ASSIGNMENT 2

CS 454: Principles of Concurrent Programming / Spring 2022

Description

In this assignment, you will implement a Lock object that allows threads to execute a critical section in mutual exclusion, using the concepts introduced in class. Your lock should extend the following abstract class and implement all abstract methods:

```
public abstract class CS454Lock implements java.util.concurrent.locks.Lock {

    public abstract void lock();

    public abstract boolean tryLock();

    public abstract void unlock();
}
```

Each operation (method) behaves as follows:

- **lock**: Acquires the lock. If another thread already has the lock, this operation blocks waiting until the current thread acquires the lock successfully.
- **tryLock**: Attempts to acquire the lock. If another thread already has the lock, this operation returns `false`. Otherwise, this operation acquires the lock and returns `true`. This operation does not block waiting for the lock.
- **unlock**: Releases the lock.

You should also add your solution for Assignment 1 and modify it to use the lock you implement.

Correctness Requirements

Your implementation should keep the following properties at all times:

1. **Mutual Exclusion:** No two threads can acquire the lock at the same time.
2. **Deadlock Freedom:** If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock (not necessarily the same thread).
3. **Happens-before**: Releasing a lock (i.e., calling `unlock`) happens-before future acquisitions of that same lock (i.e., future `lock`)
4. **Unlock integrity**: Calling `unlock` on an unlocked lock throws the exception `java.lang.IllegalMonitorStateException`.
5. **Unlock validity**: Calling `unlock` on `Thread 1` on a lock that is acquired by `Thread 2` throws the exception `java.lang.IllegalMonitorStateException`.
6. **Reentrancy**: A thread can acquire the same lock while already holding the lock. In this case, the thread must release the lock the same number of times as it acquired the lock for the lock to become unlocked (e.g., `Thread 1` calls `lock` 3 times, it must call `unlock` 3 times for `Thread 2` to be able to acquire the same lock).

Concurrency Requirements

- Your submission should support the properties listed above for an **unbounded number of threads**, not known in advance.
- Your submission should **not have data-races**. Submissions with data-races are considered incorrect, and the points of all tests passed by chance will be deducted from the final grade.

Performance Requirements

Your submission should optimize for the case of a lock under high contention with short lock-unlock periods. For instance, a large number of threads attempting to increment a counter protected by your lock. The autograder has a timeout of 60 seconds for all the tests to run, which is 3x longer than the instructor's implementation takes.

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method `Lab.createLab` and `Lab.getLock` so that it creates an instance of the class you added. Note that `getLock` always returns a new, unused, lock. You cannot change any other part of the code that is provided to you.

```
abstract class Lab {
    static Lock createLock() {
        throw new Error("Not implemented");
    }

    static Lab createLab() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **February 19 2022** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** (50% for graduate students). You can resubmit your assignment until **February 26 2021** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment-2` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 60%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 6 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Together with the code, you should submit 4 video screen-casts (**through Gradescope**) that answers the four questions below by explaining how your code works. The questions focus on concurrency/multi-threading and are worth 10% each. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

1. How does your implementation ensure Property 1 (mutual exclusion) when multiple threads attempt to acquire the same lock at the same time?
2. How does your implementation ensure Property 3 (happens-before)?
3. Explain if your implementation is fair, starvation-free, or neither. See Property 2.
4. Describe 2 steps you took to improve the performance of your implementation.

The maximum length for each video is 2 minutes, instructors will stop watching at the 2 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `System.out.println("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.