Vivek Sivakumar
PID: 303652195

Project 3 Write Up

All tests were done running the following command with the different swap states and 10 or 25 threads.

- java UnsafeMemory <Synchronized> <10> 10000 10 2 4 3 8 9 6 0 1 7 5

These are the results from one sequential round of testing. I would often see different results when I tested at different times, but I feel these results do represent the general trend I saw.

	Run #	**10 Threads**		**25 Threads**	
Null	1	4080.03		10944.5	
	2	3355.24		9002.08	
	3	3445.47		10502.0	
	4	3383.92		11054.2	
	5	3089.89		10783.4	
	AVG/STDEV	3470.91	366.67	10457.24	839.59

Null is just the plain test harness run with no swap state, it does nothing so it is DRF and runs very quickly.

	Run #	**10 Threads**		**25 Threads**	
Synchronized	1	5192.46		14856.00	
	2	5354.34		15182.20	
	3	5280.89		14567.10	
	4	5165.88		14648.00	
	5	5202.95		14457.50	
	AVG/STDEV	**5239.30**	**77.25**	**14742.16**	**285.98**

Synchronized uses the standard Java synchronization scheme on the swap method. This utilizes Java's memory model and the intrinsic locking functionality, so that threads can perform swap without stepping on each other. This method is DRF so it has 100% reliability. In this case with a relatively small array size and 10k swaps the performance was reasonably good. The performance was also very steady between runs even when moving to 25 threads.

Unsynchronized	1	3853.52		3120.19	
	2	4217.77		4990.71	
	3	3858.99		4812.97	
	4	4015.03		2923.61	
	5	3911.79		2579.77	
	AVG/STDEV	**3971.42**	**152.22**	**3685.45**	**1128.88**

Unsynchronized makes no attempt to support threads, the code is exactly the same as the code for a singly-threaded swap program. This method is obviously not DRF and fails on basically every input (the input I was using to check performance metrics fails very often). Since it has no locking overhead or extra volatile data structures its execution time is about the same regardless of the number of threads. This means an extreme speed up at 25 threads when compared to synchronized. Unsynchronized does completely stall on occasion, I think this is due to the rampant race conditions and a situation where none of the elements in the array are greater than 0 or less than maxval but all swaps have not been performed yet.

GetNSet	1	5647.40		12936.50	
	2	5957.88		5853.30	
	3	6129.63		12807.20	
	4	5672.59		9049.86	
	5	11193.80		22598.00	
	AVG/STDEV	**6920.26**	**2397.46**	**12648.97**	**6289.39**

GetNSet uses Java's AtomicIntegerArray structure to store the byte array. For this implementation to work I had to instantiate the array and transfer values over from the byte array into the AtomicIntegerArray. I also had to convert back from the AtomicIntegerArray to to a byte array for the current method. The AtomicIntegerArray structure means that only one thread can access it at a time so individual operations on the array are thread safe. This method is not DRF however because it reads the data stores it to a local variable then writes back to the array, meaning all sorts of race conditions are possible between the read and the write back. GetNSet fails pretty often and my default input yields a fail regularly. GetNSet performs similarly to Synchronized, if the swap method involved more complex calculations I'd expect GetNSet to out perform it because the threads are only competing to read and write from the array instead of an intrinsic lock on the whole method.

BetterSafe	1	5469.48		12705.60	
	2	3565.20		10418.00	
	3	4792.64		8571.26	
	4	4645.48		12231.20	
	5	4424.16		27529.60	
	AVG/STDEV	**4579.39**	**688.26**	**14291.13**	**7578.46**

I felt the best way to achieve 100% reliability in BetterSafe was to use locks, mimicking the intrinsic locks the JVM uses for synchronized methods.

My initial intuition behind BetterSafe was to eliminate the time wasted as threads sit idle hoping to acquire the lock. Since the program is swapping 2 random elements in the array a thread (thread 1) should not have to wait for the lock if the thread that has it (thread 2) is not modifying the elements it (thread 1) plans to swap. To accomplish this I tried an array of Java ReentrantLocks in a while loop, where I try the locks at the same positions of the integers that it will modify. If the thread can't acquire one of the locks it releases any locks it has and tries again.

While this implementation was DRF and seemed intuitively faster it had terrible performance (almost 10x worse at 10 and 25 threads), I expect that the sheer overhead of setting up N locks and constantly locking/unlocking them was dominating the execution time.

I eventually modified the code to just have one lock, but instead of using the blocking lock() function I kept the non-blocking tryLock() in a while loop. For some reason this managed to run slightly faster than synchronized, maybe because it wasn't using a blocking lock(). This implementation is DRF because it utilizes a locking scheme around all the code in swap (it locks out all threads except the one with the lock from performing swap).

BetterSorry	1	4388.65		4224.97	
	2	4063.26		4042.43	
	3	4100.77		5454.01	
	4	4595.97		3787.63	
	5	3942.19		3666.41	
	AVG/STDEV	**4218.17**	**267.27**	**4235.09**	**715.30**

My BetterSorry implementation is a combination of GetNSet and my original implementation idea for BetterSafe. Instead of having one AtomicIntegerArray that only one thread can access at a time, I use an array of AtomicIntegers that can each only be accessed by one thread at a

time. This way if 2 threads are looking to use swap on 2 different sets of indices they can run in parallel without colliding. This makes the implementation faster than Synchronized, GetNSet, and BetterSafe, its performance is actually quite close to Unsynchronized. I also utilized the built in getAndDecrement and getAndIncrement functions instead of getting then assigning to a local variable then adding and finally storing, this small change also may have boosted performance. BetterSorry was very reliable in tests but it is not DRF, the potential for race conditions occurs in between the check for valid indices and the increment and decrement portions. If two threads enter swap with a common input the following sequence could mean race conditions for the input: java UnsafeMemory BetterSorry 10 10000 2 1 1 1

thread 1 (1,1) checks bounds → thread 2 (1,1) checks bounds → thread 1 swaps (0,0) → thread 2 swaps (-1,0)

I wasn't able to produce the error when I ran this, and I think it's because the swap method completes too quickly before another thread can come in with the same input.

Looking at the 5 implementations, Unsynchronized and GetNSet won't work for GDI because they introduce race conditions too often. BetterSafe and Synchronized are very similar implementations for this small amount of code and since BetterSafe seems to perform better than Synchronized (most of the time) it looks like the better 100% reliable option for GDI. I think BetterSorry is ultimately the best implementation for the simulations because it has really strong scalable performance and prevents the majority of race conditions that plague Unsynchronized.