Homework 3. Java Shared Memory Performance Races

1. Testing Environment

In this assignment, I performed the Java shared memory performance races experiment on the SEASNet GNU/Linux servers lnxsrv09, and the following gave the detailed information of the testing environment.

1.1. Java Version

By using the command $java –version, I can see the java version of the SEASNet server is JDK 10.0.1, the Java Runtime Environment is 18.3 (build 10.0.1+10) and 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode).

1.2. System Information

By checking the files, /proc/cpuinfo and /proc/meminfo, the system information can be gathered. From /proc/cpuinfo, I can see that the server has 32 Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz, which has 8 cores and about 20 MB for cache size. On the other hand, from /proc/meminfo, I can know that the server has about 64GB memory in total.

2. Implementation

In this section, I briefly introduced the different classes that I implemented, and also determined the state classes is DRF (data-race free) or not.

2.1. UnsynchronizedState

The unsynchronized method simply removed the keyword *synchronized*, so that the function *swap* can be accessed by multi-threads at the same time. Thus, there could be a data-race in the swap function if different thread wants to modify the same index at the same time.

2.2. GetNSet

The GetNSet introduced *AtomicIntegerArray* in order to read or write data atomically by using get and set. However, since this method couldn't do get and set together atomically, there may be other thread access to the array between read and write step. Therefore, GetNSet is not DRF either.

2.3. BetterSafe

In BetterSafe class, I added *ReentrantLock* to the *swap* function. According to TA class, by calling lock and unlock explicitly, the program could perform better under higher contention since we only lock the instruction that may lead to data race instead of locking *swap* entirely. In this case, BetterSafe class is DRF.

2.4. UnsafeMemory

In this class, I not only integrated all the implemented classes together, but also modified the initialization of the input array. If the input arguments did not give the array, I initialize a random array with size 100. By doing so, I can measure the implemented classes' performance and reliability more reasonably and easily.

3. Experiment Results

3.1. Null and Synchronized classes

In this section, I ran the test on the NullState and SynchronizedState with different parameters, and reported the following results in *ns/transition* of the threads average. Note that all of them initialized a random array with 100 elements, and the reported result is the average of performing the experiment for 10 times.

100 Swap , minval 127	4 Thread	8 Thread	16 Thread
Null	4725830	26703600	56218900
Synchronized	5247800	21714700	48062700

10^4 Swap , minval 127	4 Thread	8 Thread	16 Thread
Null	66255.2	248185	828466
Synchronized	57033.6	98330.4	914398

10^6 Swap , minval 127	4 Thread	8 Thread	16 Thread
Null	1080.65	3317.9	14490.8
Synchronized	2258.22	5579.65	13758.3

For the above chart, we can see that generally synchronized take much longer than null and more thread takes more time. On the other hand, more swap actually takes less time per transition. Also, there is no data race happened in this experiment, we can infer that both of them are reliable.

3.2. Comparison between Implemented Classes

In this section, I ran the test on all the states and make a comparison between them with different parameters, and reported the following results in *ns/transition* of the threads average.

100 Swap , minval 127	4 Thread	8 Thread	16 Thread
Null	4725830	26703600	56218900
Synchronized	5247800	21714700	48062700

Unsynchronized	5506730	8602850	68638200
GetNSet	4916520	1331960	48567300
BetterSafe	970672	3506730	9564100

10^4 Swap , minval 127	4 Thread	8 Thread	16 Thread
Null	4725830	26703600	56218900
Synchronized	5247800	21714700	48062700
Unsynchronized	*50328.5	*160840	*532278
GetNSet	*9073.79	*15384.3	*717852
BetterSafe	36620.8	190720	641848

10^6 Swap , minval 127	4 Thread	8 Thread	16 Thread
Null	4725830	26703600	56218900
Synchronized	5247800	21714700	48062700
Unsynchronized	*654.882	*1528.15	*3589.04
GetNSet	*1289.52	*4036.22	*8821.55
BetterSafe	1763.80	4634.14	12612.9

(* indicate that there are mismatches between the sum of input array and output array)

From the above experiments, we can see that Unsynchronized and GetNSet are no reliable since there could happen data race, which inducing the mismatch between the sum of input data and output data.

In the experiment, we can also infer that BetterSafe is not only reliable, but also very efficient. By comparing the performance and reliability, I would suggest GDI use BetterSafe in their applications.