Program 4 (Paging)

(CSS 430) Operating Systems

Reza Naeemi

Purpose

This assignment focused on page replacement mechanisms and the performance improvements achieved by implementing a buffer cache that stores frequently-access disk blocks in memory. In this assignment, I implemented the **enhanced second-chance** algorithm, which is described in our OS with Java textbook – Chapter 9.4. I then measured its performance when running various test cases, and considered the merits and demerits of the implementation.

Documentation

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page When a page gets a second chance, its reference bit is cleared (set to 0), and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.
One way to implement the second-chance algorithm (sometimes referred to as the clock algorithm) is a circular queue. A pointer (that is , a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified – best page to replace.
2. (0,1) not recently used by modified – not quite as good, because the page will need to be written out before replacement.
3. (1,0) recently used but clean – probably will be used again soon.

4. (1,1) recently used and modified – probably will be used again soon, and the page will need to be written out to the disk before it can be replaced.

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.
The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

Disk Caching

Caching data from slower external main memory to the faster internal CPU memory takes advantage of both spatial and temporal locality of data reference. User programs are likely to access the same data or the memory locations very close to the data previously accessed. This is called spatial locality and is a key reason why an operating system reads a block or page of data from the disk rather than reading just the few bytes of data the program has accessed. User programs also tend to access the same data again in a very short period of time which in turn means that the least-recently used blocks or pages are unlikely to be used and could be victims for page replacement.
To accelerate disk access in ThreadOS, I implemented a cache that stores frequently-accesses disk blocks into main memory. Therefore, subsequent access to the same block can quickly read the data cached in the memory. However, when the disk cache is full and another block needs to be cached, the OS must select a victim to replace. I employed the **enhanced second-chance** algorithm to choose the block to replace. If the block to be replaced has been updated while in the cache, it must be written back to disk; otherwise, it can just be overwritten with the new block. To maintain this data consistency between disk and cached blocks, each ached block must have the following entry information:

Entry items	descriptions
block frame number	contains the disk block number of cached data. It should be set to -1 if this entry does not have valid block information.
reference bit	is set to 1 or true whenever this block is accessed. Reset it to 0 or false by the second-chance algorithm when searching a next victim.
dirty bit	is set to 1 or true whenever this block is written. Reset it to 0 or false when this block is written back to the disk.

The following instructions summarize how to read, write, cache and replace a block of data:

1. **Block-read algorithm**
   Scan all the cache entries (we were instructed that sequential search is good enough for this assignment). If the corresponding entry has been found in the cache, read the contents from the cache. Otherwise, find a free block entry in the cache and read the data from the disk

to this cache block. If the cache is full and there is not a free cache block, find a victim using the **enhanced second-chance** algorithm. If this victim is dirty, you must first write back its contents to the disk and thereafter read new data from the disk into this cache block. Don't forget to set the reference bit.

2. **Block-write algorithm**
   Scan the cache for the appropriate block (again, sequential search is adequate). If the corresponding entry has been found in the cache, write new data to this cache block. Otherwise, find a free block entry in the cache and write the data to this cache block. I did not have to write the data through to the disk device. I just marked its cache entry as dirty. If I could not find any free cache blocks, then I found a victim using the enhanced second-chance algorithm. If this victim is dirty, I must first write back its contents to the disk and thereafter write new data into the cache block. I then set the reference bit.

3. **Block-replacement algorithm**
   Used the **enhanced second-chance algorithm** mentioned above which is also described in our Operating Systems with Java Textbook, on page 415 and 416.

Implementation

Designed a disk cache based on the enhanced second-chance algorithm and implemented it in my Cache.java. Its specification is:

methods	descriptions
Cache(int blockSize, int cacheBlocks)	The constructor: allocates a *cacheBlocks* number of cache blocks, each containing *blockSize*-byte data, on memory
boolean read(int blockId, byte buffer[ ])	reads into the *buffer[ ]* array the cache block specified by *blockId* from the disk cache if it is in cache, otherwise reads the corresponding disk block from the disk device. Upon an error, it should return false, otherwise return true.
boolean write(int blockId, byte buffer[ ])	writes the *buffer[ ]*array contents to the cache block specified by *blockId* from the disk cache if it is in cache, otherwise finds a free cache block and writes the *buffer [ ]* contents on it. No write through. Upon an error, it should return false, otherwise return true.
void sync( ) and void flush( )	writes back all dirty blocks to *Disk.java* and therefater forces *Diskjava* to write back all contents to the *DISK* file. The *sync( )* method still maintains clean block copies in *Cache.java*, while the *flush( )* method invalidates all cached blocks. The former method must be called when shutting down *ThreadOS*. On the other hand, the later method should be called when you keep running a different test case without receiving any caching effects incurred by the previous test.

The new Kernel.java (renamed Kernel_org.java to Kernel.java. This was provided to us by the professor.) uses the Cache.java in its interrupt() method, so that we didn't have to modify the kernel to work with the new cache implementation.

1. **BOOT:** instantiates a Cache.java object, with 10 cache blocks.
2. **CREAD:** is called from the SysLib.cread() library call. It invokes cache.read().
3. **CWRITE:** is called from the SysLib.cwrite() library call. It invokes cache.write().
4. **CSYNC:** is called from the SysLib.csync() library call. It invokes cache.sync().
5. **CFLUSH:** is called from the SysLib.flush() library call. It invokes cache.flush().

