# Homework 3. Java Shared memory performance races

Author's Name, *Affiliation*

## Abstract

In the homework 3, I tried finding the synchronization strategy, which it has not synchronized keyword as well as it is faster than original synchronization strategy by comparing the performance of each different synchronization methods interms of real time, user CPU time, and system CPU time.

## 1. Introduction

Ginormous Data Inc (GDI) has been using the safe synchronization method which uses java's synchronized keyword. However, they want to substitue some other synchronized method that is safe and less heavyweight in comparison with the original synchronization method. In order to find faster and safe synchronization method, I compared NullState which does nothing, SynchronizedState which uses the Synchronized class, UnsynchronizedState which does not use the keyword synchroized, and AcmeSafeState which has better performance than SynchronizedState while retaining safety. To check the performance of each state, I used a simple prototype that manages a data structure that represents an array of longS, and I also used swap method, which substracts 1 from one of the entries, and adding 1 to an entry, so I am able to compare the correctness by checking the sum of all the array entries.

## 2. Coditions

I performed the program under some specific conditions. First of all, I tested the java version "13.0.2". Second, I tested the performance in different two servers lnxsrv09 and lnxsrv10 which have different types of CPUs. The lnxsrv09 server which has 31 processors, cpu model name "Intel(R) Xeon® CPU E5-2640 v2 @ 2.00GHz", while the lnxsrv10 server which has 3 processors, cpu model name "Intel® Xeon® Silver 4116 CPU @ 2.10GHz". Therefore, the cpu lnxsrv09 uses is faster than the cpu lnxsrv10 uses because lnxsrv09 has more number of processors than lnxsrv10, which lnxsrv09 is able to run more number of threads at once compared to lnxsrv 10.

## 3. Experiment

3.1. Implement

First of all, I made a new class UnsynchronizedState by removing the keyword synchronized in the class SynchronizedState. Second, I designed a new class AcmeSafeState which is safe without using the synchronized keyword by using the AtomicLonArray, returning the array which has the long[] data type, and defining the swap function which increases and decreases the value through getAndDecrement(), getAndIncrement functions. Third, I slightly modified the original UnsafeMemory class, which I considered the cases of state "Null", "Synchronized", "Unsynchronized" and "AcmeSafe".

3.2. Benchmarks

I implemented the test by setting the size of state array as 5, 100, and 200, and the number of thread as 1, 8, 25, 40, and the number of swap transitions as 100000. Therefore, the total number of test cases are 48.

3.3. NullState

An implementation of NullState does nothing. Therefore, the swapping function of NullState has no effect. Thus, NullState is just a dummy implementation and is used for finding the overhead.

3.4. SynchonizedState

An implementation of SynchronizedState uses the Synchronized class, so it is safe but slow because of using the keyword synchronized to avoid occurring the race condition while executing swap operation.

3.5. UnsynchonizedState

An implementation of UnsynchronizedState does not use the keyword, so it is much more faster than SynchonizedState but it is not safe because it is possible that other threads can call swap operation while executing swap operation. Therefore, UnsynchronizedState cannot avoid occurring the race condition.

3.6. AcmeSafeState

An implementation of AcmeSafeState is safe without using synchronized keyword. The reason why AcmeSafeState can be safe without using keyword because AcmeSafeState uses AtomicLongArray instead of general array like int[] array, long[] array. Therefore, using ActomicLongArray is able to change the value atomically without using the keyword Synchronization. In addition, ActomicLongArray can allow to change the value through the order which is increase 1,

increase 1, decrease 1, decrease 1, or, decrease 1, decrease 1, increase 1, increase 1. Originally, SyncronizedState changed the value keeping sum zero, which the order is increase 1, decrease 1, or , decrease 1, increase1. Therefore, the swapping operation of ActomicLongArray can be faster than the sapping operation of SynchronizedState because of flexibility for changing the value.

## 4. Analysis

### 4.1. NullState

The total time of NullState steadily increased as increasing the number of threads for every number of state array. I think the reason is that the more the number of thread increases, the more they can split their work into small size. Moreover, if the number of threads increases, the time of cotext switching can be increased, but I set the number of threads by 40, so I was able to deduce that the context switching occurred from small number of threads can be ignored.

### 4.2. SynchronizedState

By comparing the total time of SynchronizedState with the total time of other states, I found out that the total time of SynchronizedState is lowest under the different set value of threads, the number of size array. Therefore, I was able to deduce the reason why the total time of Synchronized State is lowest because Synchonized State has to syncronize operation whenever it calls the function, but NullState, UnsynchonizedState, and AcmeSafeState do not have to synchronize operation, so I was able to clarify that synchronizing operation is the main reason of increasing total time of SynchronizedState.

