CS 131 HW3 Report

1.Abstract

The aim of the project is to test performances of different implmentation of a wrap functions substitute for synchronized.

2.Test Environment

Server: linux07.seas.ucla.edu
Java 9.0.4
(build 9.0.4+11)

3.Performance Analysis

using maxval=100
1000000 wraps

Function	Average Time ns/transition			Reliability
	4-thread	8-thread	16-thread	
NULL	496.2	2482.7	4130.03	NP
Synchronized	1196.3	2434.46	5391.05	NP
Unsynchronized	476.889	1072.83	4082.72	mismatch
GetNSet	714.512	1869.89	3952.21	NP
BetterSafe	555.405	1069.19	2674.14	NP

Synchronized:
Synchronized has the worst performance here but is guaranteed to be DRF as it uses the synchronized keyword in Java which prevents the racing condition when multithreading.
Unsyncrhonized:
Unsyncrhonized has the best performance among four classes but this is not DRF as the sum is incorrect every time thanks to the racing condition between different threads.
GetNSet:
GetNSet is faster than synchronized and is RDF because it uses AtomicIntegerArray which is also thread-safe but better optimized than synchronized implemntation.
BetterSafe:
BetterSafe is the fastest RDF implementation among these so should be the best choice for GDI's application

4.BetterSafe Choice

java.util.concurrent:
The class provides five Syncrhonizers to use for concurrency, each with different function and features.

Semaphore is a common method to restrict the number of threads than can access some resources. Semaphore can be used as a lock for BetterSafe, but if so the performance is worse than directly using lock because Semaphore is more complicated and powerful than lock.

CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. However, it can only block once,after the count reaches zero, it cannot be reset. For BetterSafe, it needs to keep blocking and unblocking, so CountDownLatch is not qualified.

CyclicBarrier is a resettable multiway synchronization point useful in some styles of parallel programming. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. So it is still not qualified for BetterSafe.

Phaser is a reusable synchronization barrier, similar in functionality to CyclicBarrier so not optical for BetterSafe either.

Exchanger is a synchronization point at which threads can pair and swap elements within pairs. This is not qualified because BetterSafe needs to prevent different threads to modify the same object rather than share information between threads.

java.util.concurrent.atomic:
This package support lock-free thread-safe programming. Instances of classes AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference each provide access and updates to a single variable of the corresponding type, so they are not useful for array elements which is what we need.
The AtomicIntegerArray further extend atomic operation support to arrays of these types and also provides volatile access semantics for their array elements. This is ok for BetterSafe and is used in GetNSet actually, but when tested, the performance is worse than the BetterSafe we choose later.

java.util.concurrent.locks:
This package provides a a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors, so this package is suitable for BetterSafe as it can keep locking and unlocking the array elements.
In the package, I choose the class ReentrantLock as a reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements. So this will work for BetterSafe as it is close to synchronized. When I first tried the class, the performance is better than both Synchronized and GetNSet, so I am settling on this class.

java.lang.invoke.VarHandle:
A VarHandle is a dynamically strongly typed reference to a variable, or to a parametrically defined family of variables. So this can provide thread-safe single operation of a variable but cannot prevent other threads from modifying the value and race condition.

why my BetterSafe is faster and 100% reliable:
The BetterSafe code uses Reentrantlock and calls the lock function at the start of swap function which prevents other threads from modifying the array element. And it only calls the unlock function at the end of swap function after already testing and trying to swap. So there will be only the current working thread that is able to modify the values, which makes the swap function thread-safe and 100% reliable.
There are several reasons why BetterSafe using reentrantlock is faster than Synchronized. First it can have more than one condition variable per monitor rather than only one in synchronized, so it can support more than one wait() queue. Second, it can make the lock “fair” which favors granting access to the longest-waiting thread and makes it more efficient. From IBM website, when many threads are attempting to access a shared resource, the JVM will spend less time scheduling threads and more time executing them.

5.Problem:
The biggest problem I have is that the linux server cannot run thread bigger than 20 due to limitation. When testing the states with more than 20 threads, the server will output “Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached”, so it limits the range of test results.

6.RDF:
I have already talked about RDF for every state in Performance Analysis part.