Performance Tests

Wrote a user-level test program called Test4.java that conducts disk validity tests and measures the performance. Test4.java is able to perform the following four tests:

1. **Random Accesses:**
   read and write many blocks randomly across the disk. Verify the correctness of your disk cache.

2. **Localized Accesses:**
   read and write a small selection of blocks many times to get a high ration of cache hits.

3. **Mixed Accesses:**
   90% of the total disk operations should be localized accesses and 10% should be random accesses.

4. **Adversary Accesses:**
   generate disk accesses that do not make good use of the disk cache at all.

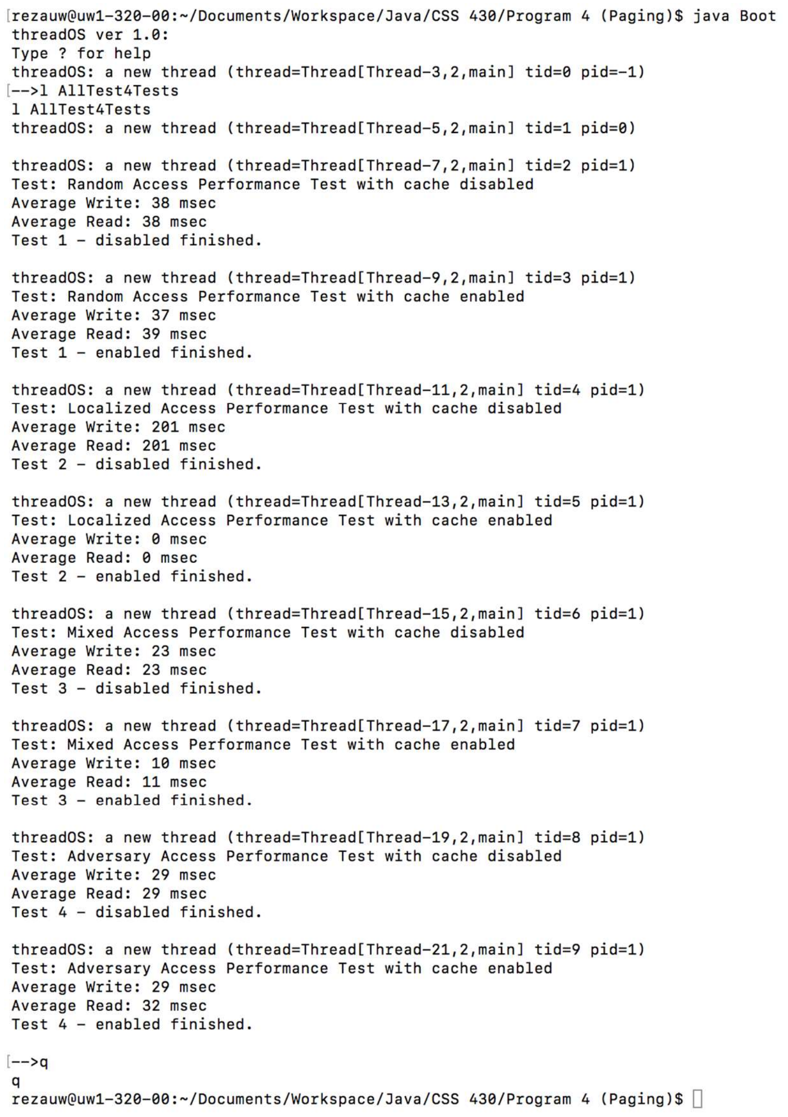
I Implemented all these test items in Test4.java. This java program receives the following two arguments and performs a different test according to a combination of those arguments. (Note: I created another java Thread class that runs all 4 performance tests, both as disabled and enabled. So, a total 8 performance tests.)

1. **The 1st argument:** directs that a test will use the disk cache or not. Test4.java uses SysLib.rawread, SysLib.rawwrite, and SysLib.sync system calls if the argument specifies no use of disk cache (that is, **-disabled**). Otherwise, **-enabled** specifies using SysLib.cread, SysLib.cwrite, and SysLib.csync system calls.

