Filipe Sena (41924), Pedro Trindade (41661)
Departamento de Informática - FCT-UNL
{f.sena, pm.trindade}@campus.fct.unl.pt

Keywords: Concurrent hash table; Locks; Consistency;
Deadlock prevention

1. *Introduction*

With the increasing of multi-core processors, data consistency is more and more a must have in concurrent data structures. One wants to achieve the maximum performance for the resources it has. However programing those concurrent data structures can be very hard, since threads accessing those structures can lead to an unpredictable output. If we add up the fact that those structures can depend of each other, the job can be even harder. We propose a solution on how to prevent the occurrence of deadlocks [2] and data races for a particular instance of this problem: when we are using separate chain hash tables [1] as data structures.

2. *Approach*

In our delivery we present several different implementations:
***ArticleRep* -** Our default implementation, using Java reentrant read and writes locks [3] to lock each collision list in the hash table. In this case, concurrent reads are possible.
***ArticleRep2* -** Variation of *ArticleRep*, using Java reentrant locks [4].
***ArticleRep3* -** Implementation using Java synchronized methods to make each insert and remove operation atomic.
***ArticleRep4* -** Alternative version that uses Java reentrant locks to lock each element of the collisions lists instead of the whole list as in *ArticleRep*.

We define our approach for only *ArticleRep* since the others are either similar to the referred or trivial.

In our approach, we extended the Map interface to provide other methods:

```
void singleReadLock(K key);
void listReadLock(List<K> keys);
void singleReadUnlock(K key);
void listReadUnlock(List<K> keys);
void singleWriteLock(K key);
void listWriteLock(List<K> keys);
void singleWriteUnlock(K key);
void listWriteUnlock(List<K> keys);
```

As we said it before, we decided to solve this problem using Read/Write locks from Java. We used an auxiliary lock array to lock the row of the hash table. We also had to implement a method to prevent deadlocks. We use a sorted set [5] to prevent a circular wait to happen. If two threads want to lock a list of elements they have to do it in an ordered manner, it's thus trivial to prevent a circular wait. [6]

The unlocking of the locks is also made in an ordered manner, although not necessary.

After all the necessary locks are done, the user can invoke the usual methods of the hash table.

3. *Validation*

To validate our solution, we started by determining the situations which could occur while running the program that would cause inconsistency in our data (inside the three hash maps). Knowing those situations, we formulated invariants that must never be broken in order to maintain data consistency.

The ***first rule*** we thought of was *the producer and consumer invariant.*[7] Being **c** the number of consumed data items, **p** the number of produced data items and **k** the size of a buffer containing the data items waiting to be consumed:

Invariant. $c >= 0; p >= c; p <= c + k$.

To implement this, we created the three mentioned variables inside the hash table class, incremented **p** inside the put method, incremented **c** inside the remove method and verified the invariant inside an auxiliary method to verify the invariant given **p**, **c** and **k** (named **size** in our program). This auxiliary method is then called inside the given validation method that the validation thread runs in the *repository* class.

The ***second rule*** we thought of was already implemented in the given code (for any article, all the authors and keywords inside it's lists must also have the article inside their respective article lists).

The ***third rule*** is similar to the second one, but the logic is reversed (for any author/keyword, all the articles inside their articles list must exist inside the articles hash map). This prevents the case described in the project description to happen: if a given article is removed from the articles hash map but is still referenced inside any authors/keywords article list.

To implement this invariant, we had to complete the key iterator method in hash table class, in order to iterate through all the existent authors/keywords inside the validation method. To check if each article list of every author/keyword contained only existent articles, we implemented two auxiliary methods (*checkAuthorArticles(String author)* and *checkKeywordArticles(String keyword)*) to be called for every author/keyword.

To test our validation solution we ran our program with the *-Dcp.articlerep.validate=true* flag, which suspends all worker threads and runs the validation method we altered in a single validation thread.

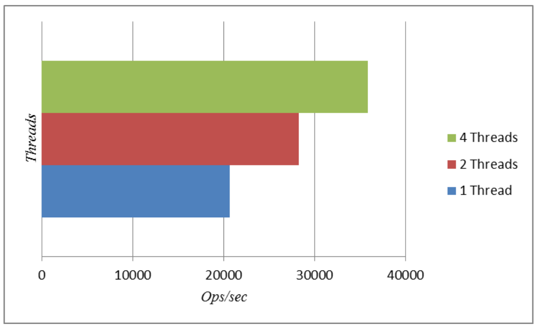
Our project version implemented using reentrant read and write locks passed the validation test, meanwhile the alternative version *articlerep4* failed, which lead us to conclude that it did not guarantee data consistency and that our validation method did indeed verify if the data was kept consistent.

4. *Evaluation*

In this section we evaluate the efficiency of our solution. All tests were performed on a Intel® Core™ i7-3537U CPU @ 2.00GHz × 4 machine. We used the program presented in the previous sections to test our solution. The tests receive the following arguments:
Time - for how long will the program run (in seconds)
Nthread - the number of threads
Nkeys - size of the hash-map – according with the project assignment this is wrong. We explain why in *Section 4.3*
Put(%) - percentage of insert operations
Del(%) - percentage of remove operations
Get(%) - percentage of lookup operations
Nauthors - average number of co-authors for each article (higher number implies more conflicts)
Nkeywords - average number of keywords for each article
Nfindlist - number of authors (or keywords) in the find list
Where, (*Put* + *Del* + *Get* = 100)
All tests were run 5 times during 10 second periods with the internal validation of consistency turned off.

