Name: Thuan Tran
CSS 430 Program 4
Date: May 21th, 2017

# REPORT

## Table of Contents

## Purpose of the program

In this program, I implemented a cache for ThreadOS and a test class to test the newly implemented cache. The purpose of using a cache is to store data at a convenient location so that the program will not have to access the disk that is time consuming and expensive procedure to make
The test class (Test4) test for the cache by using 4 types of test with cache on and off

- Test by randomly access the block all over the disk
- Test for localized access by accessing nearby block
- Test for mixed access by combining the above 2
- Test for adversary access by accessing in a way that do not make use of the cache

## Cache.java

The cache is basically a vector of elements that represent a block with data in it. Here are some of the methods that the cache implemented:

- Read: Read the data into a buffer array from a block ID within the cache

- Write: Write the data from the buffer array into the block within the cache
- Sync: Sync all of the new data back into the disk
- Flush: Basically like sync and reset everything : invalidate all reference block
- findVictim(): Using the enhanced second chance algorithm to find the next available block

The Cache.java make use of the of an inner class called Data to hold the represent a block in the disk to hold the data. When the Data is generated, it will represent as an empty block. It will have the following attributes:

- referenceBit: true if the block has been recently used. False otherwise
- dirtyBit: true if the data in the block is new (different from the disk). False otherwise
- blockNumber: same block number within the disk
- a byte array to act as the buffer in order to store information

## FindVictim()

This method implemented the enhanced second chance algorithm by looking for the next available block to replace

### Algorithm

The ideal block to replace is the block that has not been used recently and its data does not chance (dirtyBit is equal to false). So it will look for this block. Along the way, it will also set the reference bit to 0, indicating that the block has not been recently use.

However, if it still can't find it, it means that the blocks are dirty. And it need to select its next to optimal victim. It will decided to do that when it loop through the cache for the second time. When it reach that point, a boolean flag will be used and it will select the next victim. Even if the dirty bit is true

Note that it the algorithm does not write back the data to the disk. This will be the job of the read and write method

## Read()

This method take in a byte array and an integer that represent the block id where it want the information from the block into the array

### Algorithm

The read method first search within the cache to see if the block is already in the cache. If it is, then it simply copy the data from the block in the cache to the byte array. Oh, and set reference bit to True since we just used it

However, if it couldn’t find any, then it need to look for empty block (block that has block number equal to -1). When it find it, it need to get the data from the disk to the block in the cache and then copy the data into the buffer array. It will allocate this block by setting the block number to the block ID
If it can’t still find an empty block, then it need to select a victim by using the enhanced second chance algorithm. When it find a victim, it will also check if the data need to be updated by writing back to the disk. After that, it simply load the data from the disk and copy back to the buffer and allocating this block using the block ID in the parameter

## Write()

This method write the data from the byte buffer into the disk block in the cache

### Algorithm

Same like the read algorithm. It will search if the block is already in the cache. If it is, it updated the data in the block and set the dirty bit to true. Indicating that the data is now different from the disk

If the block is not in the cache, then it search for empty block and update the empty block to represent the block on the disk.

If that didn’t work as well, it will find the next victim using the enhanced second chance algorithm. And write back the data from the block in the cache to the disk. It will then overwrite this block to represent the new block and copy the data from the buffer to the new block. Also set the dirty bit to True

## Sync()

This method sync the data from the cache to the disk. It does this by going through the cache one by one block and check to see if it is dirty or not. And write back all the dirty block to the disk and change the dirty bit to False

## Flush()

This method flush (invalidate) all the data from the cache. It works like the sync method where it write back the dirty block to the disk. It will also set the reference bit, dirty bit to false, block number = -1 to represent a newly created block

# Test4.java

This class tests for the cache by using different methods to write and read data from it. It will take in 2 arguments. The first one indicate if it will use the cache or not and the second one indicate which test it will use

## RandomAccess()

This method will use the Random class to generate random index of the block to access. After that, it will use those same index to read the data into the buffer

## LocalizedAccess()

This method access localized block by only accessing the nearby 10 blocks. It behaves the same as RandomAccess()

