CS-131 Lab #3 Report Akila Welihinda

My Testing Platform

I developed and tested all my code on the SEASNET server. The Java version on the server is version 1.8.0_45. The machine I was using (lnxsrv04) has 16 CPU's, each of which is an Intel(R) Xeon(R) CPU E5620 running at 2.40GHz with 4 cores. The machine also had 32GB of RAM.

BetterSafe v.s. Synchronized

In my BetterSafe implementation, instead of making the swap function synchronized, I create a lock and force all callers of the swap function to acquire the lock once they enter the body of the function. I have placed the locks such that the critical section is minimized as much as possible. Since the critical section is smaller than the critical section in SynchronizedState, that results in BetterSafeState running faster while having the same accuracy. Although I have reduced the critical section, I have not compromised accuracy because only the code updating the array values is classified as a critical section.

BetterSorry

In my BetterSorry implementation, I create an array of locks with each array element containing its own lock. Whenever a thread wants to update an array element, it must acquire the lock associated with that element. This implementation is faster than BetterSafe because threads processing different elements of the array can now run in parallel. BetterSorry is more reliable than Unsynchronized because there are locks involved, which prevents race conditions. Consider the example where Thread #1 is updating value[1] and value[3] while Thread #2 is simultaneously updating value[3] and value [7]. Unsynchronized is subject to race conditions in this situation, however my BetterSorry is safe in this case because race conditions on value[3] will be avoided due to locking.

However, BetterSorry still has race conditions. In my BetterSorry implementation, I only lock before updating the values. I do not lock before reading the values. This means BetterSorry is still subject to race conditions because two threads can separately view the same element and decide to each decrement that element. This could potentially result in obtaining a negative element.

Here is a test that increases the chance BetterSorry will fail:

```
java UnsafeMemory BetterSorry 32 1000000 2 1 1 1 1 1
```

This test increases the chances of BetterSorry failing because it is more likely that two threads will read an element of value 1 and both try to decrement that element, resulting in an element of value -1.

Performance and Reliability of Each Model

Below contain the average times of each state model. The parameters passed for testing were “8 10000 127 3 4 5 6 3”. The times were averaged over 5 trials for each state model.

Test	Sync	Null	Unsync	GetNSet	BetterSafe	BetterSorry
Avg Time (ns)	10338	8374	8914	12998	13677	9907

In order to reliably get timings for the state models with race conditions, I had to make "maxval" as large as possible and lower the number of required transitions. If I didn't do this, then Unsynchronized and GetNSet would hang instead of terminating. Note that in this table the BetterSafe average time is greater than the Synchronized time. However, if we increase the number of transitions, then BetterSafe will have a lower average time than Synchronized. The reason I have the transitions this low is due to the limitations of GetNSet and Unsynchronized.

Synchronized: This model is DRF because the swap method is synchronized, only allowing one thread at once to execute the swap method. This model is reliable.

Null: This model is DRF because it does nothing. Technically, the Null model is a reliable model.

Unsynchronized: This model is not DRF because there is nothing stopping race conditions. A test with "8 1000000 6 5 6 3 0 3" is likely to fail. This model is not very reliable and will hang with a large transition count.

GetNSet: This model is not DRF although it has atomic read and writes. It's not DRF because the reads and writes aren't done simultaneously, so another thread could modify the value in between the read and write of another thread. A test with "8 1000000 6 5 6 3 0 3" is likely to fail. This model is not very reliable and will hang with a large transition count.

BetterSafe: This model is DRF because I use locks in order to guarantee mutual exclusion. This model is a reliable.

BetterSorry: This model is not DRF because I don't lock around the reads of the array values, which can lead to race conditions. However, this model is still reasonably reliable almost always yields proper results. This is the test case which will most likely cause this model to fail: "32 1000000 2 1 1 1 1 1".

It seems that GDI would want to use the BetterSorry model because it is the second fastest model, and it is still very reliable. If GDI is willing to sacrifice a small amount of correctness for performance, then BetterSorry is its best choice.