CS 131 Homework 3 Report

Name: Kaiwen Huang UID: 204171803 Date: 02/05/2016

1. Testing Environment

I tested by ssh to lnxsrv07@seas.ucla.edu
java version "1.8.0_51"
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)
Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
cpu MHz : 1735.234
cache size : 20480 KB
cpu cores : 8

2. Run Test Harness

I ran the test harness on the Null and Synchronized models by varying different parameters and the results show that the average time per transition increases with the increase in number of threads in both models, decreases with the increase of number of swap transitions, increases with the increase in number of state values in the array. For number of state values in the array, I failed to observe an monotonic trend in the change of performance in both models. Both Null and Synchronized models give 100% reliability.

3. Implemented Models

1) Unsynchronized:

I modified the Synchronized model (DRF because there is synchronization at hardware-level) by removing the 'synchronized' keyword from the signature of the 'swap' function. This change resulted in the program without synchronization and therefore the output is unreliable (sum mismatch) due to data races. It is not DRF, because there is no synchronization. A failure case is:

```
java UnsafeMemory Unsynchronized 16 10000 20 2 3 4 5 2 16 7
Threads average 13541.3 ns/transition
sum mismatch (39 != 79)
```

Its performance is better than synchronized because there is no overhead in synchronization implementation.

2) GetNSet:

I used 'AtomicIntegerArray' and the swap method used 'get()' and 'set()'. GetNSet is better than Unsynchronized model because it uses get() and set() to ensure the read and write on the state values are atomic. It is still not DRF because between get() and set() there can be data race. A failure case is:

```
java UnsafeMemory GetNSet 8 10000 20 2 3 4 5 2 10 6 7
```

```
Threads average 12478.5 ns/transition
sum mismatch (39 != 137)
```

'GetNSet' almost always gives sum mismatch. In particular, when the 'maxval' is small such as 6, the program hangs. But maxval above 10 is fine.

3) BetterSafe:

I used the 'ReentrantLock' to synchronize access to the states in the array. This approach makes process of get()->Increment/Decrement->set() atomic and thus is DRF. This 'ReentrantLock' will lock the state before accessing the element so other threads cannot modify the state while one thread is accessing it thus it prevents data races and is DRF(100% reliable). It is also faster than 'Synchronized' model because 'ReentrantLock' can interrupt threads instead of blocking the thread just to wait for a block. But in 'Synchronized' model, the thread is blocked while waiting for the lock which causes much overhead.

4) BetterSorry:

I implemented BetterSorry using 'AtomicIntegerArray' and the 'getAndIncrement', 'getAndDecrement' methods can ensure the modification on the states to be atomic.
It is faster than BetterSafe because there is no usage of 'ReentrantLock' and the above two methods are built at low level and CPU supports them; they don't have the overhead of getting and releasing the locks. It is not DRF because the part of code for updating the states is not ensured synchronization as a whole.
For cases where the maxval is not too small and the state values are not all too close to the upper bound(maxval) or all too close to lower bound, BetterSorry will give reliable results. But Unsynchronized basically fails all cases due to race conditions.
However, BetterSorry still suffers the following case:

```
java UnsafeMemory BetterSorry 8 1000 6 5 6 6 6 6
Threads average 37714.7 ns/transition
output too large (7 != 6)
```

I tested BetterSorry with 15 test cases where the state values are all close to the maxval or all close to 0. BetterSorry fails 2 out of 15 test cases which is about 13.3% failure.

To summarize:

Reliability: 100%=Synchronized = BetterSafe > BetterSorry > GetNSet > Unsynchronized
Performance: Unsynchronized > BetterSorry > GetNSet > BetterSafe > Synchronized

5. Suggestion for GDI application

I would suggest BetterSorry model because BetterSorry has better performance than Synchronized and BetterSafe although there is some sacrifice in the reliability. If there are not many writes to data in GDI application, this is a fair choice.