CS454 Principles of Concurrent Programming
Assignment 3

Prof Luís Pina — University of Illinois Chicago

Spring 2026

In this assignment, you will implement a utility to manage an electric bicycle rental system called Yvvid 🚲. The system comprises of many docks spread throughout a city. Users can ride bikes from one dock to another.

Given that docks have a limited capacity, Yvvid 🚲 requires users to register their trip in advance to ensure there is enough capacity on the destination dock, otherwise the trip is not allowed. Yvvid 🚲 also allows users to rent many bikes for the same trip, to encourage groups of people to ride together. It is important that the Yvvid 🚲 system accounts for all bikes at all times, even if a bike is being ridden right now it still "belongs" to a dock (in this case, the destination dock).

When bikes become damaged or run out of battery, they are removed from the system to be repaired or recharged, respectively. Each bike in the system has a unique ID, which is not reused when a bike is removed for the reasons above. If the same physical bike returns to the system after being repaired or recharged, it counts as a whole new bike and receives a new ID.

All bikes belong to the same Yvvid 🚲 system (in case other cities adopt it), and cannot be moved from one system to another.

Changes between Assignment 3 and Assignment 1 are highlighted in this document.

Description

Your submission should extend the following abstract class:

```
01:  abstract class Yvvid {
02:      abstract Dock createDock(int capacity);
03:      abstract Bike createBike(int id);
04:
05:      abstract boolean addBikes(Dock d, Set bikes);
06:      abstract boolean rideBikes(Dock from, Dock to, Set bikes);
07:
08:      abstract boolean repairBikes(Dock d, Set bikes);
09:      abstract boolean rechargeBikes(Dock d, Set bikes);
10:
11:      abstract Set getBikes();
12:      abstract Set getBikes(Dock d);
13:
14:      abstract Set getBikes(List<Dock> d);
15:
16:      abstract List<Action<Dock>> audit(Dock d);
17:      abstract List<Action<Bike>> audit(Bike b);
18:  }
```

Each operation (method) behaves as follows:

- `createDock`: Creates a dock that accommodates a number of bikes — `capacity`.
- `createBike`: Creates a bike with a given `id`. The `id` is unique across the same Yvvid.
- `addBikes`: Adds all bikes to the dock.
  - `addBikes` should fail when:
    - There is not enough capacity in the dock for all the bikes.
    - Any of the bikes is already added to some other dock.
    - Any of the bikes is either repaired or recharged (see below).
    - For instance, attempting to add two bikes to a dock that only has room for one should fail.
  - If all the bikes are added to the dock, `addBikes` returns `true`. From now on, the dock has all the bikes.
  - If `addBikes` fails, the dock remains unchanged and `addBikes` returns `false`.
- `rideBikes`: Moves `bikes` currently present in the `from` dock to the `to` dock
  - `rideBikes` should fail when:
    - There is not enough room in the `to` dock
    - Any of the bike is not present in the `from` dock
    - Any of the bikes is already added to some other dock.
    - Any of the bikes is either repaired or recharged (see below).
  - If all the bikes are moved from the `from` dock to the `to` dock, `rideBikes` returns `true`. From now on, the `to` dock has all the bikes, and the `from` dock has none of the bikes.
  - If `rideBikes` fails, all docks remain unchanged and `rideBikes` returns `false`.
- `repairBikes`: Marks all the given bikes as repaired, and removes them from the dock.
  - `repairBikes` should fail when:
    - Any of the bikes is not added to the dock.
    - Any of the bikes is already added to some other dock.
    - Any of the bikes is already repaired or recharged.
  - If all the bikes are marked as repaired, `repairBikes` returns `true`. From now on, the dock does not have any of the bikes.
  - If `repairBikes` fails, the dock and all the bikes remain unchanged and `repairBikes` returns `false`.
- `rechargeBikes`: Similar to `repairBikes` described above, except that all bikes are marked as recharged instead.
- `getBikes`: Gets all the bikes in the current dock (i.e., added/moved to the dock and not yet repaired or recharged).
  - Without arguments, `getBikes` lists all the bikes currently in all the docks part of the Yvvid.
  - With a dock argument, `getBikes` lists all the bikes currently in the provided dock.
  - With a list of dock argument, `getBikes` lists all the bikes currently in all the docks contained in the provided list.
- ~~`audit`: Returns an audit log that tracks docks and bikes.~~
  - ~~With a bike argument, returns a list of all the docks in which the bike ever was.~~
    - ~~The order of the list matters.~~
    - ~~When using `rideBikes`, bikes should be removed from one dock before being added to another dock.~~
  - ~~With a dock argument, returns a list of all the bikes that were ever in the provided dock.~~
    - ~~The order of the list matters.~~
    - ~~If an operation changes many bikes at once, the order between those bikes does not matter.~~
    - ~~However, all those bikes should be on the list after preceeding operations and before later operations.~~