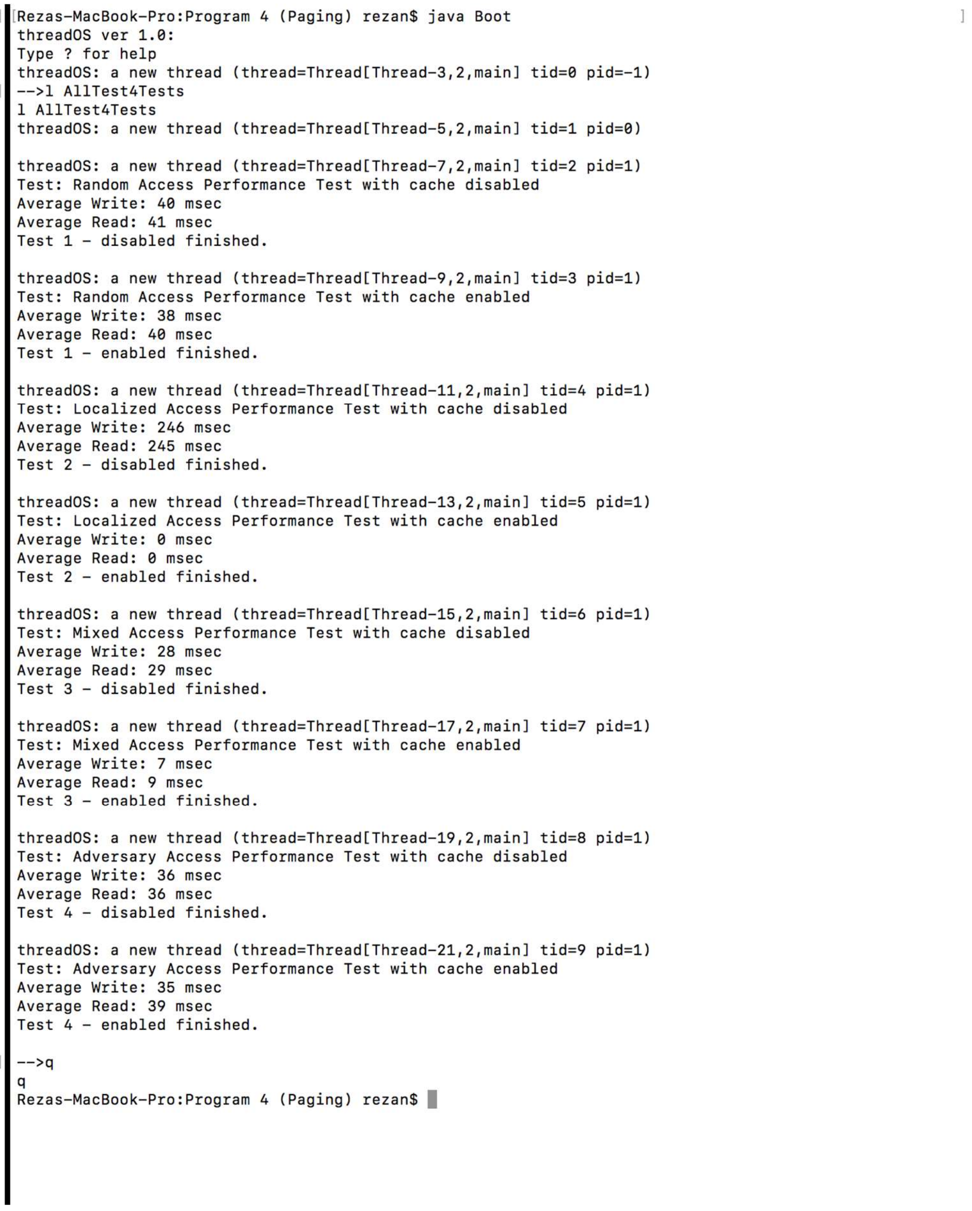
2. **The 2nd argument:** directs one of the above four performance tests.

Execution Console Output (All Performance Tests)

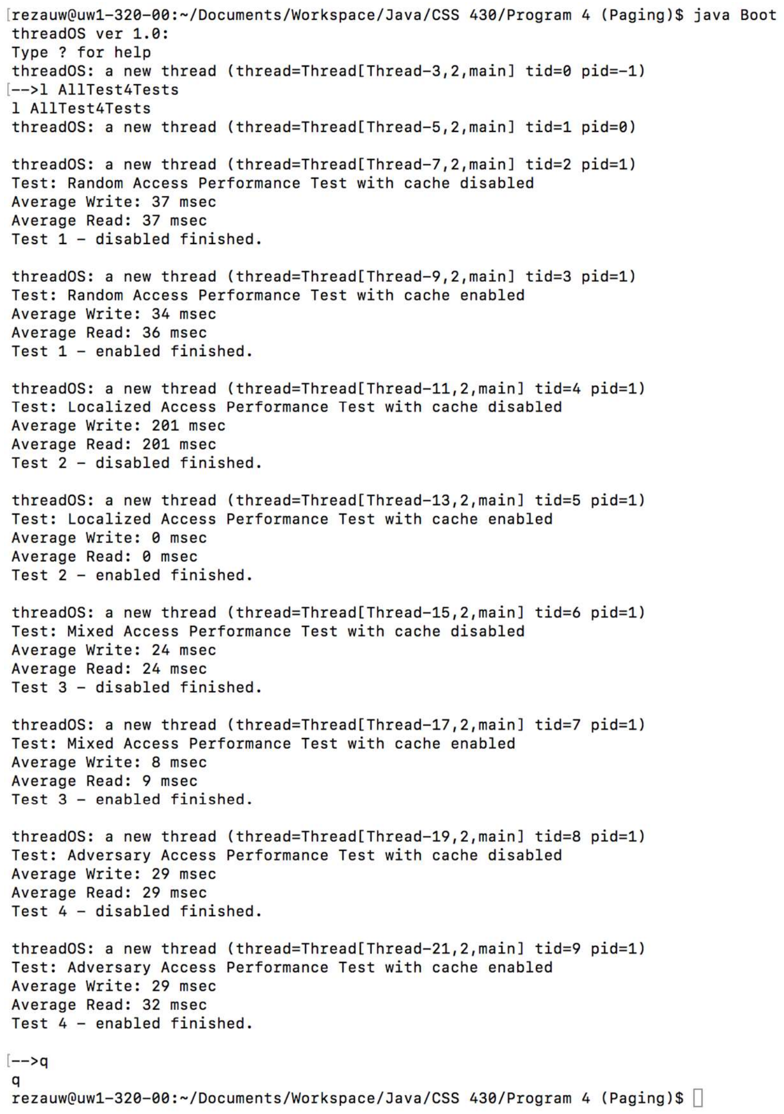
Test4 was ran using my AllTest4Tests script like Java code. I ran each series of tests 3 times on two different systems. I tested on both the UWB Linux Lab (uw1-320-00.uwb.edu) machine, as well as my own local box running macOS. The left side is the UWB system and the right side is my local machine.