## MixedAccess()

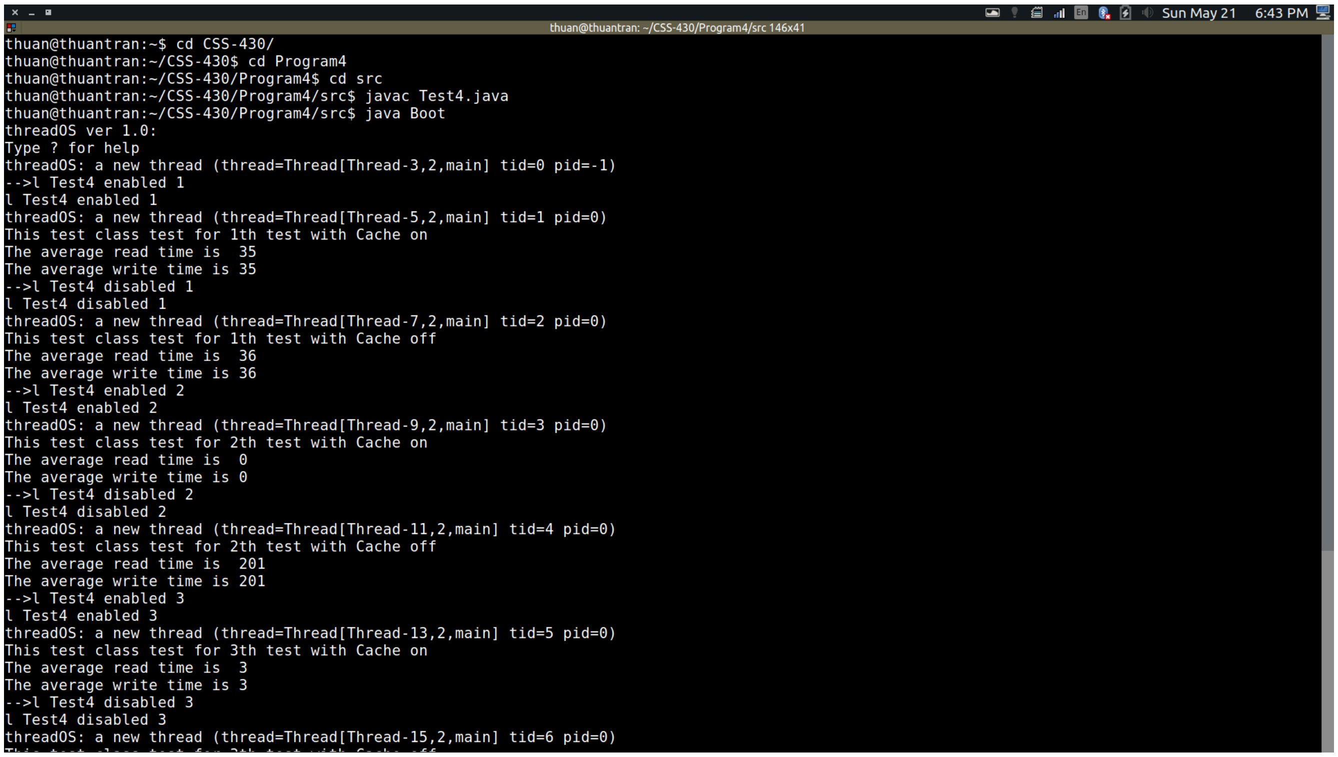
This is the combination of both random access and localized access where 90% of it will be localized and the remaining 10% will be randomly

## Adversary Access()

This is an adversary access where it accesses the index that have not been used in the previous 10 blocks