Besides the `Yvvid` class described above, your submission should also implement the `Bike` interface for each bike, which defines the `getStatus` operation:

```
17:  interface Bike {
18:      enum Status { READY, IN_CIRCULATION, REPAIRED, RECHARGED }
19:      Status getStatus();
20:  }
```

Each bike should behave as follows:

- All bikes are created as `READY`.
- A bike are that is `READY` can be added to a `dock`, in which case it becomes `IN_CIRCULATION`.
- Once a bike is `IN_CIRCULATION`, it cannot become `READY` again.
- A bike that is `IN_CIRCULATION` can be repaired. The status of that bike becomes `REPAIRED`.
- A bike that is `IN_CIRCULATION` can be recharged. The status of that bike becomes `RECHARGED`.
- Once a bike is either repaired or recharged, it is no longer in its dock and it cannot go back to any dock in the same Yvvid.

Correctness Requirements

Your implementation should keep the following properties at all times:

Correctness 1: `getBikes` operations never list more bikes than the capacity of the dock.

Correctness 2: `getBikes` operations never list more bikes for the whole Yvvid than the sum of the capacity of all the docks.

Correctness 3: Adding ready bikes to any dock successfully results in those bikes being present in later `getBikes` operations on the same dock.

Correctness 4: Once a bike is repaired or recharged, that bike is not listed in later `getBikes` operations.

Correctness 5: Each bike is listed in one dock at most by `getBikes` operations.

Correctness 6: Bikes are never "in-transit" due to `rideBikes` operations (i.e., `getBikes` operations not listing bikes removed from the `from` dock and still not added to the `to` dock).

Correctness 7: Once the status of a bike is observed to be `REPAIRED` or `RECHARGED`, it cannot be observed to be anything else from that point on.

Correctness 8: It is not possible to observe partial results of any operation. Each operation either happens completely, or not at all.

Correctness 9: Your implementation shoud not have any data races.

Correctness 10: ~~The current contents of any dock can be explained by following the entries in the audit log, by the order in which they appear in the log.~~

Correctness 11: ~~The current state and location of any bike can be explained by following the entries in the audit log, by the order in which they appear in the log.~~

Concurrency Requirements — Progress

Your implementation should allow multiple threads to make progress at the same time, according to the following properties:

Progress 1: Concurrent `getBikes` operations should all make progress in parallel.

Progress 2: The following operations should all make progress in parallel when they are applied to different docks and bikes: `rideBikes`, `addBikes`, `repairBikes`, and `rechargeBikes`.

For instance, consider 3 threads executing at the same time as follows:

1: `rideBikes` bike 1 from dock 1 to dock 2

2: `repairBikes` bike 2 on dock 3

3: `addBikes` bike 3 on dock 4

In this example, all 3 threads should make progress at the same time as they operate on different docks and different bikes.

Progress 3: `rideBikes`, `addBikes`, `repairBikes`, `rechargeBikes`, and `getBikes` operations that do not target the same dock should all make progress in parallel.

For instance, consider 4 threads executing at the same time as follows:

1: `addBikes` to dock 1

2: `getBikes` to dock 2

3: `getBikes` to docks 3 and 4

4: `getBikes` without arguments (on the whole Yvvid, including all docks 1, 2, 3, and 4).

