HomeWork 4: CS 131

Array Used : **5 6 3 0 3 5 6 7 8 9 0 2 3 4 5 6 7**
MaxVal: **18**

Test	Threads: 8 Swaps: 1M	Threads: 16 Swaps: 1M	Threads: 8 Swaps: 10M	Threads: 16 Swaps: 10M	Threads: 32 Swaps: 1M	Reliability
Synchronized	3559.49	6119.13	3726.45	4953.07	9805.67	100%
Null	679.126	1924.75	67.6684	195.970	8074.62	N/A
BetterSafe	2141.16	4100.47	1473.17	3476.28	8288.60	100%
BetterSorry	1490.38	2880.72	1187.93	1722.92	7739.52	Less than 100%
GetNSet	2293.22	3704.88	1647.26	2831.26	8581.74	Less than 100% Less than BetterSorry

Note: All results are averages over 5 runs of the test harness

BetterSafe :

My BetterSafe Implementation restricts the multithread access on increment and decrement only. Previously all the statement in the swap function is locked but in BetterSafe only the instructions involved in incrementing/decrementing the array are synchronized. With this approach a lot of the cases where the write is not needed, the thread need not to wait for the other threads to free the lock. These threads can read and return along with other threads where increment/decrement is not required. This implementation saves a lot of time in the case where even read only threads have to wait too in the previous implementation. Eliminating this read only waiting time makes BetterSafe faster. I am also maintaining an array of locks for each element of array so that the threads do no have to wait if other threads are not working on the elements assigned to it. Moreover the byte array is also volatile to make sure that the other threads write happens actually before reading.
This is DRF because the increments and decrements operations are wrapped inside synchronized.

BetterSorry :

BetterSorry implements all the features of the BetterSafe and over that it saves time at the point where threads wait to write back into array in the previous implementation. Now since we have flexibility of let go of some reliability, we can concurrently write on the array with out letting the threads to wait for others to finish write. Implementation is same as GetNSet except its more reliable in the case where the increment is happening. Better Safe is more reliable than Unsynchronized in general. Since BetterSorry applies synchronize on read, incrementing same read twice is not possible which was possible in Unsynchronized.
But BetterSorry will suffer from the lack of DRF. Let's say the two threads increment the same element in the array sequentially. But since write is not synchronized the write of the later increment thread might happen first. While the value should be +2 it is actually +1, which will cause the mismatch in the sum of the array.

BetterSorry Test:

Failing BetterSorry totally depends on the interleaving of the multiple threads. Designing a test case to make it fail would be extremely difficult since we cannot control the thread level interleaving. One

thing that makes it probably for better sorry to fail is to run maximum amount of threads with large number of transactions on smaller array size.
Array size : 4
Transaction : 100M
Threads: 32

Problems While Testing:

The only thing that was problematic is the deadlock when all the elements in the array are either 0 or max value. In that case the swap goes into infinite loop and does no return. This is actually not a lock based deadlock instead a logical error which can be avoided with out effecting the thread implementation.

Sample Runs:

You can use the following command to run test harness. The reliability can also be tested through this command.
> java UnsafeMemory "Model" 8 10000000 18 5 6 3 0 3 5 6 7 8 9 0 2 3 4 5 6 7

Best Choice :

BetterSafe is reliable and fast and because BetterSorry's reliability is significantly low although it is faster.

Specifications:

Version:
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)

Processors: 12
processor : 0
vendor_id : GenuineIntel
cpu family: 6
model : 44
model name : Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
stepping : 2
cpu MHz : 1596.000
cache size : 12288 KB
physical id : 1
siblings : 8
core id : 0
cpu cores : 4
apicid : 32
initial apicid : 32
fpu : yes
fpu_exception : yes
cpuid level : 11
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 xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt aes lahf_lm ida arat epb dts tpr_shadow vnmi flexpriority ept vpid
bogomips : 4788.05
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management:

Memory :

```
MemTotal:       32865604 kB
MemFree:         4306760 kB
Buffers:          466320 kB
Cached:         23368580 kB
SwapCached:         4460 kB
Active:          3711624 kB
Inactive:       20152364 kB
Active(anon):      21924 kB
Inactive(anon):     7232 kB
Active(file):    3689700 kB
Inactive(file): 20145132 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:      20479992 kB
SwapFree:       20470264 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:         25160 kB
Mapped:            10192 kB
Shmem:                60 kB
Slab:            4457232 kB
SReclaimable:    4352860 kB
SUnreclaim:       104372 kB
KernelStack:        2912 kB
PageTables:         5180 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    36912792 kB
Committed_AS:     142164 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      335492 kB
VmallocChunk:   34342227032 kB
HardwareCorrupted:     0 kB
AnonHugePages:      2048 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:        4580 kB
DirectMap2M:     2082816 kB
DirectMap1G:    31457280 kB
```