Documentation for Program 4

Building Files

For Part 1:
Have *Kernel_org.java* replace the current *Kernel.java*. To build *Kernel.java, Cache.java* simply use the following command:
`javac *.java` or specified java classnames.

For Part 2:
Build the following file: *Test4.java*

Testing Command Line for Test4:
Test4 Enabled 1 ; Test4 Disabled 1 ; Test4 Enabled 2 ; Test4 Disabled 2 ; Test4 Enabled 3 ; Test4 Disabled 3 ; Test4 Enabled 4 ; Test4 Disabled 4 ;

Alternatively
Test4 enabled 5 ;Test4 disabled 5

Casing for enabled|disabled string shouldn't matter.

Program Purpose
In this program, a cache was implemented for ThreadOS as well as a new test class file to demonstrate how an implementation of a cache will differ in performance comparative to ThreadOS without a cache. Given a cache is to store data at a more convenient location (closer to the processor) we generally expect that the runtimes with a cache be quicker than runtimes without one.
 The test class uses 4 different test cases in order to demonstrate the difference in cache-usage performance and the performance when not using the cache.

Case 1: Random Access Test, randomly access the disk with reads and writes
Case 2: Localized Access Test, test for random writes and reads in a small sector of the disk
Case 3: Mixed Access Test, test for 90% Localized Access Test and 10% Random Access Test
Case 4: Adversary Access Test, test that doesn't make good use of the cache

Cache.java
The Cache uses a private class of *PageEntry*, which represents a single page entry as well as the associated data contained within a *PageEntry*. An array called *PageTable* is used to hold multiple *PageEntries*.

The following methods are utilized within *Cache.java*:

read(int blockId, byte[] buffer**)**
This method takes in a byte array termed as buffer, and an int of blockId to read from.

Algorithm:
First an initial check to see if the blockId passed in is valid or not, if so then the first scan is done to determine if the block is already within the cache by blockId. If the block is found, the data is copied from the block in the cache to the buffer byte array. The reference bit is set to True as well due to having accessed the block.

If the block is not already found within the cache, then an empty block is searched. When found, the data is gotten from the disk to the block in the cache, then copied over from the data to the buffer byte array.

If an empty block cannot be found, then a victim selection needs to be done with a second chance algorithm (SCA). When a victim is found, a check is done to see if the data needs to be written back. Afterwards, the same procedure is done, from loading data from the disk, and copying back into the buffer.

write(int blockId, byte[] buffer**)**
This method takes in a byte array termed as buffer, and an inte of blockId to read from.

Algorithm:

This algorithm is very similar to the *read()* algorithm listed above. The primary difference is that the cache isn't simply read into, but instead written. Where the main difference is the dirty bit is set to true and that the data is overwritten from the buffer.

sync()
This method has no parameters to pass in.

The method syncs the data from the cache to the disk by checking if each block has a dirty bit set or not. If a set dirty bit is found, it's written back to the disk and reset.

flush()
This method has no parameters to pass in.

This method is similar to *sync()* where it will check each block in the cache, but the difference is that it will invalidate all blocks or flush them by setting all the block parameters to default. If a set dirty bit is found, it will be written back to the disk and then reset.

diskWrite(int victimIndex**)**
This method takes in the *victimIndex* where it's supposed to have a disk write to.

This method checks if the dirty bit is set, and the block is not empty. A write back is done to the disk then and the dirty bit reset.

findVictim()
This method has no parameters to pass in.

This method uses the enhanced Second Chance Algorithm (SCA) with the assumption that the victim index is already set to one index before the length of the Page Table array. Mainly by looking at the next block to replace through the reference bit.

findBlock(int valToFind**)**
This method has valToFind passed in which indicates an value to check against for the blockId.

This method simply uses a for loop over the entire Page Table byte array to check if any of the blocks' blockId inside is equal to the value that is passed in. If so, the value is returned, otherwise the NOT_FOUND static value is returned.

readCache(int index, int blockId, byte[] buffer**)**
This method has an index, blockId, and a buffer to passed in.

This method uses the index to determine which block to read from and copy from, as well as the blockId to set the current Page Table byte array (Page Entry) to blockId. The reference bit is also set here.

addCache(int index, int blockId, byte[] buffer**)**
This method has an index, blockId, and a buffer to passed in.

This method uses the index to determine which block to write to and copy from, as well as the blockId to set the current Page Table byte array (Page Entry) to blockId. Both the dirty and reference bit are set here.