In this example, threads 1, 2, and 3 should all make progress in parallel because threads 2 and 3 read the contents of docksnot being changed by thread 1. Thread 4 cannot make progress in parallel with Thread 1, and it is acceptable for only one of these threads (i.e., either 1 or 4) to make progress until the other finishes its operation.

Progress 4: Your implementation should be as least blocking as possible. Each operation listed in this document should acquire the fewest amount of locks needed and hold them for the shortest duration possible to remain correct.

Progress 5: No combination of operations issued concurrently by different threads should result in a deadlock in any possible execution.

Concurrency Requirements — Correctness

Your implementation should be **linearizable**. All operations in any execution should appear to take place in a single point. You can use locks to ensure linearizability.

Linearizability 1: When adding multiple bikes b1 and b2 to a dock d with `addBikes`, it is not possible for other threads to see b1 added and b2 missing. Other threads calling `getBikes` either see all bikes b1 and b2 in d, or none. It is also not possible for other threads to see b1's status as `READY` and b2's as `IN_CIRCULATION` (either both are `READY` or both are `IN_CIRCULATION`).

Linearizability 2: If two threads attempt to add the same bike b at the same time to docks d1 and d2, only one thread will succeed. This holds if the docks are different (d1 $\neq$ d2) or the same (d1 = d2).

Linearizability 3: When moving multiple bikes b1 and b2 between docks d1 and d2 with `rideBikes`, it is not possible for other threads calling `getBikes` to see b1 in d1 and b2 in d2, or b1 in d2 and b2 in d1. All bikes moved are either in d1 or d2. This does not affect other bikes in d1 or d2 not targeted by the `rideBikes` operation.

Linearizability 4: When calling `repairBikes` or `rechargeBikes` on bikes b1 and b2, it is not possible for other threads calling `Bike.getStatus` to see b1 as `IN_CIRCULATION` and b2 as `REPAIRED`/`RECHARGED` (or the other way around). Either all the bikes are `IN_CIRCULATION` or all the bikes are `REPAIRED`/`RECHARGED`. This does not affect other bikes in the same dock not being used in the `repairBikes`/`rechargeBikes` operations.

Linearizability 5: If two threads attempt to `repairBikes` or `rechargeBikes` the same bike b at the same time, only one thread will succeed.

Bonus Extra Functionality

Besides implementing all of the above, you can claim a 10% bonus by implementing `Bike.nonLinearStatus`:

- `nonLinearStatus` should execute faster than `getStatus`. You cannot make `getStatus` slower just for `nonLinearStatus` to be faster. You do not have to show that `nonLinearStatus` executes faster, but you have to explain why you expect it to.
- Properties **Linearizability 1**, **Linearizability 4**, **Progress 4**, and **Correctness 8** may not hold for `nonLinearStatus`.
- You are free to use any data-structure from `java.util.concurrent` seen in lectures to implement `nonLinearStatus`.
- All other correctness, linearizability, and progress properties must hold for `nonLinearStatus`.

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method `Yvvid.createYvvid` so that it creates an instance of the class you added. You cannot change any other part of the code that is provided to you.

```
21:  abstract class Yvvid {
22:      static Yvvid createYvvid() {
23:          throw new Error("Not implemented");
24:      }
25:  }
```

Due Date and Resubmission Policy

This assignment is due on **March 14 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 **March 21 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 20% bonus points.

You can earn a 10% bonus by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment3` label, and non-anonymous to the instructors to count towards the bonus.

You can earn a 10% bonus by implementing and explaining (correctly) `nonLinearStatus` as described in this document. There are no partial points for the bonus portion, and you cannot resubmit it.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 70%. 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%.

You need to pass Test 3 to get credit for Tests 4, 5, 6, and 7. This requirement prevents submitting an unchanged Assignment 1 and still get points without any effort towards the progress/concurrency requirements explained above.

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. Which operations can potentially violate property **Progress 5**, and what steps did you take to avoid that?
2. How do you ensure concurrent progress on operations that get the contents of the same dock (**Progress 1**)?
3. How do you ensure linearizability when adding bikes to a dock and when repairing/recharging bikes(**Linearizability 1** and **Linearizability 4**)?
4. **(Bonus)** How did you ensure that `nonLinearStatus` performs as described in this document?

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.