# UCLA CS 131 Programming Language
# Homework 3 Report

**Name:** Tianyang Zhang

**UID:** 404743024

**Server:** lnxsrv07

**Java Version:** 9.0.1+11

**CPU:** Dell PowerEdge 6650

2.8GHz quad processor

Oct. 24th. 2017

# 1 Pros and Cons of Packages

## 1.1 java.util.concurrent

The java.util.concurrent package provides a classic concurrency tool – Semaphore.
The advantage of Semaphore is that we can use it to control the number of how many threads could get access to the part of code.
However, the performance of Semaphore is about same as using Synchronized keyword, sometimes even a little bit slower.
Therefore, I didn't choose to use this package to implement BetterSafe.

## 1.2 java.util.concurrent.atomic

The atomic package implements concurrent in hardware-level using CAS(Compare and Swap) instruction of CPU. The advantage of atomic package is that it has a high performance on dealing with single sharing variable.
However, as the number of threads increasing, or say as the concurrency increasing, the failure of CAS will increase. If the CAS cannot successfully done in a long time, the cost on CPU would be incredibly high. CAS also cannot guarantee the atomic operation success when dealing with multiple sharing variables, thus it may leads racing condition in multiple sharing variables program.
In BetterSafe, using Atomic will leads racing condition, and thus I didn't choose this package.

## 1.3 java.util.concurrent.locks

Reentrantlock provides a high performance on multi threading programs with multiple sharing variables. The biggest advantage of Reentrantlock is the high speed performance, it has about 2.5 times better performance than Synchronized and Atomic in multiple sharing variables program. Reentrantlock also provide the fairness option but we don't nee to use it in this assignment.
There are also several cons of Reentrantlock. When using Reentrantlock, we have to put the .lock() and .unlock() option correctly, it is easier to have dead lock comparing to Synchronized and Atomic due to incorrectly lock or unlock. In this assignment, Reentrantlock satisfies the needs of BetterSafe the most.
Because of the high performance and high reliability, I chose Reentrantlock to implement BetterSafe.

## 1.4 java.lang.invoke.VarHandle

The using of VarHandle in multi-threading is similar to Atomic. VarHandle could use volatile access to variables. Therefore it has the same problem as Atomic – unreliable when handling multiple sharing variables. Therefore I didn't choose VarHandle to implement BetterSafe.

# 2 Performance of BetterSafe vs. Synchronized

BetterSafe is faster than Synchronized since it is implemented by Reentrantlock. Reentrantlock is designed to handle high concurrency multi-threading program with better performance than Synchronized keyword. Therefore, Reentrantlock is more fit in this assignment than Synchronized, hence BetterSafe it is faster.

# 3 Difficulty and Reliability

## 3.1 Problems during measurement

My first idea during measuring is to write a simple shell script that repeated call UnsafeMemory and gathering the output numbers. However I found that I couldn't deal with the Unsynchronized and GetNSet class since they would run into infinite loop. Therefore I wrote the shell script for Synchronizedstates and BetterSafe only, which called each of them 40 times with different numbers of threads and transitions

then collected the out put of transition speed and calculate the average of them.

### 3.2 Reliability of each class

**SynchronizedState:**

This class used Synchronized keyword in it's swap() method. The Synchronized keyword make sure that only one thread could access the swap() method, therefore, it prevented the happening of racing condition. Thus, this class is DRF.

**Unsynchronized:**

This class doesn't use any technic to prevent racing condition. Multiple threads may access the swap() method at the same time. For example, value[0] = 0 at a certain time, at the time Thread1 executed value[0]--,Thread2 may just finished the check of if-statement, although the value of value[0] already became -1, which the swap() method did something it not suppose to. Thus, this class is not DRF.
The result of reliability test of Unsynchronized is almost 0%, I tried to run the test 40 times but none of them successfully done.

**GetNSet:**

This class has the similar problem with class Unsynchronized. It changes the array of byte to AtomicIntegerArray so it can use the atomic functions get() and set() to access the data in array. However, it can only make sure one thread can access one single chosen element in get() or set() at the same time. Since the program will involve both get() and set() as two different step, one thread may accessing data by set() at the same time when another thread is accessing data by get(). Thus, it is also not DRF.
The reliability test of GetNSet also shows a nearly 0% on GetNSet's reliability.

**BetterSafe:**

BetterSafe is implemented by Reentrantlock, which setup a lock at the begin of swap() method. The lock at the begin of swap() could make sure only one thread can execute the code below it, and all other thread which reaches that line later than the first thread have to wait until the unlock() triggered. The lock will be unlocked after the all the data access and before function returns. The lock prevent racing condition, and thus, BetterSafe is a DRF class.

