Mustafa Majeed

CSS 430

P4 Report

**Cache.java and Test4.java**

*Design and explanation of Cache.java*

For this project, we were asked to create a Cache.java class to simulate a cache memory chip for ThreadOS. The main focus was on page replacement mechanisms and the performance improvements achieved by implementing a buffer cache that stores frequently-accessed disk blocks in memory. We were asked to implement this using **enhanced second-chance** algorithm. E**nhanced second-chance** can be best described as second-chance algorithm with the addition of modified bit (dirty bit). The modified bit is set to 1 or true anytime the block is altered while in cache, thus signifying that it needs to be written out to disk memory if it needs to be evicted from cache memory. The purpose of enhanced second-chance is to reduce the number of I/O’s required by giving preference to those blocks in cache memory that have been modified. This means keeping them in cache as long as possible until there is no choice but to write them to disk memory prior to eviction. Professor Hogg also asked us to try to implement this project using FIFO + enhanced second-chance. To achieve this, we needed to add an addition field to the private Entry class for Cache.java.

My Cache.java contained:

A private **Entry** class made up of {

Int blockNum // used to store the block number of an entry in the cache block

Boolean ref // used to signify if this entry has been recently accessed of not

Boolean diry // used to signal is this entry has been modified or not

Int tStamp // used to keep track of the position in line of this entry

}

Vector for cache blocks each block of size 512KB

Int time // used to track the position in line for each entry

Int blockSize // used to store the size for each block

Entry[] // used to store information about each corresponding cache block

The main methods in Cache.java are:

**Boolean read**(int blockId, byte[] buf) - This method is used to read a requested blockId into the passed in byte[] using the bytes stored in cachePage Vector. DISK reads will only be performed if the requested blockId is not in cachePage Vector. If the cache is full, and the requested blockId is not found, a victim will be selected for eviction using FIFO + Enhanced 2nd Change Algorithm. DISK writes will also only occur if the dirty bit of the evicted victim is set to true.

**Boolean write**(int blockId, byte[] buf) - This method will write to the requested blockId the contents that are passed in byte[]. If requested blockId is already in the cache, the previous contents of that cachePage will be overwritten with the new passed in content. If the cache is FULL and the requested blockId is not found, then a victim will be selected for eviction to be replaced by the requested content. If that Victim has a dirty bit of true, it is then written to DISK in the appropriate block. After performing a successful write, this this method will always set the blockId cache Entry dirty bit to true.

**Void sync()** - This method will attempt to write all cache blocks to DISK if they contain a valid blockId. Then it will reset the time for this cache to 0 and call SysLib.sync().

**Void flush()** - This method will attempt to write all cache blocks to DISK if they contain a valid blockId and reset all value of cEntries array to default values, like clearing the cache. Then it will reset the time for this cache to 0 and call SysLib.sync().

*How a Victim is determined when an eviction from cache is necessary*

Because of the FIFO + Enhanced second-chance implementation of this Cache class, determining a victim for eviction requires 3 steps.

Step 1 – determine which entry has been residing in cache the longest. This is the reason we needed to add a time stamp to each entry to track its position in line. The lower the time stamp, the longer the entry has been in cache. Upon creation of the Cache object, time in the Cache object is set to 0 and thus the first entry into cache memory will have a time stamp of 0. Time is post incremented after every entry is stamped. To find the longest resident in cache, we just need to loop through all the cache entries and find the smallest time stamp. This will ensure that the next step will always begin with the clock hand pointing at the longest residing entry in cache memory.

Step 2 – now that the clock hand is pointing at the longest residing entry in cache memory, we can begin to look for the first reference bit that is set to false. This will signify that this entry has not been “touched” in a while and can be used as a potential victim for eviction to proceed to step 3. If the current clock hand is pointing to an entry in cache memory that has a reference bit that is set to true, this means that this entry has been accessed recently and is likely to be needed again. So we can set the reference bit to false and move to next entry in cache block to repeat the process until we get to a reference bit that is set to false.