4.1 Throughput vs Threads

On this test we only variate the number of threads. We are trying to understand if the throughput scales with the number of threads. On this test we used the following parameters [10 N 1000 10 10 80 100 100 5], where N is the number of threads {1, 2, 4}.

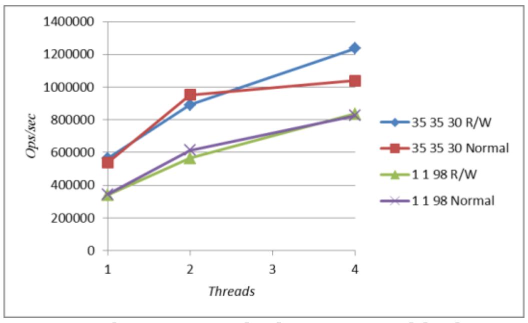


Graphic 1 - Throughput per Threads

As you can see in the graphic the implementation is highly scalable, since for every time we double the threads number we get a plus 9000 ops/sec, and this is in a heavy collision rate that is having 100 as the number of authors per article and the number of keywords per article.

4.2. Read/Write Locks vs Reentrant Locks

On this test we decided to test the Read/write locks implementation vs the Normal locks implementation. This Normal Lock implementation follows the same idea of implementation than the Read/Write that was described in the chapter 2. We are trying to understand in which case is better to use. On this test we used the following parameters [10 N 1000 X X Y 5 5 10], where N is the number of threads {1, 2, 4} and X and Y are represented in the graphic.



Graphic 2 – R/W locks vs Normal locks

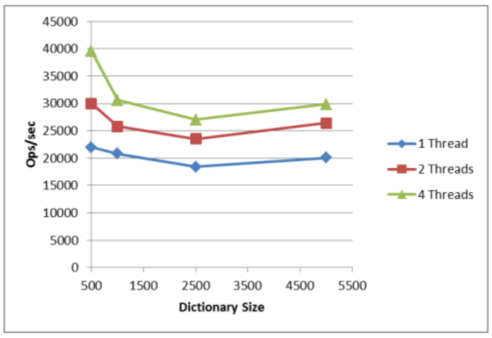
As you can see in the Graphic 2, with one thread the best results are with R/W locks with a (% 35 35 30), after that we have the normal locks really close and then both implementations with (% 1 1 98). One can see that throughput is bigger when the percentages are even distributed. When we increase the number of threads the number of ops/sec increases as well, but this time normal locks overpass W/R locks in both tests. This is because of the overhead that is using W/R locks. At this moment we can declare that, if we want to work in a concurrent environment with 2 threads it's better to use Normal locks. Finally, as soon as we increase the number of threads again we can see an increase of performance in both cases. But let's interpret the results in a separated way.

Consider the case (% 35 35 30) for both R/W and Normal locks (blue and red). One can see that they go in different directions, the opposite than with 2 threads. This is because we have a lot of writes 70% against 30% of reads, but now we can do some reads in parallel since they differentiate from writes. Now our strategy begins to work, since it's more viable than normal locks. We can assume that with more threads this difference would be even bigger, though not possible to verify.

Now consider the other case (% 1 1 98) R/W and Normal locks (green and purple). One can see that the results are the same. By the considered parameters, we are making the read and write operations with W/R locks to be the same as with normal locks, since the operation that will always happen is the read. On this case one can use simply normal locks.

4.3 Varying size of the dictionary

On this test we decided to use out Read-Write Lock implementation and see what we could get from increasing the size of the dictionary. We used as arguments the following parameters: [10 N 10 10 80 100 100 5], where N is the number of threads {1, 2, 4}.



Graphic 3 – Dictionary variation

As we can see in the *Graphic 3*, as soon as we start increasing the size of the dictionary, the throughput decreases. This leads us to our conclusion that we presented in the beginning of this Section. In the assignment specification the definition *Nkeys* represented: “size of the hash-map”, but that doesn’t verifies in this test. Which after this, we can conclude that it was in fact the “Size of the dictionary”, because the bigger the size the more conflicts we have since the words are all the same. In fact, the size of the hash-map is hardcoded in the constructor of Repository that is *40000.* As for the threads we can see that thread 4 is always better than the others, no matter the size of the dictionary. One can see that the implementation is scalable.

5. *Conclusions*

We present our implemented solutions to anyone who might want to try them. We defined the steps that you can do in order to avoid internal deadlocks and data races. We have shown that in a low collision environment one can increase the performance by adding more threads. Although we had successful results our only regret is not having a sufficient number of threads to get more important information between R/W locks and normal locks.
We thought about the drawbacks of using locks on small sets of data, yet, even though we were expecting them, the results were surprisingly high.

6. Acknowledgments

We would like to thank Nuno Martins from DI at FCT-UNL for his help, teaching us how to do an academic report.

7. Bibliography

[1] 'Hash table from Wikipedia',
http://en.wikipedia.org/wiki/Hash_table
[2] 'Deadlock from Wikipedia',
http://en.wikipedia.org/wiki/Deadlock
[3] 'Java 7 Class ReentrantReadWriteLock',
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html
[4] 'Java 7 Class ReentrantLock',
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html
[5] 'Java 7 Interface SortedSet<E>',
http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html
[6] 'Deadlock Prevention',
http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/deadlockPrevent.htm
[7] 'João Lourenço’s Concurrency and Parallelism lecture T8 - Synchronization',
https://clip.unl.pt/utente/eu/aluno/ano_lectivo/unidades/unidade_curricular?aluno=74660&institui%E7%E3o=97747&ano_lectivo=2017&edi%E7%E3o_de_unidade_curricular=11158,97747,2017,s,1