## 4 Performance Test

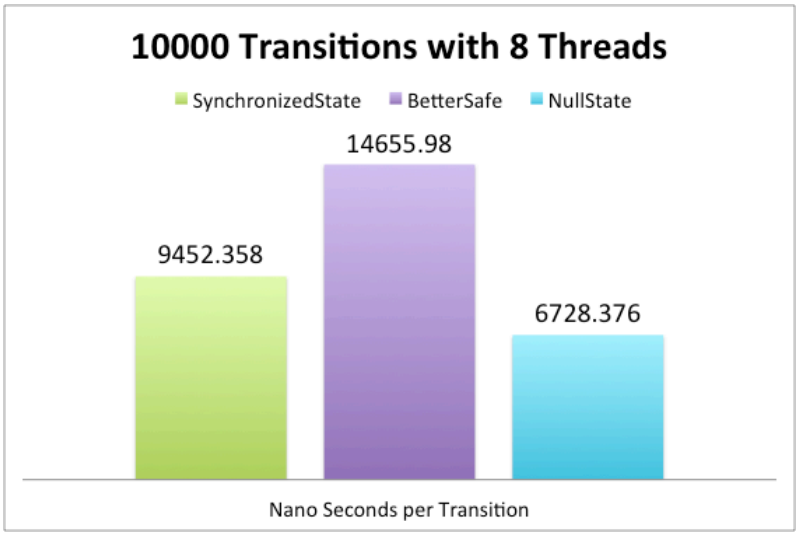
### 4.1 Interesting fact on performance test

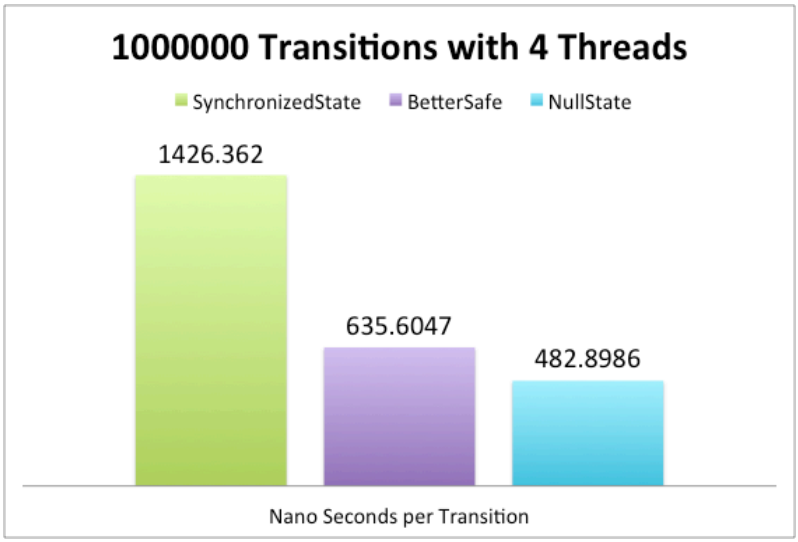
One confusing but interesting fact is that when I testing the performance on lnxsrv07, the time cost per transition of Null is more than BetterSafe when testing with 8 threads and 1000000 transitions. And as the number of threads decreasing, the relevant time cost per transition of Null becomes less than BetterSafe. However, when I test on my own laptop, the time cost per transition of Null is always less than BetterSafe. My laptop has following hardware configuration:
CPU: 2.5 GHz Intel Core i5
Memory: 8 GB 1600 MHz DDR3
Java Version: 9.0.1+11
My guessing on this fact is that this problem may caused by the difference of processor, since the way they run multi-threading program may in different way and configuration.

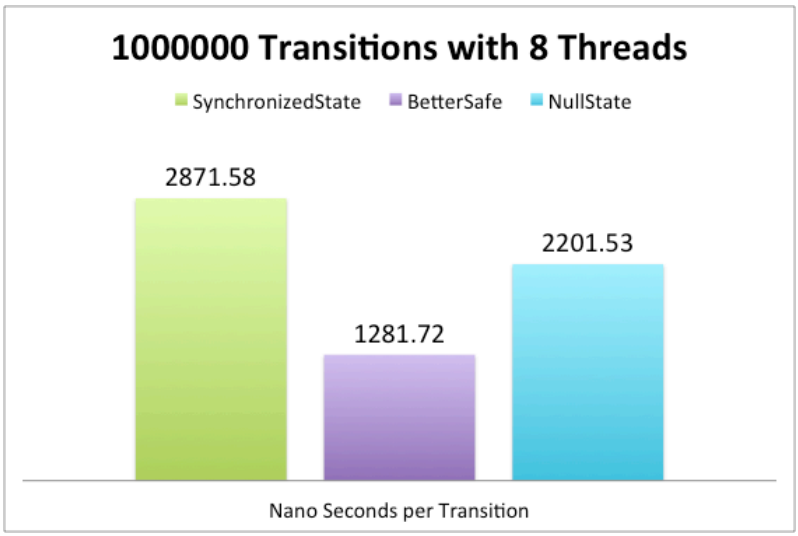
### 4.2 Performance

As the graph showing, BetterSafe performs bad when dealing with small transitions, but the performance is increasing in a high rate as the workload of each thread grows.

*The Transition & Threads vs. Average ns/transition graph is showing below in the next pag




**Reference:**

https://docs.oracle.com/javase/9/docs/api/overview-summary.html