CS131 HW3
Alan Covarrubias
103996981

Hardware and Software Info

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)

32 processors total

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 62
model name : Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
stepping : 4
microcode : 0x428
cpu MHz : 2299.921
cache size : 20480 KB
physical id : 0
siblings : 16
core id : 0
cpu cores : 8
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm ida arat epb pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms xsaveopt
bogomips : 3999.68
clflush size : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:

MemTotal: 65759080 kB
MemFree: 36923848 kB
MemAvailable: 64456080 kB
Buffers: 415032 kB
Cached: 25445504 kB
SwapCached: 0 kB

Active: 10615540 kB
Inactive: 15411412 kB
Active(anon): 226472 kB
Inactive(anon): 89280 kB
Active(file): 10389068 kB
Inactive(file): 15322132 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 20479996 kB
SwapFree: 20479996 kB
Dirty: 104 kB
Writeback: 0 kB
AnonPages: 166484 kB
Mapped: 90044 kB
Shmem: 149348 kB
Slab: 2326320 kB
SReclaimable: 2158904 kB
SUnreclaim: 167416 kB
KernelStack: 8736 kB
PageTables: 15228 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 53359536 kB
Committed_AS: 686312 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 390044 kB
VmallocChunk: 34325793276 kB
HardwareCorrupted: 0 kB
AnonHugePages: 8192 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
DirectMap4k: 144320 kB
DirectMap2M: 5052416 kB
DirectMap1G: 61865984 kB

Question 7:

Running Synchronized and Null with different parameters:

3 test cases with 5 trials each

1.
java UnsafeMemory Synchronized 2 10000 6 5 4 3 2
1607.85 1406.09 2078.93 2074.26 1366.59 ns/transition

java UnsafeMemory Null 2 10000 6 5 4 3 2
1477.79 1598.35 1470.32 1593.52 954.569 ns/transition

2.
java UnsafeMemory Synchronized 8 10000 6 5 4 3 2
8733.64 8604.09 8884.84 8523.86 8722.83 ns/transition

java UnsafeMemory Null 8 10000 6 5 4 3 2
7116.53 6325.43 6265.84 8685.69 6263.12 ns/transition

3.
java UnsafeMemory Synchronized 8 100 12 5 4 3 2 0 12 11 9
213499 240771 211242 194246 191063 ns/transition

java UnsafeMemory Null 8 100 12 5 4 3 2 0 12 11 9
247367 166094 221699 216647 204529 ns/transition

Both Null and Synchronized pass all test cases. On average Null performs better than Synchronized. This is most likely because Null performs no actual work. Synchronized did not perform significantly slower than Null despite its workload.

Question 11
BetterSafe is more efficient on average than Synchronized because rather than using synchronized methods which forces threads to wait for any thread running the swap function to finish executing before running swap on its own, BetterSafe only locks the reads and writes for variables from other threads. It uses an intrinsic lock of the State object to block multiple threads from incrementing the byte array, but that is the only section of the swap function that is locked. This reduces the size of the critical section by following the Goldilocks principle.

Question 13
BetterSorry is more efficient than BetterSafe because it allows interleaving of operations that increment or decrement different memory values by using two separate locks, one for each byte array index, rather than just one for both indices. This allows thread to increment one memory index while another thread increments a different address. This was not allowed in BetterSafe. An issue with BetterSorry occurs when one thread's i index is the same as another thread's j index variable. The program will think that these two indices point to separate addresses in the byte array, which it normally does, but not for these rare cases. When the byte array is small, the chances of this occurring increases dramatically. One test case that is likely to fail for BetterSorry:
java UnsafeMemory BetterSorry 4 1000 3 2 1 1 2