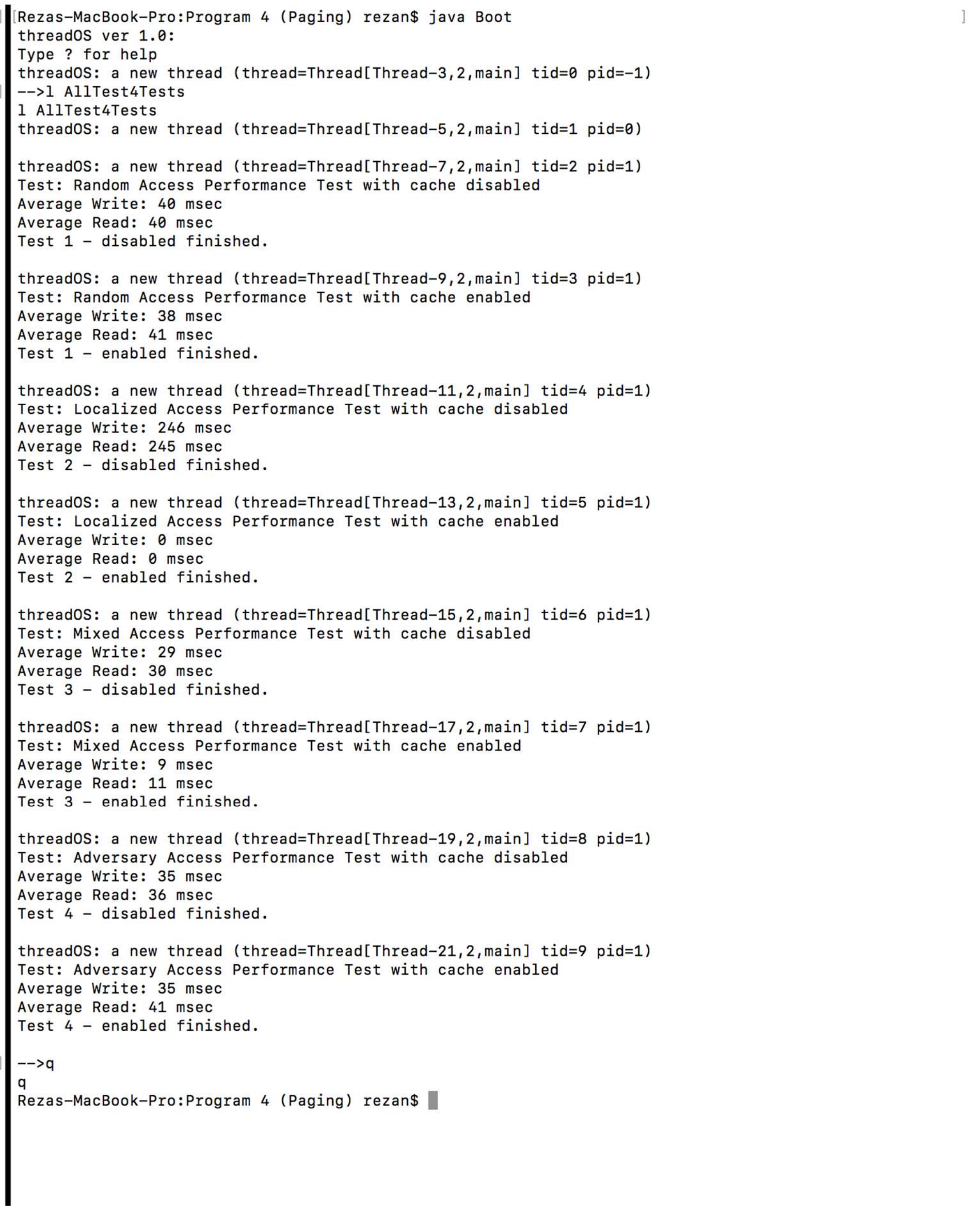
```
rezauw@uw1-320-00:~/Documents/Workspace/Java/CSS 430/Program 4 (Paging)$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l AllTest4Tests
l AllTest4Tests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test: Random Access Performance Test with cache disabled
Average Write: 38 msec
Average Read: 38 msec
Test 1 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test: Random Access Performance Test with cache enabled
Average Write: 37 msec
Average Read: 39 msec
Test 1 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test: Localized Access Performance Test with cache disabled
Average Write: 201 msec
Average Read: 201 msec
Test 2 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test: Localized Access Performance Test with cache enabled
Average Write: 0 msec
Average Read: 0 msec
Test 2 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
Test: Mixed Access Performance Test with cache disabled
Average Write: 23 msec
Average Read: 23 msec
Test 3 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
Test: Mixed Access Performance Test with cache enabled
Average Write: 10 msec
Average Read: 11 msec
Test 3 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=1)
Test: Adversary Access Performance Test with cache disabled
Average Write: 29 msec
Average Read: 29 msec
Test 4 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-21,2,main] tid=9 pid=1)
Test: Adversary Access Performance Test with cache enabled
Average Write: 29 msec
Average Read: 32 msec
Test 4 - enabled finished.

-->q
q
rezauw@uw1-320-00:~/Documents/Workspace/Java/CSS 430/Program 4 (Paging)$
```



