CS454 Principles of Concurrent Programming
Assignment 2

Prof Luís Pina — University of Illinois Chicago

Spring 2026

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.

Description

Your lock should extend the following abstract class and implement all abstract methods:

```java
public abstract class CS454Lock implements java.util.concurrent.locks.Lock {
    public abstract void lock();
    public abstract boolean tryLock();
    public abstract void unlock();
    public abstract int isReentered();
}
```

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`. In either case, `tryLock` does not block waiting for the lock.
- `unlock`: Releases the lock.
- `isReentered`: Returns `true` if the lock is currently owned more than once by one thread, `false` otherwise. May be called by any thread, even if the calling thread does not own 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., calling `lock`) by any thread (the same thread that just released the lock or any other thread)
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 a different 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 any other 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 implementation of `isReentered` must be as fast as possible. It should use as few memory operations as possible, as few instructions as possible, and it should use the weakest possible memory operations. Your implementation should remain correct as defined above.
- 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 methods `Yvvid.createYvvid` and `Yvvid.createLock` so that they create instances of the class you added. You cannot change any other part of the code that is provided to you.

```
abstract class Yvvid {
    static CS454Lock createLock() {
        throw new Error("Not implemented");
    }

    static Yvvid createYvvid() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **February 21 2026 (Saturday)** **5pm CT**. 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** (including all bonuses) should be **equal to or higher than 30%** for undergraduate students, and **50%** for graduate students. You can resubmit your assignment until **February 28 2026 (following Saturday)** **5pm CT**. 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 `assignment2` 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 7 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Together with the code, you should submit three video screen-cast (**through Gradescope**) that answers the three questions below by explaining how your code works (one video per question). The questions focus on concurrency/multi-threading and are worth 10% each.

1. How does your implementation ensure Property 3 (happens-before)?
2. Explain if your implementation is fair, starvation-free, or neither. See Property 2.
3. Explain why your implementation of `isReentered` is as fast as possible.
4. Describe 2 steps you took to improve the performance of your implementation locking mechanisms under contention.

The instructors will grade your submissions by looking at parts of the code you submitted and the video you recorded. Note that **describing any single-threaded behavior will lead to a 0% grade**. Your answer should only consider multi-threaded behavior in the presence of concurrent operations. Also note that correctness properties may be non-local, so you may have to show more code than just what the property refers to (e.g., to ensure that a counter does not suffer from data-races when adding to it, some implementations have to show all operations that modify the counter, such as decrementing the counter).

You can record such a video using Zoom, which you may already have installed to attend lectures remotely. Simply start a meeting (without any other participants), share your screen, and start recording. Note that Zoom requires some time to process your video after you record it, so plan accordingly. Extension requests to upload videos after the due time and date because Zoom is still processing them will be denied.

The maximum length for each video is 1 minute, instructors will stop watching at the 1 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.

Harcoding 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.

Important Considerations

Your assignment implementation should follow all the considerations in this section. Note that this section may be empty now but updated later, please check Piazza for updates.