Question 15

Test Case
State
Speed
Reliability

java UnsafeMemory State 4 1000 3 2 1 1 2
Null
15807.6 18397.2 16286.2 15904.3 14338.3 ns/transition
Synchronized
19488.2 15774.5 17086.2 15662.8 16512.5 ns/transition
Unsynchronized
16834.7 15739.2 16266.7 16979.9 16447.9 ns/transition
6 != 7 6 != 7 6 != 8 6 != 7 6 != 7
GetNSet
Code Unreliable
BetterSafe
18003.7 14021.2 16364.8 15917.5 14163.4 ns/transition
BetterSorry
17923.9 15119.6 16186.6 15850.8 17029.3 ns/transition
6 != 3 6 != 4 6 != 4 6 != 4 6 != 4

java UnsafeMemory State 8 1000000 6 5 6 3 0 3
Null
2127.43 2291.40 2285.22 2186.73 2236.64 ns/transition
Synchronized
2837.09 2802.59 3021.03 2450.02 3282.45 ns/transition
Unsynchronized
Code Unreliable
GetNSet
Code Unreliable
BetterSafe
2990.74 2687.79 2783.24 3076.71 2988.40 ns/transition
BetterSorry
Code Unreliable

java UnsafeMemory State 2 1000 8 4 3 6 5 7 2 1 3 4
Null
6233.72 6036.45 6585.14 6254.63 6235.39 ns/transition
Synchronized
6548.34 5981.65 5722.93 5982.49 6020.52 ns/transition
Unsynchronized
6523.56 6720.13 7256.10 6758.24 6768.32 ns/transition
35 != 33 35 != 41 35 == 35 35 != 33 35 != 35
GetNSet
9804.92 8991.93 11294.5 9694.23 9187.19 ns/transition
35 != 50 35 != 36 35 != 38 35 != 48 35 != 45
BetterSafe
5558.80 6530.66 6181.35 5954.91 5285.06 ns/transition
BetterSorry
4118.79 4262.72 5505.99 3903.62 4032.98 ns/transition
35 == 35 35 != 36 35 == 35 35 == 35 35 != 34

Discussion:

Null and Synchronized:
My test cases show that Null and Synchronized ran at about the same speed. In some cases, Synchronized sometimes surpassed Null's speed surprisingly. This might be due to how busy the processors in SEASNET were at the time.

Unsynchronized:
Unsynchronized always produces incorrect results due to data races which is expected. It runs slower than Null and Synchronized, and is also very unreliable. Any test case would cause this class to produce inaccurate results.
Ex. java UnsafeMemory Unsynchronized 2 1000 8 4 3 6 5 7 2 1 3 4

GetNSet:
GetNSet uses an AtomicIntegerArray to synchronize the instructions of the swap function. There is an issue with concurrency with this implementation due to the fact that although we are accessing the variables in the array atomically, the order in which the values are accessed, updated, and stored is still random. This implementation fixes the memory consistency errors that occur with multithreading, but not thread interference. GetNSet is about as inaccurate as Unsynchronized. Any test case would cause this class to produce inaccurate results.
Ex. java UnsafeMemory Unsynchronized 2 1000 8 4 3 6 5 7 2 1 3 4

BetterSafe:
BetterSafe improves performance at 100% reliability as expected. The use of synchronized locks instead of making the entire function synchronized locks gives the threads more freedom by shrinking the critical section of the swap function.

BetterSorry:
BetterSorry improves performance even higher than BetterSorry but at the cost of errors on certain rare cases where separate threads using separate locks are pointing to the same memory address by chance thus causing a data race which could produce inaccurate results. Despite this design flaw, allowing each index to have its own separate lock allows two threads to access memory variables at the same time which increases concurrency thus improving performance. A test case that it is likely to fail is one where there is a small sized array which means the likelihood of the two locks to have the same index is high.
Ex. java UnsafeMemory BetterSorry 4 1000 3 2 1 1 2

Based on these results, BetterSorry is the best option for GDI's application. It improves performance more than BetterSafe's implementation, while creating some rare inaccuracies. This is acceptable because we are trading reliability for performance as the GDI assigned us to do.

An issue that I found that I cannot explain is that sometimes when an unreliable state is used on a test case with many threads and many transitions, no calculated speed is output unless I place a print to screen function in order to slow the processor down. This multithreading bug goes to show how obscure bugs in multithreaded code can be when slowing down a processor by placing a print function can change whether code returns successfully or not.