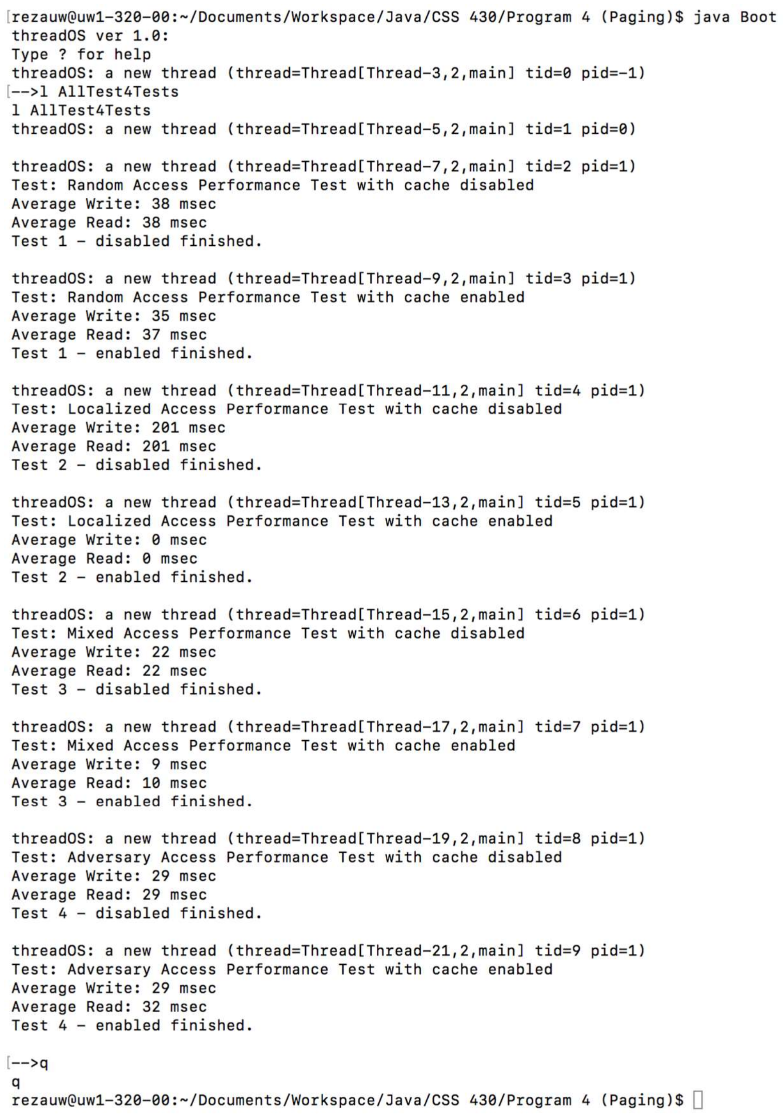
```
Rezas-MacBook-Pro:Program 4 (Paging) rezan$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l AllTest4Tests
l AllTest4Tests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test: Random Access Performance Test with cache disabled
Average Write: 40 msec
Average Read: 41 msec
Test 1 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test: Random Access Performance Test with cache enabled
Average Write: 38 msec
Average Read: 40 msec
Test 1 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test: Localized Access Performance Test with cache disabled
Average Write: 246 msec
Average Read: 245 msec
Test 2 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test: Localized Access Performance Test with cache enabled
Average Write: 0 msec
Average Read: 0 msec
Test 2 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
Test: Mixed Access Performance Test with cache disabled
Average Write: 28 msec
Average Read: 29 msec
Test 3 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
Test: Mixed Access Performance Test with cache enabled
Average Write: 7 msec
Average Read: 9 msec
Test 3 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=1)
Test: Adversary Access Performance Test with cache disabled
Average Write: 36 msec
Average Read: 36 msec
Test 4 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-21,2,main] tid=9 pid=1)
Test: Adversary Access Performance Test with cache enabled
Average Write: 35 msec
Average Read: 39 msec
Test 4 - enabled finished.

-->q
q
Rezas-MacBook-Pro:Program 4 (Paging) rezan$
```



