Environment Information

Test on server lnxsrv07.seas.ucla.edu

java -version: 1.8.0_51

Processor model: Inter(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz - 8 cores

Memory: 65759080 kB total

Implementation

Synchronized: It is DRF because it restricts the access to the "swap" method. Meaning that only one thread has access to the resources.

Unsynchronized: Same as Synchronized, however no"synchronized" keyword. It is not DRF. There is no protection agains out of order Reads and Writes that affect the behavior of the program.

GetNSet: No use of "Synchronized" keyword. Uses "private AtomicIntegerArray value;", so that the values of the array are volatile. Increments the array as "value.set(j,value.get(j)+1)". Decrements in a similar way. A drawback of this is that the State class receives byte[] for its methods. Then it was necessary to implement auxiliary methods to convert from int[] to byte[] and byte[] to int[] so it is possible to use AtomicIntegerArray.

It uses volatile accesses, meaning that all threads have access to the more recent data. There is no stale information. It is faster than Synchronized in several ocasions because it does not block the access to the "swap" method.

It is not DRF. From table 1 we see that it commited mistakes in all the executions. Specifically it tends to deadlock a lot since its error rate is high. That means that it sets the entire array to either maximum or minimum value and the if statement of "swap" keeps returning false forever.

An execution that most likely will make both GetNSet and Unsynchronized deadlock is:

java UnsafeMemory GetNSet 16 10000 2 1 1 1

BetterSafe: Implemented using "private ReentrantLock stateLock = new ReentrantLock();". A thread acquires the lock when enters the "swap" method and releases it before exiting it. Therefore it is DRF, because there can't be inconsistencies of writes and reads. It tends to be faster than using "synchronized" because it has less overhead.

BetterSorry: Similar to GetNSet. However, this time gets and increments the value in the same atomic operation using: "value.getAndIncrement(j);". Furthermore, in order to avoid deadlocks, the if statement of swap returns true even if it is satisfied. It is faster than BetterSafe because it doesn't lock the "swap" method. Instead, it just performs the decrement and increment atomically.

I tested several combinations of parameters to make it fails, but there were no fails. In theory, it is not DRF because after it reads a value in its "if" statement, it is possible that a thread might change it. However I was not able to get this situation.

Results

To get more informative statistics, I created a python script called "stats.py" that is in the project folder. It executes the script several times given the specified parameters and gets the number of times there were errors in the execution, the mean time of all executions and standard deviation. The standard deviations are ommited here for breviety, but the script can be run for more information. For the results here, each experiment was run 10 times. Tables 1 ,2 and3 present the results of the script for 2, 4 and 8 threads. The input array for all of them is initialize with all elements equal to 50 while the maximum is 100. This was chosen this way so in none of the executions we get deadlocks. Deadlocks in this program occurs when there are so many values set wrong that the entire array ends up being either 0 or the maximum value. For table 1 Number of swaps is 100K while input array has length 10. Table 2 has nput array of length 10 and number of swaps of 1M. Table 3 presents results for 100K swaps and array of length 1000.

	2 Thread		4 Threads		8 Threads	
	Mean	**Errors**	**Mean**	**Errors**	**Mean**	**Errors**
Null	533.8	0	887.3	0	2057.1	0
Synchronized	681.8	0	1917.5	0	4520.3	0
Unsynchronized	671.5	10	866.4	10	1700.3	10
GetNSet	958.8	10	1782.8	10	3155.2	10
BetterSafe	1224.5	0	1826.8	0	3784.4	0
BetterSorry	649.4	0	1470.8	0	2594.4	0

Table 1: Summarization of data for 100000 swaps, input array of length 10, maximum value 100 and each element of the array initialized to 5

	2 Thread		4 Threads		8 Threads	
	Mean	**Errors**	**Mean**	**Errors**	**Mean**	**Errors**
Null	1400.6	0	3450.1	0	6930.5	0
Synchronized	1583.0	0	4307.6	0	8623.8	0
Unsynchronized	1524.9	10	3569.4	10	6884.2	10
GetNSet	2426.1	10	5676.7	10	10823.6	10
BetterSafe	2866.5	0	4709.4	0	9859.1	0
BetterSorry	1951.8	0	4717.4	0	8707.8	0

Table 2: Summarization of data for 1M swaps, input array of length 10, maximum value 100 and each element of the array initialized to 50

Conclusion

The speed-up between implementations depends on the input data and the number of threads. For 8 threads and the configuration of table 1, we see that the performance is in decreasing order: Null>Unshynchronized>Better-Sorry>GetNSet>BetterSafe>Synchronized. However, that does not hold true for 2 or 2 threads. This happens because the overhead of some of the implementations weighs more when there are few threads.

Overall, the best choice for GDI's application seem to be BetterSorry implementation. It prevents good performance and it never deadlocks and, even though it is theoretically not DRF, I could not make it fail.

	2 Thread		4 Threads		8 Threads	
	Mean	**Errors**	**Mean**	**Errors**	**Mean**	**Errors**
Null	590.1	0	883.9	0	1587.3	0
Synchronized	960.3	0	2678.1	0	4896.3	0
Unsynchronized	654.0	10	1221.0	10	2248.8	10
GetNSet	951.9	10	1391.7	9	2241.3	10
BetterSafe	1557.7	0	1978.3	0	4064.5	0
BetterSorry	846.9	0	1299.8	0	2025.3	0

Table 3: Summarization of data for 100K swaps, input array of length 1000, maximum value 100 and each element of the array initialized to 50