### 4.3. UnsyncrhonizedState

The speed of UnsynchronizedState was pretty fast as much as NullState because UnsynchronizedState also does not have to synchronize operation when it calls the function. However, lots of sume mismatch happened in UnsynchronizedState. For example, when I operated UnsynchronizedState with the 40 threads and 5 size of state array, the value of output sum is -269. This output some considered, the reliability of UnsynchronizedState is bad as much as it is not safe to use UnsynchronizedState as the synchronization method of program.

### 4.4. AcmeSafeState

The total time of AcmeSafeState was faster than SynchronizedState, and sum mismatch also not occurred in every test case of AcmeSafeState. Therefore, I was albe to clarify that AcmeSafeState class is data-race free. Moreover, the speed of AcmeSafteState was even almost same with NullState and UnsynchrnizedState. From above things considered, AcmeSafteState class can perfectly substitute to SynchronizedState in terms of speed and reliability.

## 5. Problem Faced

When I implement the program with the number of swapping operation which is bigger than 100000, I got the error message "OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limited reached". I thought the reason as to why OutofMemoryError happened because the JVM has a limited heap size, but I used it all, so I reduced the number of swapping operation to 100000, and then performed tests.

## 6. Conclusion

From above the results, the speed of AcmeSafeState was faster than SynchronizedState and almost similar NullState and UnsynchronizedState, and Sum mismatch not occurred in AcmeSafeState. Thus, AcmeSafeState did have any concurrency issue, so it means that AcmeSafeState class is data-race free (DRF). As a result, GDI should uses AcmeSafeState instead of SynchronizedState in terms of speed and reliability.

## 7. Measurements

Measurement of lnxsrv 09

**(I only put the Measurements of lnxsrv09 because of exceeding 5 pages)**

| **NullState** | State array size 5 |
|---|---|
| 1 thread | Total time 0.0151085 s real<br>CPU time 0.0138614 s CPU<br>Average swap time 151.085ns real<br>138.614 ns CPU |
| 8 thread | Total time 0.0170096 s real<br>CPU time 0.114806 s CPU<br>Average swap time 1360.76 ns real |

| | |
|---|---|
| | 1148.06 ns CPU |
| 25 thread | Total time 0.0246200 s real, 0.339947 s CPU Average swap time 6155.01 ns real, 3399.47 ns CPU |
| 40 thread | Total time 0.0198394 s real, 0.302843 s CPU Average swap time 7935.76 ns real, 3028.43 ns CPU |
| | State array size 100 |
| 1 thread | Total time 0.0156502 s real, 0.0145473 s CPU Average swap time 156.502 ns real, 145.473 ns CPU |
| 8 thread | Total time 0.0199595 s real, 0.139608 s CPU Average swap time 1596.76 ns real, 1396.08 ns CPU |
| 25 thread | Total time 0.0181922 s real, 0.293643 s CPU Average swap time 4548.04 ns real, 2936.43 ns CPU |
| 40 thread | Total time 0.0248630 s real, 0.317432 s CPU Average swap time 9945.21 ns real, 3174.32 ns CPU |
| | State array size 200 |
| 1 thread | Total time 0.0161563 s real, 0.0145272 s CPU Average swap time 161.563 ns real, 145.272 ns CPU |
| 8 thread | Total time 0.0189435 s real, 0.129096 s CPU Average swap time 1515.48 ns real, 1290.96 ns CPU |
| 25 thread | Total time 0.0183891 s real, 0.308130 s CPU Average swap time 4597.28 ns real, 3081.30 ns CPU |
| 40 thread | Total time 0.0254478 s real, 0.502309 s CPU |

| | |
|---|---|
| | Average swap time 10179.1 ns real, 5023.09 ns CPU |
| SynchronizedState | State array size 5 |
| 1 thread | Total time 0.0159824 s real, 0.0148573 s CPU Average swap time 159.824 ns real, 148.573 ns CPU |
| 8 thread | Total time 0.0576176 s real, 0.207510 s CPU Average swap time 4609.41 ns real, 2075.10 ns CPU |
| 25 thread | Total time 0.0535120 s real, 0.187340 s CPU Average swap time 13378.0 ns real, 1873.40 ns CPU |
| 40 thread | Total time 0.0558408 s real, 0.201813 s CPU Average swap time 22336.3 ns real, 2018.13 ns CPU |
| | State array size 100 |
| 1 thread | Total time 0.0180738 s real, 0.0159974 s CPU Average swap time 180.738 ns real, 159.974 ns CPU |
| 8 thread | Total time 0.0494694 s real, 0.172756 s CPU Average swap time 3957.55 ns real, 1727.56 ns CPU |
| 25 thread | Total time 0.0579282 s real, 0.200038 s CPU Average swap time 14482.0 ns real, 2000.38 ns CPU |
| 40 thread | Total time 0.0593012 s real, 0.201611 s CPU Average swap time 23720.5 ns real, 2016.11 ns CPU |
| | State array size 200 |
| 1 thread | Total time 0.0157211 s real, 0.0146420 s CPU Average swap time 157.211 ns real, 146.420 ns CPU |
| 8 thread | Total time 0.0507671 s real, 0.175945 s CPU |