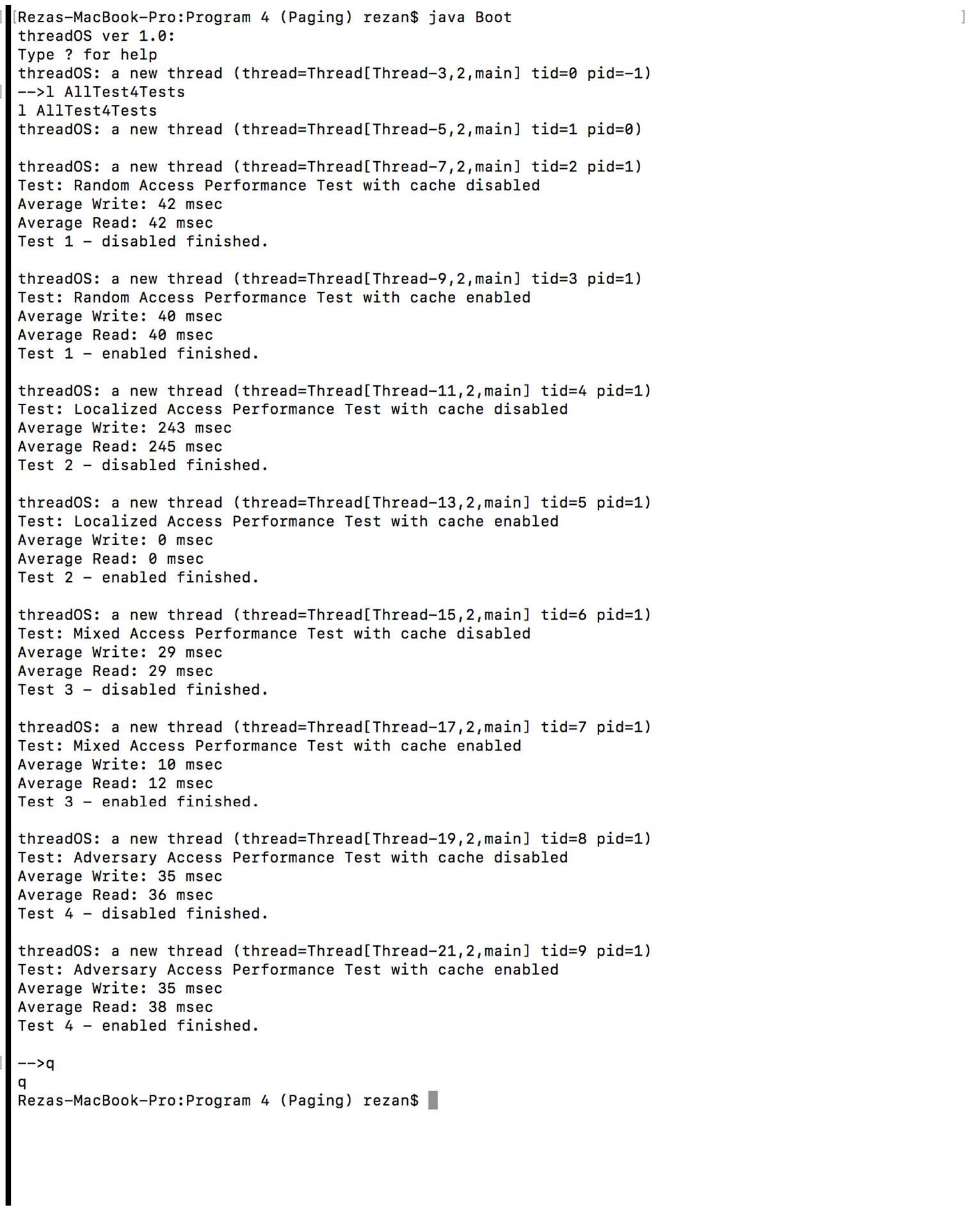
```
rezauw@uw1-320-00:~/Documents/Workspace/Java/CSS 430/Program 4 (Paging)$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l AllTest4Tests
l AllTest4Tests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test: Random Access Performance Test with cache disabled
Average Write: 37 msec
Average Read: 37 msec
Test 1 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test: Random Access Performance Test with cache enabled
Average Write: 34 msec
Average Read: 36 msec
Test 1 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test: Localized Access Performance Test with cache disabled
Average Write: 201 msec
Average Read: 201 msec
Test 2 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test: Localized Access Performance Test with cache enabled
Average Write: 0 msec
Average Read: 0 msec
Test 2 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
Test: Mixed Access Performance Test with cache disabled
Average Write: 24 msec
Average Read: 24 msec
Test 3 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
Test: Mixed Access Performance Test with cache enabled
Average Write: 8 msec
Average Read: 9 msec
Test 3 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=1)
Test: Adversary Access Performance Test with cache disabled
Average Write: 29 msec
Average Read: 29 msec
Test 4 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-21,2,main] tid=9 pid=1)
Test: Adversary Access Performance Test with cache enabled
Average Write: 29 msec
Average Read: 32 msec
Test 4 - enabled finished.

-->q
q
rezauw@uw1-320-00:~/Documents/Workspace/Java/CSS 430/Program 4 (Paging)$ 
```



```
Rezas-MacBook-Pro:Program 4 (Paging) rezan$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l AllTest4Tests
l AllTest4Tests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test: Random Access Performance Test with cache disabled
Average Write: 40 msec
Average Read: 40 msec
Test 1 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test: Random Access Performance Test with cache enabled
Average Write: 38 msec
Average Read: 41 msec
Test 1 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test: Localized Access Performance Test with cache disabled
Average Write: 246 msec
Average Read: 245 msec
Test 2 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test: Localized Access Performance Test with cache enabled
Average Write: 0 msec
Average Read: 0 msec
Test 2 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
Test: Mixed Access Performance Test with cache disabled
Average Write: 29 msec
Average Read: 30 msec
Test 3 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
Test: Mixed Access Performance Test with cache enabled
Average Write: 9 msec
Average Read: 11 msec
Test 3 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=1)
Test: Adversary Access Performance Test with cache disabled
Average Write: 35 msec
Average Read: 36 msec
Test 4 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-21,2,main] tid=9 pid=1)
Test: Adversary Access Performance Test with cache enabled
Average Write: 35 msec
Average Read: 41 msec
Test 4 - enabled finished.

-->q
q
Rezas-MacBook-Pro:Program 4 (Paging) rezan$ 
```



```
rezauw@uw1-320-00:~/Documents/Workspace/Java/CSS 430/Program 4 (Paging)$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l AllTest4Tests
l AllTest4Tests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test: Random Access Performance Test with cache disabled
Average Write: 38 msec
Average Read: 38 msec
Test 1 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test: Random Access Performance Test with cache enabled
Average Write: 35 msec
Average Read: 37 msec
Test 1 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test: Localized Access Performance Test with cache disabled
Average Write: 201 msec
Average Read: 201 msec
Test 2 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test: Localized Access Performance Test with cache enabled
Average Write: 0 msec
Average Read: 0 msec
Test 2 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
Test: Mixed Access Performance Test with cache disabled
Average Write: 22 msec
Average Read: 22 msec
Test 3 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
Test: Mixed Access Performance Test with cache enabled
Average Write: 9 msec
Average Read: 10 msec
Test 3 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=1)
Test: Adversary Access Performance Test with cache disabled
Average Write: 29 msec
Average Read: 29 msec
Test 4 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-21,2,main] tid=9 pid=1)
Test: Adversary Access Performance Test with cache enabled
Average Write: 29 msec
Average Read: 32 msec
Test 4 - enabled finished.

-->q
q
rezauw@uw1-320-00:~/Documents/Workspace/Java/CSS 430/Program 4 (Paging)$
```



```
Rezas-MacBook-Pro:Program 4 (Paging) rezan$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l AllTest4Tests
l AllTest4Tests
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test: Random Access Performance Test with cache disabled
Average Write: 42 msec
Average Read: 42 msec
Test 1 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
Test: Random Access Performance Test with cache enabled
Average Write: 40 msec
Average Read: 40 msec
Test 1 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test: Localized Access Performance Test with cache disabled
Average Write: 243 msec
Average Read: 245 msec
Test 2 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test: Localized Access Performance Test with cache enabled
Average Write: 0 msec
Average Read: 0 msec
Test 2 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
Test: Mixed Access Performance Test with cache disabled
Average Write: 29 msec
Average Read: 29 msec
Test 3 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
Test: Mixed Access Performance Test with cache enabled
Average Write: 10 msec
Average Read: 12 msec
Test 3 - enabled finished.

threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=1)
Test: Adversary Access Performance Test with cache disabled
Average Write: 35 msec
Average Read: 36 msec
Test 4 - disabled finished.

threadOS: a new thread (thread=Thread[Thread-21,2,main] tid=9 pid=1)
Test: Adversary Access Performance Test with cache enabled
Average Write: 35 msec
Average Read: 38 msec
Test 4 - enabled finished.

-->q
q
Rezas-MacBook-Pro:Program 4 (Paging) rezan$
```

