ASSIGNMENT 3

CS 454: Principles of Concurrent Programming / Spring 2025

Description

In this assignment, you will update your Assignment 1 depot.

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

Your submission should extend the following abstract class:

```
abstract class Depot {
  abstract Bus createBus(int capacity);
  abstract Ticket issueTicket(int id);

  abstract boolean boardBus(Bus b, Set tickets);
  abstract boolean transferTickets(Bus from, Bus to, Set
tickets);

  abstract boolean useTicket(Bus b, Set tickets);
  abstract boolean expireTicket(Bus b, Set tickets);

  abstract Set getTickets();
  abstract Set getTickets(Bus b);
  abstract Set getTickets(List<Bus> b);


  abstract List<Action<Bus>> audit(Bus b);
  abstract List<Action<Ticket>> audit(Ticket t);
}
```

Each operation (method) behaves as follows:

- **createBus**: Creates a bus that can carry a number of passengers – capacity
- **issueTicket**: Issues a ticket with a given id. The id is unique across the same Depot.
- **boardBus**: Adds all tickets to the bus
    - This operation either adds all the tickets, if the bus has enough capacity, or none.
    - For instance, attempting to add two tickets to a bus that only has room for one should not change the contents of the bus.

    - If all the tickets are added to the bus, this operation returns true. If the bus remains unchanged, this operation returns false.
- **transferTickets**: Moves tickets currently present in the from bus to the to bus.
    - If there is not enough room in the to bus, this operation should fail and return false.
    - If any ticket is not present in the from bus, this operation should fail and return false.
    - This operation returns true if it succeeds in moving all the tickets between buses.
- **useTicket**: Marks all the given tickets as used, which should be present on the given bus.
    - Similarly to **boardBus**, this operation either marks all the tickets or none.
    - Trying to use a ticket that is not in the current bus results in failure of the whole operation (i.e., no tickets are used).
    - If all the tickets are used, this operation returns true. If any ticket was not used, then no tickets are modified and this operation returns false.
- **expireTicket**: Similar to useTicket described above, but marks all tickets as expired
- **getTickets**: Gets the tickets in a given bus (i.e., added to the bus, and not used or expired).
    - Without arguments, this operation lists all the tickets currently in circulation in all the buses
    - With a bus argument, this operation lists all the tickets currently in that bus.
    - With a list of buses argument, this operation lists all the tickets in all the buses provided.
- ~~**audit**: Returns an audit log that tracks buses and tickets.~~
    - ~~With a ticket argument, returns a list of all the buses in which the ticket ever was~~
        - ~~The order of the list matters~~
        - ~~When transferring, tickets should be removed from one bus before being added to another bus~~
    - ~~With a bus argument, returns a list of all the tickets that passed by that bus~~
        - ~~The order of the list matters~~
            - ~~If an operation changes many tickets at once, the order between those tickets does not matter.~~

Besides the Depot interface, your solution should also implement the Ticket interface for each ticket, which defines the **getStatus** operation:

```
interface Ticket {
    enum Status { ISSUED, IN_CIRCULATION, USED , EXPIRED }
    Status getStatus();
}
```

Each ticket should behave as follows:

- All tickets are issued as ISSUED
- An ISSUED ticket can be added to a bus, in which case it becomes IN_CIRCULATION
- Once a ticket is in circulation, it cannot become ISSUED again
- An IN_CIRCULATION ticket can be used, and becomes USED

- An IN_CIRCULATION ticket can expire, and becomes EXPIRED
- Once a ticket is not in circulation, it cannot go back in circulation or be issued again

Correctness Requirements

Your implementation should keep the following properties at all times:

Correctness 1. **getTickets** operations never list more tickets than a bus's capacity

Correctness 2. **getTickets** operations never list more tickets for the whole depot than the sum of the capacity of all the buses.

Correctness 3. Adding issued tickets to a bus successfully results in those tickets being listed in later **getTickets** operations.

Correctness 4. Once a ticket is used or expires, that ticket is not listed in later **getTIckets** operations.

Correctness 5. Each ticket is listed in one bus at most by **getTickets** operations.

Correctness 6. Tickets are never "in-transit" due to transfer operations (i.e., **getTickets** operations not listing tickets removed from the from bus and still not added to the to bus).

Correctness 7. Once the status of a ticket is observed to be USED or EXPIRED, 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 should not have any data races

~~**Correctness 10.** The current contents of any bus 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 ticket 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 **getTickets** operations should all make progress in parallel.

Progress 2. The following operations should all make progress in parallel when they are applied to different buses and tickets: **transferTickets, boardBus**, **useTicket**, **expireTicket**.

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

1. transfer ticket 1 from bus 1 to bus 2
2. use ticket 2 on bus 3
3. board ticket 3 on bus 4

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

Progress 3. **transferTickets, boardBus**, **useTIcket**, **expireTicket** and **getTickets** operations that do not target the same bus should all make progress in parallel.

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

1. **boardBus** to bus 1
2. **getTickets** on bus 2
3. **getTickets** on buses 3, and 4
4. **getTickets** without arguments (on the whole depot, buses 1, 2, 3, and 4)

In this example, thread 1, 2, and 3 should all make progress in parallel because threads 2 and 3 read the contents of buses not 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 tickets T1 and T2 to a bus B with **boardBus**, it is not possible for other threads to see T1 added and T2 missing. Other threads calling **getTickets** either see all tickets T1 and T2 in B, or none. It is also not possible for other threads to see T1.**getStatus** as ISSUED and T2.**getStatus** as IN_CIRCULATION (either both are ISSUED or both are IN_CIRCULATION).

Linearizability 2. If two threads attempt to add the same ticket T at the same time to buses B1 and B2, only one thread will succeed. This holds if the buses are different (B1 ≠ B2) or the same (B1 = B2).

Linearizability 3. When moving multiple tickets T1 and T2 between buses B1 and B2 with **transferTickets**, it is not possible for other threads calling **getTickets** to see T1 in B1 and T2 in B2, T1 in B2 and T2 in B1. Either all tickets moved are in B1 or B2. This does not affect other tickets in B1 or B2 not targeted by the move.

Linearizability 4. When using or expiring tickets T1 and T2 from a bus with **useTicket,** or **expireTicket**, it is not possible for other threads calling **Ticket.getStatus** to see T1 as IN_CIRCULATION and T2 as USED/EXPIRED. Either all the tickets are IN_CIRCULATION or all the tickets are USED/EXPIRED. This does not affect other tickets in the same bus not being used in the use/expire operations.

Linearizability 5. If two threads attempt to use/expire with the same ticket T 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 **Ticket.nonLinearStatus**:

- **nonLinearStatus** should execute faster than **getStatus**. You don't have to show that it executes faster, but you have to explain why you expect it to.
- All correctness properties hold for **nonLinearStatus**
- Properties **Linearizability 1 and 4** may not 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 Depot.createDepot 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.

```
abstract class Depot {
    static Depot createDepot() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **March 15 2025** (Saturday) at **5pm CST**. There is no late policy.9

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%** for undergraduate students, and **50%** for graduate students. You can resubmit your assignment until **March 22 2025** (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-3` 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 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. You can record such a video using Zoom (create a meeting and record your screen)

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 bus (**Progress 1**)?
3. How do you ensure linearizability when adding tickets to a bus and when using/expiring tickets (**Linearizability 1 and 4**)?
4. (Bonus) How did you ensure that **nonLinearStatus** performs as described in this document?

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. **Altering the playback speed of your recorded video (e.g., play at 2x so you get more time) is considered a violation of academic integrity and will result in the penalties described in the syllabus.**

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.