CS131 HW4 Report

Why BetterSafe is faster than Synchronized and is still 100% reliable?

1. Use of ReentrantLock

ReentrantLock provides much better performance under heavy contention. If we use ReentrantLock, when many threads are attempting to have access to the same resource, JVM will spend less time scheduling.

Why JVM can spend less time scheduling? I think this lies in the difference of mechanisms used in reentrant lock and synchronized. In reentrantlock, a thread is allowed to obtain a lock for multiple times, with an acquisition count index. As long as the thread can release the lock for such many times, there will be no problem. In contrast, a thread cannot obtain a lock for multiple times in synchronized. This means that synchronized has to release the lock first to obtain another new lock.

To further explain this in practical sense: in the example, it is allowed to change the value/do the if condition check for the multiple times first in reentrantlock, while for synchronized, you have to do these one by one; that is, you have to acquire the lock first, execute the command and then release the lock. In contrast, if we use reentrantlock, we can obtain lock recursivley, and release lock recursively.

This mechanism should have sped up the execution process and give it better performance.

Why BetterSorry is faster than BetterSave?

1. Use of AtomicIntegerArray functions

Instead of using locks, I choose to use AtomicIntegerArray functions GetAndIncrement and get AndDecrement. They are faster for the following reasons: 1.they are built at lower level, supported by CPU. 2.They don't have the time-consuming lock-gaining and lock-releasing process.

Reliability and Performance

BetterSorry

Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory BetterSorry 10 100000000 6 5 6 3 0 3 6 5 6 3 0 3
Threads average 599.689 ns/transition
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory BetterSorry 10 100000000 6 5 6 3 0 3 6 5 6 3 0 3
Threads average 606.016 ns/transition

DRF? yes

BetterSafe

Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory BetterSafe 10 100000000 6 5 6 3 0 3 6 5 6 3 0 3
Threads average 715.544 ns/transition
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory BetterSafe 10 100000000 6 5 6 3 0 3 6 5 6 3 0 3
Threads average 635.319 ns/transition

DRF? yes

GetNSet

java UnsafeMemory GetNSet 10 100000000 6 5 6 3 0 3 6 5 6 3 0 3

Program hangs.

```
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory GetNSet 10  100000 6 5  3 0 3 6 5 6 3 0
3
Threads average 4173.45 ns/transition
sum mismatch (34 != 25)
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory GetNSet 10  100000 6 5  3 0 3 6 5 6 3 0
3
Threads average 4870.04 ns/transition
sum mismatch (34 != 9)
```

DRF? no

Failing test: test cases listed above is sufficient.

Unsynchronized

```
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory Unsynchronized 10  100000000 6 5 6 3 0
3 6 5 6 3 0 3
```

Program hangs.

```
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory Unsynchronized 10  1000000 6 5 6 3 0 3
6 5 6 3 0 3
Threads average 814.870 ns/transition
sum mismatch (40 != 37)
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory Unsynchronized 10  1000000 6 5 6 3 0 3
6 5 6 3 0 3
Threads average 738.225 ns/transition
sum mismatch (40 != 49)
```

DRF? no

Failing test: test cases listed above is sufficient.

Synchronized

```
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory Synchronized 10 100000000 6 5 6 3 0 3
6 5 6 3 0 3
Threads average 938.457 ns/transition
Cs-210-92:homework4 Nick_Guan$ java UnsafeMemory Synchronized 10 100000000 6 5 6 3 0 3
6 5 6 3 0 3
Threads average 923.704 ns/transition
```

DRF? yes

Which model fits GDI's application the most?

BetterSorry. The reasons are following: Despite we don't expect BetterSorry to be that reliable compared to BetterSafe, its performance is better than BetterSafe. Also, I believe use of the atomic functions also guarantees its reliability, but I am not sure whether it is 100% or not. Anyways, BetterSorry fits GDI's application the most.