Performance Considerations for All Tests

		Average Write (msec)	Average Read (msec)
Random Access with cache disabled			
	UWB Linux - Run 1	38	38
	Local Mac - Run 1	40	41
	UWB Linux - Run 2	37	37
	Local Mac - Run 2	40	40
	UWB Linux - Run 3	38	38
	Local Mac - Run 3	42	42
	Average of both systems together:	**39.16666667**	**39.33333333**
Random Access with cache enabled			
	UWB Linux - Run 1	37	39
	Local Mac - Run 1	38	40
	UWB Linux - Run 2	34	36
	Local Mac - Run 2	38	41
	UWB Linux - Run 3	35	37
	Local Mac - Run 3	40	40
	Average of both systems together:	**37**	**38.83333333**
Localized Access with cache disabled			
	UWB Linux - Run 1	201	201
	Local Mac - Run 1	246	245
	UWB Linux - Run 2	201	201
	Local Mac - Run 2	246	245
	UWB Linux - Run 3	201	201
	Local Mac - Run 3	243	245
	Average of both systems together:	**223**	**223**

Localized Access with cache enabled

UWB Linux - Run 1	0	0
Local Mac - Run 1	0	0
UWB Linux - Run 2	0	0
Local Mac - Run 2	0	0
UWB Linux - Run 3	0	0
Local Mac - Run 3	0	0
Average of both systems together:	**0**	**0**

Mixed Access with cache disabled

UWB Linux - Run 1	23	23
Local Mac - Run 1	28	29
UWB Linux - Run 2	24	24
Local Mac - Run 2	29	30
UWB Linux - Run 3	22	22
Local Mac - Run 3	29	29
Average of both systems together:	**25.83333333**	**26.16666667**