| | |
|---|---|
| | Average swap time 4061.37 ns real, 1759.45 ns CPU |
| 25 thread | Total time 0.0597362 s real, 0.205113 s CPU Average swap time 14934.1 ns real, 2051.13 ns CPU |
| 40 thread | Total time 0.0533947 s real, 0.179568 s CPU Average swap time 21357.9 ns real, 1795.68 ns CPU |
| UnsynchronizedState | State array size 5 |
| 1 thread | Total time 0.0152875 s real, 0.0141171 s CPU Average swap time 152.875 ns real, 141.171 ns CPU |
| 8 thread | Total time 0.0198664 s real, 0.139781 s CPU Average swap time 1589.31 ns real, 1397.81 ns CPU output sum mismatch (557 != 0) |
| 25 thread | Total time 0.0204849 s real, 0.351917 s CPU Average swap time 5121.23 ns real, 3519.17 ns CPU output sum mismatch (198 != 0) |
| 40 thread | Total time 0.0233052 s real, 0.388499 s CPU Average swap time 9322.07 ns real, 3884.99 ns CPU output sum mismatch (-269 != 0) |
| | State array size 100 |
| 1 thread | Total time 0.0160946 s real, 0.0147605 s CPU Average swap time 160.946 ns real, 147.605 ns CPU |
| 8 thread | Total time 0.0194855 s real, 0.134944 s CPU Average swap time 1558.84 ns real, 1349.44 ns CPU output sum mismatch (195 != 0) |
| 25 thread | Total time 0.0183913 s real, 0.335322 s CPU Average swap time 4597.82 ns real, 3353.22 ns CPU output sum mismatch (243 != 0) |
| 40 thread | Total time 0.0198199 s real, 0.314898 s CPU Average swap time 7927.96 ns real, 3148.98 ns CPU output sum mismatch (34 != 0) |
| | State array size 200 |
| 1 thread | Total time 0.0175035 s real, 0.0164786 s CPU Average swap time 175.035 ns real, 164.786 ns CPU |
| 8 thread | Total time 0.0220143 s real, 0.154819 s CPU Average swap time 1761.15 ns real, 1548.19 ns CPU output sum mismatch (-33 != 0) |
| 25 thread | time timeout 3600 java UnsafeMemory Unsynchronized 25 100000 200 Total time 0.0196986 s real, 0.354222 s CPU Average swap time 4924.66 ns real, 3542.22 ns CPU output sum mismatch (113 != 0) |
| 40 thread | time timeout 3600 java UnsafeMemory Unsynchronized 40 100000 200 Total time 0.0190937 s real, 0.289957 s CPU Average swap time 7637.46 ns real, 2899.57 ns CPU output sum mismatch (68 != 0) |
| AcmeSafeState | State array size 5 |
| 1 thread | Total time 0.0252555 s real, 0.0222855 s CPU Average swap time 252.555 ns real, 222.855 ns CPU |
| 8 thread | Total time 0.0281938 s real, 0.200687 s CPU Average swap time 2255.50 ns real, 2006.87 ns CPU |
| 25 thread | Total time 0.0258226 s real, |

| | |
|---|---|
| | 0.463127 s CPU<br>Average swap time 6455.64 ns real, 4631.27 ns CPU |
| 40 thread | otal time 0.0537258 s real, 1.00972 s CPU<br>Average swap time 21490.3 ns real, 10097.2 ns CPU |
| | State array size 100 |
| 1 thread | Total time 0.0209924 s real, 0.0200855 s CPU<br>Average swap time 209.924 ns real, 200.855 ns CPU |
| 8 thread | Total time 0.0210920 s real, 0.145089 s CPU<br>Average swap time 1687.36 ns real, 1450.89 ns CPU |
| 25 thread | Total time 0.0249680 s real, 0.482355 s CPU<br>Average swap time 6242.01 ns real, 4823.55 ns CPU |
| 40 thread | Total time 0.0312655 s real, 0.593448 s CPU<br>Average swap time 12506.2 ns real, 5934.48 ns CPU |
| | State array size 200 |
| 1 thread | Total time 0.0161305 s real, 0.0149782 s CPU<br>Average swap time 161.305 ns real, 149.782 ns CPU |
| 8 thread | Total time 0.0227051 s real, 0.160514 s CPU<br>Average swap time 1816.41 ns real, 1605.14 ns CPU |
| 25 thread | Total time 0.0221460 s real, 0.396061 s CPU<br>Average swap time 5536.50 ns real, 3960.61 ns CPU |
| 40 thread | Total time 0.0251991 s real, 0.398167 s CPU<br>Average swap time 10079.6 ns real, 3981.67 ns CPU |