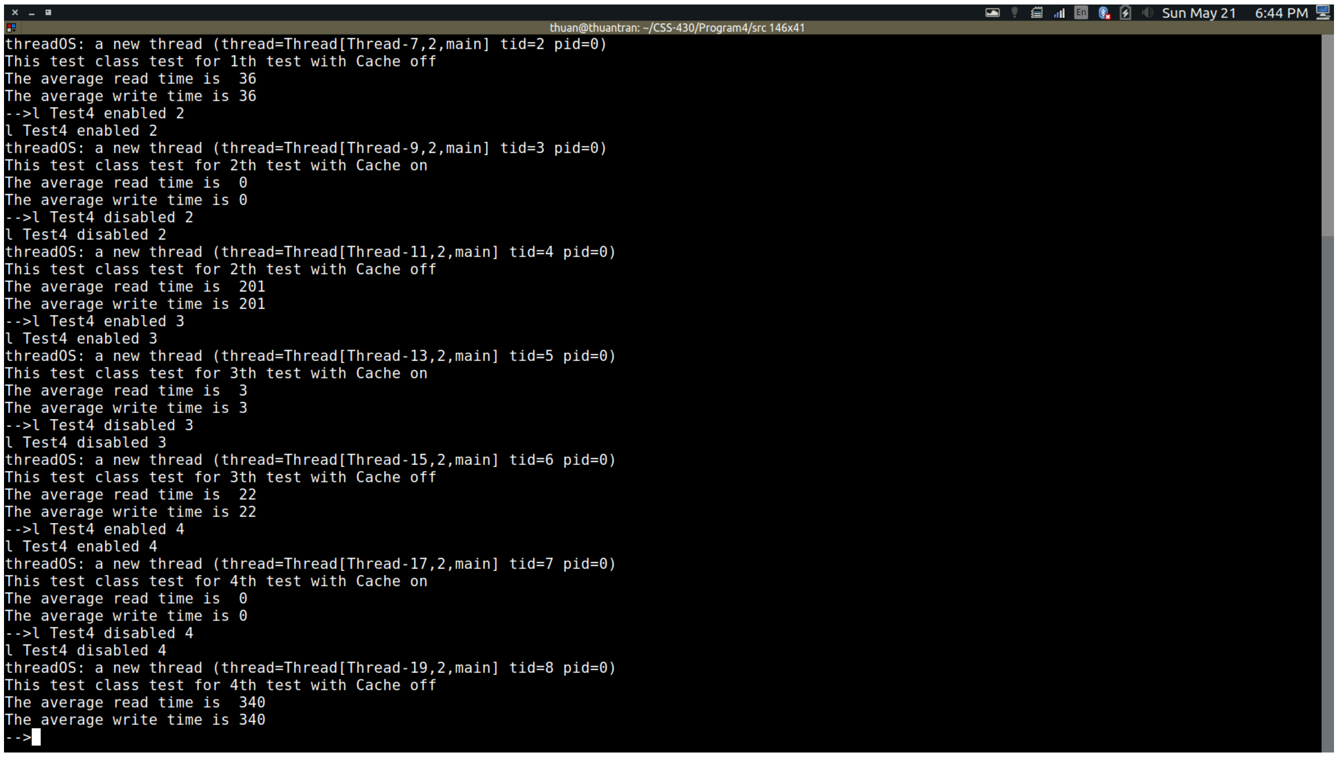
## Result

Here are the results when I ran all the tests with cache on and cache off



```
thuan@thuantran:~$ cd CSS-430/
thuan@thuantran:~/CSS-430$ cd Program4
thuan@thuantran:~/CSS-430/Program4$ cd src
thuan@thuantran:~/CSS-430/Program4/src$ javac Test4.java
thuan@thuantran:~/CSS-430/Program4/src$ 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 enabled 1
l Test4 enabled 1
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
This test class test for 1th test with Cache on
The average read time is  35
The average write time is 35
-->l Test4 disabled 1
l Test4 disabled 1
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=0)
This test class test for 1th test with Cache off
The average read time is  36
The average write time is 36
-->l Test4 enabled 2
l Test4 enabled 2
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=0)
This test class test for 2th test with Cache on
The average read time is  0
The average write time is 0
-->l Test4 disabled 2
l Test4 disabled 2
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=0)
This test class test for 2th test with Cache off
The average read time is  201
The average write time is 201
-->l Test4 enabled 3
l Test4 enabled 3
threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=0)
This test class test for 3th test with Cache on
The average read time is  3
The average write time is 3
-->l Test4 disabled 3
l Test4 disabled 3
threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=0)
```

*Illustration 1: First output*



```
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=0)
This test class test for 1th test with Cache off
The average read time is  36
The average write time is 36
-->l Test4 enabled 2
l Test4 enabled 2
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=0)
This test class test for 2th test with Cache on
The average read time is  0
The average write time is 0
-->l Test4 disabled 2
l Test4 disabled 2
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=0)
This test class test for 2th test with Cache off
The average read time is  201
The average write time is 201
-->l Test4 enabled 3
l Test4 enabled 3
threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=0)
This test class test for 3th test with Cache on
The average read time is  3
The average write time is 3
-->l Test4 disabled 3
l Test4 disabled 3
threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=0)
This test class test for 3th test with Cache off
The average read time is  22
The average write time is 22
-->l Test4 enabled 4
l Test4 enabled 4
threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=0)
This test class test for 4th test with Cache on
The average read time is  0
The average write time is 0
-->l Test4 disabled 4
l Test4 disabled 4
threadOS: a new thread (thread=Thread[Thread-19,2,main] tid=8 pid=0)
This test class test for 4th test with Cache off
The average read time is  340
The average write time is 340
-->
```

*Illustration2 : Second output*

Looking at the pictures, I can determine that using a cache provided an excellent performance increase in the read and write time in certain situations.

For the random access, even though the one with cache one is faster, its difference is only 1. There aren't much boost in that one. One reason because this is a random access, so it costs equally to access random part of the disk and the cache (where the cache have to load the data from the disk as well since the block might not been touch before)

For the localized access, it is easy to see that cache on provided a significant improvement. That is because the data is already in the cache and we just have to access it directly. We do not need to go to the disk and access it

For the mixed access, it also shows that having the cache is beneficial. Because 90% of the test is about localized access and only 10% is random access. So most of the data is already in the cache

For the adversary access, it access block from different track at every time. Using the cache allow the data to be stored inside the cache for quick access later. However, without the cache, it will have to move the head to different track everytime in order to read or write, which is very expensive. Hence the cache provided an excellent boost in this case.