Mixed Access with cache enabled

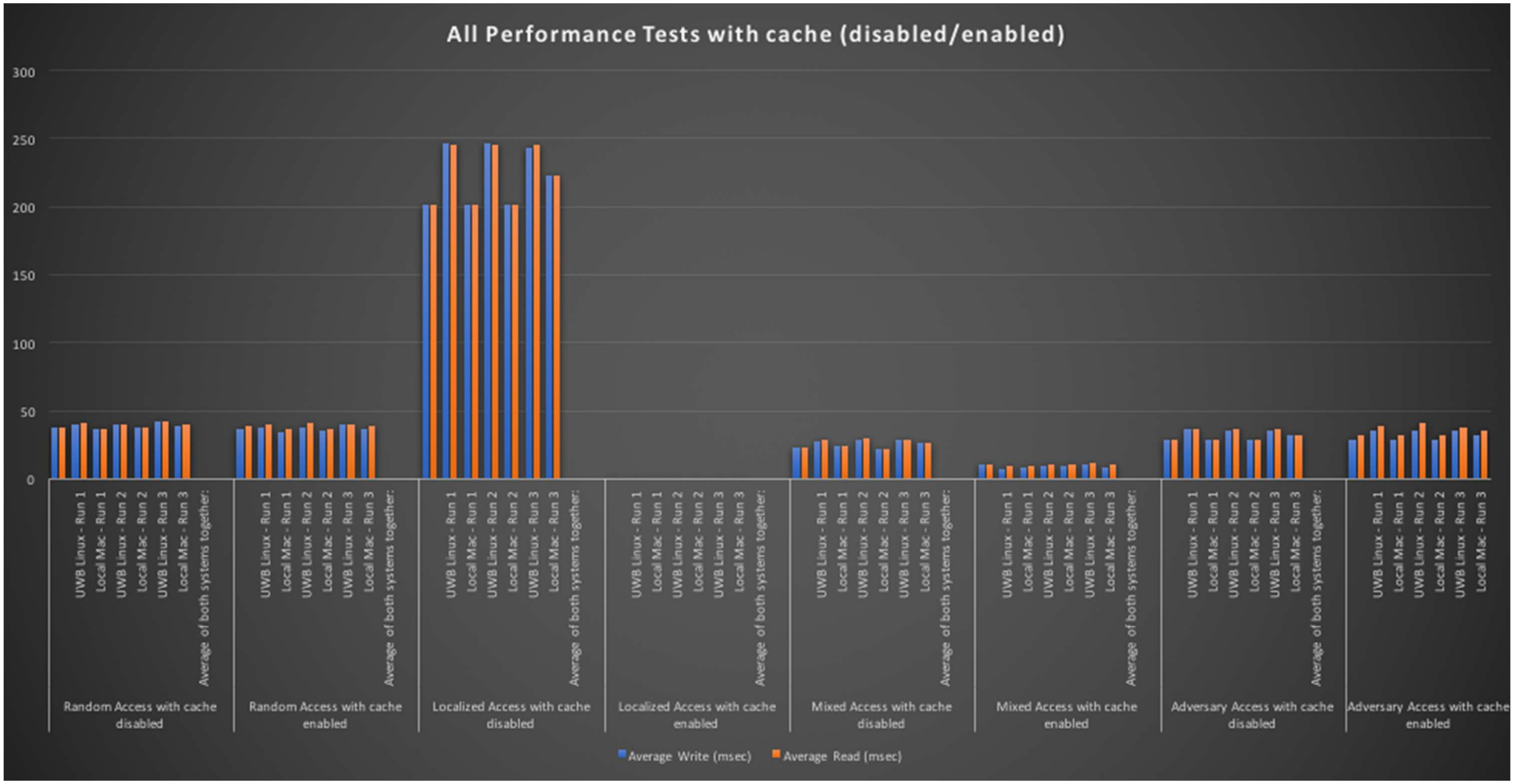
UWB Linux - Run 1	10	11
Local Mac - Run 1	7	9
UWB Linux - Run 2	8	9
Local Mac - Run 2	9	11
UWB Linux - Run 3	9	10
Local Mac - Run 3	10	12
Average of both systems together:	**8.833333333**	**10.33333333**

Adversary Access with cache disabled		
UWB Linux - Run 1	29	29
Local Mac - Run 1	36	36
UWB Linux - Run 2	29	29
Local Mac - Run 2	35	36
UWB Linux - Run 3	29	29
Local Mac - Run 3	35	36
Average of both systems together:	**32.16666667**	**32.5**
Adversary Access with cache enabled		
UWB Linux - Run 1	29	32
Local Mac - Run 1	35	39
UWB Linux - Run 2	29	32
Local Mac - Run 2	35	41
UWB Linux - Run 3	29	32
Local Mac - Run 3	35	38
Average of both systems together:	**32**	**35.66666667**

By running each test 3 times on two different systems and getting the average of each gives me a better understanding of how the disk performance and caching algorithms different systems whose processing power vary. As we can see, we had consistent results, even though clearly the UWB Linux Lab machine(s) out performed my local MacBook Pro, even though my Mac is powerful compared to other student's machines.
We can see that the **worst performance** for both Average Write and Average Read is when we have **Localized Access with cache disabled**, but awesomely enough we can see that our **best performance** for both Average Write and Average Read is when we have **Localized Access with cache enabled**. I am very impressed by these results, as it makes it very clear how important Localized Access with cache enabled is to a performance increase of disk access.

We can see that **Mixed Access with cache enabled** has a performance increased then that of the cache being disabled.

With the other algorithms (**Random Access** and **Adversary Access**), we end of with roughly the same performance with or without caching, for both Average Write and Average Read.


All Performance Tests with cache (disabled/enabled)
300
250
200
150
100
50
0
UWB Linux - Run 1
Local Mac - Run 1
UWB Linux - Run 2
Local Mac - Run 2
UWB Linux - Run 3
Local Mac - Run 3
Average of both systems together:
Random Access with cache disabled
Random Access with cache enabled
Localized Access with cache disabled
Localized Access with cache enabled
Mixed Access with cache disabled
Mixed Access with cache enabled
Adversary Access with cache disabled
Adversary Access with cache enabled
Average Write (msec)
Average Read (msec)