Step 3 – once we get an entry that has a reference bit set to false, we now make sure that we are not writing to disk memory unless we have no other choice. This means that we must perform one last search of cache entries to find the first occurrence of an entry with a reference bit set to false and a dirty bit that is also set to false. That is the ideal victim for eviciton in an enhanced second-chance algorithm. If no such entry exists in our cache memory, then we will be forced to perform a write to disk memory and evict the cache entry with the true dirty bit.

**Performance Results**

We were required to perform 4 different tests with our Cache.java methods enabled and disabled.

*Random Access –* read and write many blocks randomly across the disk. Verify the correctness of your disk cache.

*Local Access –* read and write a small selection of blocks many times to get a high ratio of cache hits*.*

*Mixed Access –*  90% of the total disk operations should be localized accesses and 10% should be random accesses.

*Adversary Access -* generate disk accesses that do not make good use of the disk cache at all.

**Figure 1**

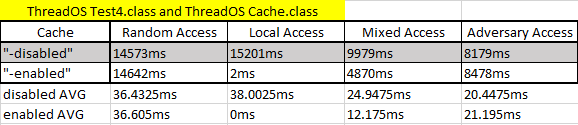


Figure 1 shows the results of these tests using the provided Test4.class and Cache.class in ThreadOS files from the linux cluster. I used this as a barometer for to measure against my Cache.java.

**Figure 2**

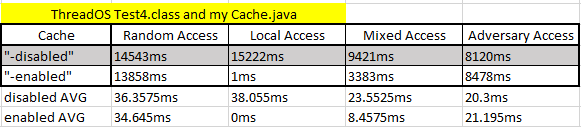


Figure 2 shows the results of the 4 tests using my Cache.java file. I was able to get similar results while improving on the Mixed Access with cache enabled and Random Access with cache enabled. I know that a lot of randomness goes into each test on purpose, but I was delighted to see that my implementation was able to do no worse than the original Cache.class for ThreadOS.

After I was able to get a baseline for testing, I wrote my own Test4.java.

All of my tests perform 400 random reads or writes in sequential fashion to Random blockIds in Random Access, local blockIds in Local Access (the same 10 ids to ensure cache hits), a mixture of 90% local access and 10% random access for Mixed Access, and block ids that ensure cache page faults for each access in Adversary Access.

**Figure 3**

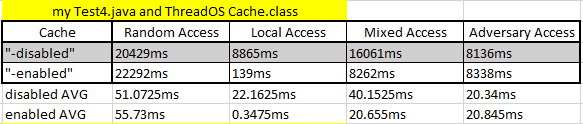
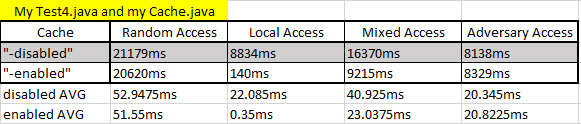


Figure 3 is where things began to look a lot different than with the provided Test4.class in ThreadOS.

I believe that this is because the implementation of Test4 is different than the original. So I tested it with the original Cache.class provided in ThreadOS and my version of Cache.java in figure 4.

**Figure 4**



I was delighted to see that my Cache.java performed about the same as the ThreadOS provided version. I think the main difference is that I perform random reads or writes with each test.

**Results Conclusion**

To conclude this project after evaluating that performance tests.

It is clear that with cache enabled:

* Local Access tests run lightning quick and prove how important caches can be in helping speed up performance for frequently accessed memory blocks and 100% cache hits.
* Mixed Access performance also benefits from having cache enabled because the 90% likelihood of a cache hit will speed up access time and increase performance.
* Random Access makes no difference because it does not guarantee that a cache hit is going to occur so having a cache does not speed up performance at all
* The same can be said for Adversary Access because if each access is going to cause a page fault, then there is no point to have a cache.