We did this by doing a sequential write and read that would cause the cache to write and read constantly. This is because none of the same block would be called in this scenario. In effect, cached vs non cached performed similar. (Much like random access)