CDD Lab Sheet 2 - Timing Report

Author: Evin Darling C00144257

Description:
Lab 2 requires timings to be carried out on the submitted solutions. The original code and the mutex, synchronised, and AtomicInteger versions were executed 5 times each while being timed.

The command used to record these timings was:

```
$ time java com.itc.mutexexample.Main
```

Original Project

These are the timings for the original, unmodified code provided for Lab 2:

	#1	**#2**	**#3**	**#4**	**#5**
Time (sec):	2.632	2.583	2.562	2.559	2.558

Min: 2.558
Max: 2.632
Avg: 2.579

Synchronised Project

These are the timings for the synchronised version of the project:

	#1	**#2**	**#3**	**#4**	**#5**
Time (sec):	2.594	2.560	2.556	2.557	2.560

Min: 2.556 (-0.002)*
Max: 2.594 (+0.038)*
Avg: 2.565 (-0.014)*

Mutex Lock Project

These are the timings for the mutex lock version of the project:

	#1	#2	#3	#4	#5
Time (sec):	2.564	2.562	2.604	2.559	2.559

Min: 2.559 (+0.001)*
Max: 2.604 (-0.028)*
Avg: 2.566 (-0.013)*

AtomicInteger Project

These are the timings for the AtomicInteger version of the project:

	#1	#2	#3	#4	#5
Time (sec):	2.566	2.561	2.568	2.566	2.555

Min: 2.555 (-0.003)*
Max: 2.568 (-0.064)*
Avg: 2.563 (-0.016)*

* compared to the original project

	Original	Synchronised	Mutex	AtomicInteger
Min	2.558	2.556	2.559	2.555
Max	2.632	2.594	2.604	2.568
Avg	2.579	2.565	2.566	2.563

Comparisons

The **original** project appears to be the worst-performing, with the highest values for max and average time. This is surprising given the fact that each other project introduces additional functionality in order to solve the race condition.

The **synchronised** project has the `synchronized` keyword added to the `IntegerObj.inc()` method. This allows only one thread to execute that instance method at a time. Synchronised methods are implemented using a reentrant mutex lock. This might explain the similarity between the min and average times of the mutex project, which uses a semaphore limited to one thread.

The **mutex** project uses a semaphore limited to one thread at a time as a mutex lock. The timings are comparable to that of the synchronised project. A one-thread semaphore might be slightly less performant than the reentrant lock implementation of the synchronized keyword, given the sizeable difference in max times, and the slight difference in average times.

The **AtomicInteger** project uses atomic variables and operations to solve the race condition. This project had the lowest times of all. This may imply that usage of atomic variables and operations is more efficient than using mutex locks. While using mutex locks, a thread must wait until the thread currently executing the critical section unlocks. In the case of atomic operations, the critical section code can be looped indefinitely until it is carried out exclusively, as opposed to waiting for an unlock and then executing.

Conclusion

The times for each project were quite similar so it is difficult to come to any solid conclusion. However, it does look like the use of atomic variables and operations is more efficient than using locks. I expected the original project's timings to be faster than that of the lock-based synchronised and mutex projects because of the inclusion of a mechanism that causes additional wait times before threads can execute.