UCLA Computer Science 131 Homework 3

Qingwei Lan (404458762)

October 27, 2015

1 Testing Environment Information & Setup

Testing is on `lnxsrv05.seas.ucla.edu` with Java version `java version "1.8.0_45"`. The CPU (`Intel(R) Xeon(R) CPU E5620 @ 2.40GHz`) has 16 4-core-processors. It has `32GB` memory.

2 Individual Implementation Explanations

`UnsafeMemory.java`: This module is modified to signal a timeout when the process hangs, which will happen in case of a deadlock, which is common in the testing of `Unsynchronized`, since it encounters deadlocks often.

`UnsynchronizedState.java`: This module is implemented similar to `Synchronized` except that it removes the `synchronized` keyword. By removing this keyword, the program will not synchronize and will be extremely unreliable, subject to data races. This is also shown in the tests.

`GetNSetState.java`: This module uses the `AtomicIntegerArray` and the `get` and `set` methods of the library `java.util.concurrent.atomic.AtomicIntegerArray`. This implementation will guarantee that the array elements are modified atomically, but it will not guarantee that the upper/lower limit is satisfied for all threads. There will be a data race condition in which two threads both passed the upper/lower limit test but then both modified the same element to get an out of bounds result.

`BetterSafeState.java`: This module is designed with locks using `ReentrantLock` within the library that is called `java.util.concurrent.locks.ReentrantLock` and will lock the the state before accessing array elements. This makes the implementation thread-safe and 100% reliable but also faster than the `Synchronized` model. As a side note, I originally called `lock()` after the upper/lower limit testing. This will result in the same data race condition described above, where two threads both passed the upper/lower limit test but then both modified the same element to get an out of bounds result. Thus I moved the `lock()` to the very beginning of `swap()` to eliminate this race condition.

`BetterSorryState.java`: This module is designed with an array of elements of `AtomicInteger` within the library `java.util.concurrent.atomic.AtomicInteger`. We create an array of `AtomicInteger` and initialize them to be the values in `byte[] v`. This implementation will ensure that all elements of the array will be updated atomically with fast performance, but it is still not free of the race condition mentioned above, where two threads both passed the upper/lower limit test but then both modified the same element to get an out of bounds result.

3 Performance & Reliability Testing

All tests were conducted using `java UnsafeMemory <state type> 8 <#transitions> 100 2 4 6 8 10 10 20 30 40 50 5 6 3 0 3 10 20 30 40 50`.

#Transitions	Null	Synchronized	GetNSet	BetterSafe	BetterSorry	Unsynchronized
10^4	7559.85	10079.59	11213.30	13002.23	8512.51	7884.21
10^5	1624.74	4807.44	2677.43	4701.63	2039.17	1811.51
10^6	1760.39	3108.09	1865.67	1564.16	1471.79	1283.51
10^7	122.81	2846.85	1525.96	934.39	1034.02	312.32

`Null & Synchronized`: Since `Synchronized` uses the `synchronized` keyword, the path is deterministic and is thus reliable. It is also DTF since it is synchronized. `Null` does nothing and returns `true` so it is also 100% reliable in this case. I tested both with different #transitions, #threads, size and sum of array. The basic trend is that average transition time increases with #thread due to overhead and decreases with # of transitions since the large number of transitions level out the overhead caused by threads. I noticed that performance peaked at around 8 threads with 10^7 transitions.

`Unsynchronized`: This version removes the `synchronized` keyword and does not have any synchronizing within it, so its path is undetermined and is thus unreliable. By testing, it almost always produces a mismatch or hangs (from my tests by running `java UnsafeMemory Unsynchronized 8 100000 10 2 4 6 8 10` for 500 times, it produces a mismatch every time).

`GetNSet`: This version is more reliable than `Unsynchronized` but still produces data races. The data race condition is explained in the Individual Implementations section above. This implementation is reliable most of the times but is subject to race conditions when the number of threads is large and array has elements close to the boundary conditions. I wrote a specific test script to test this condition. It runs the command `java UnsafeMemory GetNSet 8 1000 6 5 6 6 6 6` for 500 times and it produced a mismatch 14 times. We can calculate the reliability as follows $reliability = \frac{\#\ tests\ successful}{\#\ tests\ total} = \frac{500-14}{500} = 97.2\%$. So this implementation is quite reliable in this sense that it maintains 97% reliability even in bad cases of input.

`BetterSafe`: This implementation is also 100% reliable because the path is deterministic. Since we put locks around the critical section, only one thread can access the shared data and therefore there will be no race conditions. This implementation also has better performance than `Synchronized` because locks are lightweight compared to the implementations of the `synchronized` keyword.

`BetterSorry`: This version is more reliable than `Unsynchronized` but still not 100% reliable. The only data race condition it will encounter is explained in the Individual Implementations section above. However, `Unsynchronized` does not enforce atomic updating while `BetterSorry` does, so this version is more reliable than `Unsynchronized` since it will not encounter one of the data race conditions that `Unsynchronized` will. I also wrote a specific test script to test this version, which also tests the case where the number of threads is large and array has elements close to the boundary conditions. The script runs `java UnsafeMemory BetterSorry 8 1000 6 5 6 6 6 6` for 500 times and it produced a mismatch 7 times. We calculate reliability as shown in the `GetNSet` case $reliability = \frac{\#\ tests\ successful}{\#\ tests\ total} = \frac{500-7}{500} = 98.6\%$. It has better reliability than `GetNSet` in bad cases, and definitely better than `Unsynchronized`. It also yields better performance than `Synchronized`. When compared to `BetterSafe`, `BetterSorry` is also faster because it does not lock up the data access as does `BetterSafe`.

4 Conclusion & Recommendations

According to the assignment, GDI seems to want to find patterns in current data, which involves only reading the data and analyzing. This means that it would not actually write to the data that is recorded. Thus with read-only data we do not need to synchronize even if we have multiple readers, so my recommendation would be `Unsynchronized`.

But if we need to write to the data, I would recommend `BetterSorry` since runs fast and maintains a high probability (typically 100% for normal cases) even at boundaries.