Test4.java

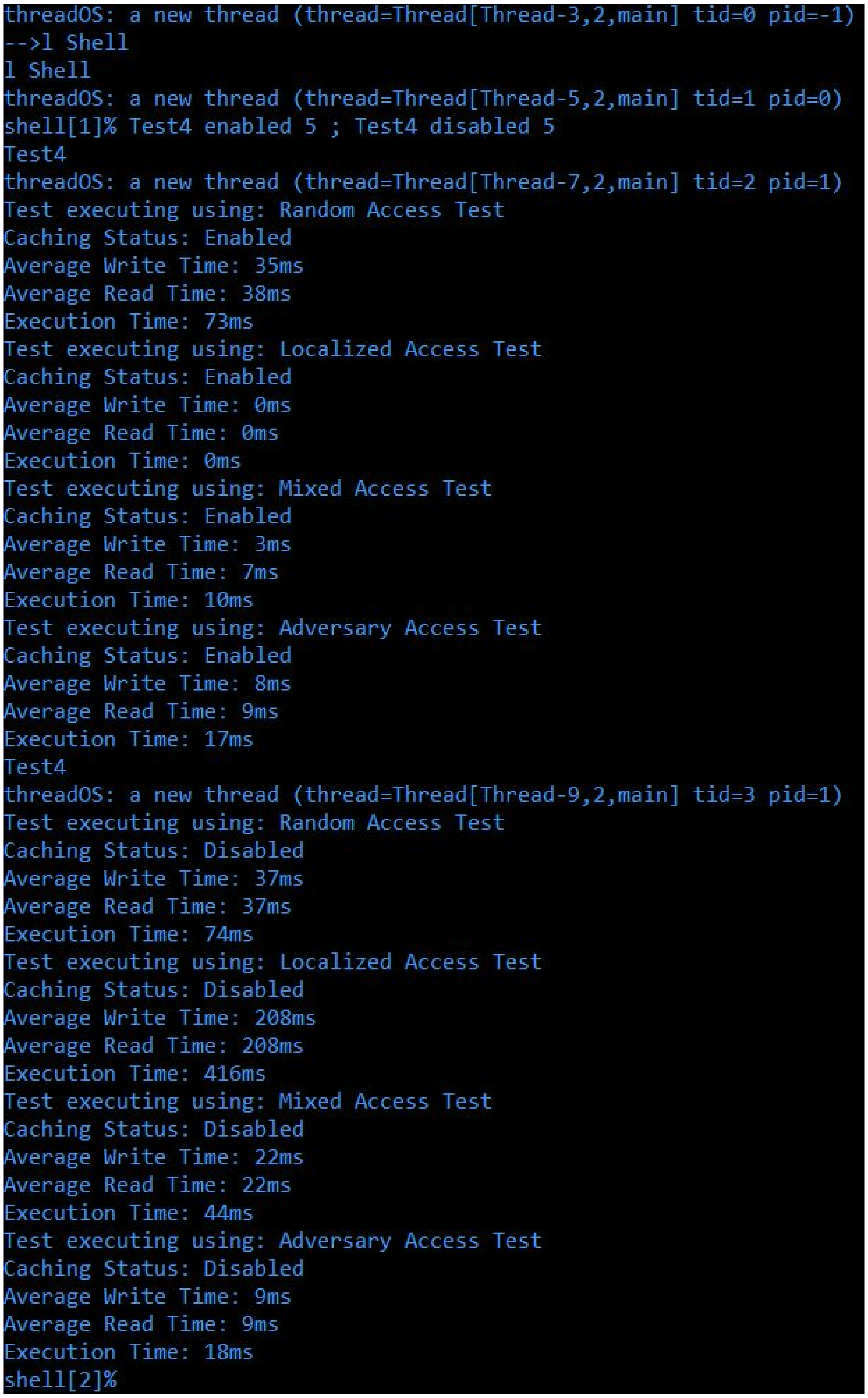
RandomAccessTest()
- Random class is used to generate a random index for the block to access. The index is then used to read/write the data into the buffer.

LocalizedAccessTest()
- Localized block by accessing the nearest 20 blocks.

MixedAccessTest()
- Mixed of localized for 90% through random decision and 10% for random.

AdversaryAccessTest()
- Adversary access which uses odd multiplicative jumps to have inconsistent prior blockIds or block access.

Picture of Test Results Comparing Cached and Non-Cached Versions



```
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Shell
l Shell
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
shell[1]% Test4 enabled 5 ; Test4 disabled 5
Test4
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test executing using: Random Access Test
Caching Status: Enabled
Average Write Time: 35ms
Average Read Time: 38ms
Execution Time: 73ms
Test executing using: Localized Access Test
Caching Status: Enabled
Average Write Time: 0ms
Average Read Time: 0ms
Execution Time: 0ms
Test executing using: Mixed Access Test
Caching Status: Enabled
Average Write Time: 3ms
Average Read Time: 7ms
Execution Time: 10ms
Test executing using: Adversary Access Test
Caching Status: Enabled
Average Write Time: 8ms
Average Read Time: 9ms
Execution Time: 17ms
Test4
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test executing using: Random Access Test
Caching Status: Disabled
Average Write Time: 37ms
Average Read Time: 37ms
Execution Time: 74ms
Test executing using: Localized Access Test
Caching Status: Disabled
Average Write Time: 208ms
Average Read Time: 208ms
Execution Time: 416ms
Test executing using: Mixed Access Test
Caching Status: Disabled
Average Write Time: 22ms
Average Read Time: 22ms
Execution Time: 44ms
Test executing using: Adversary Access Test
Caching Status: Disabled
Average Write Time: 9ms
Average Read Time: 9ms
Execution Time: 18ms
shell[2]%
```

Tests	Cached Execution (ms)	Non-cached Execution (ms)
Random Access	73	74
Localized Access	0	416
Mixed Access	10	44
Adversary Access	17	18

Discussion of Test Results

As you can tell from the picture, the cached versions most significantly improved the performance times of the localized access and the mixed access tests.

However for the random access and the adversary access tests, it seems the difference is negligible.

These results make sense, given that localized and mixed access make the best use of a cached version, given that localized centralizes the disk area to a small subset, which is ideal for a cache as a much higher hit ratio can be obtained as less unique blockIds are likely to be found.

Mixed access has most of the same idea of localized access, given 90% is made to be localized access. As such, it's expected that it also has a high performance increase due to the cached version.

Regarding random access, it's expected that the performance increase is not as significant, given the accessing that is all over the place would cause a low hit ratio to occur. Which doesn't play benefit to the spatial and temporal locality of the caches. While some of the accesses may be in the cache, it would be much more likely that the random access causes another miss than another hint.

Adversary access has a similar approach, where it forces more unique blockIds to be found rather than a small subset of the same kinds. As such, the head of the blockIds bounces around the tracks, which is similar to the random access, causing most of the cache lookups to miss due to the blocks not being in the caches.

As such it’s evident that caching is best done for a more standardized access which has as small as known subset as possible due to having the potential for the highest hit ratio for caching to occur.