Name: Chen Chen

UID: 004710308

Dis 1C

I. PACKAGES

A. *java.util.concurrent*

This package provides five different Synchronizers.

1) Semaphore: Semaphore is a variable used to control the number of threads which are accessing public resource in concurrent system.
Pros: Semaphore can be used to specify exactly of number of thread which are accessing public resource in concurrent system or multip-threads ironment.
Cons: In my lab, the public resource only allow one thread to access when executing the program. So even I implement Semaphore in my program, it will work like a lock. As a result, I can use a lock instead.

2) CountDownLatch: CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
Pros: Can be used to used to control the starting time of set of threads or one threads
Cons: In my lab, it is not the correct scenario to use.

3) CyclicBarrier: CyclicBarrier is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
Pros: Can be used to solve Producer-consumer problem or Bounded-buffer problem.
Cons: In my lab, it is not the correct scenario to use.

4) Phaser: Phaser is a reusable synchronization barrier, similar in functionality to CyclicBarrier and CountDownLatch but supporting more flexible usage.
Pros: compared to CyclicBarrier, phaser can be used to block execution during each round.
Cons: In my lab, it is not the correct scenario to use.

5) Exchanger: is a synchronization which allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs.
Pros: Allow two seperate object to communicate.

Cons: In my lab, it is not the correct scenario to use. I don't need a Exchanger to exchange data between threads.

B. *java.util.concurrent.atomic*

This package provides a small toolkit of classes that support lock-free thread-safe programming on single variables similar with the usage of Syntax *voliate*. Classes implemented under this package will read directly from memory not from the CPU's cache.

Pros: It provides a lock-free and thread-safe soluation which can allow one or none thread to write public resource and unlimited thread to read public resource.

Cons: If mul-threads will write public resource which is using this package, this package can't

promise each thread is reading or writing on correct value. It will cause race condition. In my lab, each threads will read and write public resource.

C. java.util.concurrent.locks

This package provides a lock is a more flexible and sophisticated thread synchronization mechanism than the standard synchronized block.

Pros: Compared to synchronized block which going to block entire execution scope, this package can use a lock to block certain execution not entire scope.

Cons: It still block the execution of other threads. It will slow down the execution time of entire program.

D. java.lang.invoke.VarHandle

This package is a dynamically strongly typed reference to a variable.
Pros: compared to *voliate*, this package still can do plain read/write access and provides a extra protection which onle can be accessed under access mode.
Cons: This package still going to cause race condition similar with java.util.concurrent.atomic and *voliate*.

II. BETTERSAFE

My BetterSafe class uses the third package java.util.concurrent.locks. As I introduced above this package provides a flexible usage of synchronized block. It is a optimization of Synchronized class. In my implementation, I only block the parts will cause race condition. So it definitely will generate shorter the execution time of program as the same time guarantees the reliablility of the swap();

III. PROBLEMS

I misunderstood the meaning of maxval. I thought it is the length of the arrary, but It represents the maxval of the arrary. I tried to return maxval in the implementation of function size(). It caused me long time to solve this bug.

IV. ANALYSIS DATA

A. Hardware and Java Virtual Machine info

java version ”9.0.1”
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
CPU:Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz

B. Lab

I try 1,000,000 times of swap with 8/16/32 threads and each execute 30 times and calculate the average.

Class	Time(ns/transition)			DNF
	8 threads	16 threads	32 threads	
Synchronized	801.664	1729.62	4081.98	Yes
Unsynchronized	370.224	861.47	1836.233	No
GetNSet	541.984	1226.70	2546.53	Yes
BetterSafe	565.048	1240.67	2804.96	No

C. Comparison

The Unsynchronized and GetNSet involve read and write operations on each threads without using the

lock. As a result, it will cause race condition. However, because of not using the lock, the time cost on each transitions is considerable small. Because the usage of lock and synchronized, the time cost on each transitions is considerable lager for the class BetterSafe and Synchronized. For my suggestion based on my environment, I suggested GDI use the BetterSafe. Bacause it guarantees the reliablility and the execution time is slightly slower than GetNSet.