ASSIGNMENT 4

CS 454: Principles of Concurrent Programming / Spring 2025

Description

In this assignment, you will implement a utility to manage touristic sight-seeing bus routes and respective tickets. In this particular case, a ticket allows passengers to ride any bus for any portion of its route. Passengers are also allowed to transfer buses on the same ticket. Once issued, tickets get a unique ID and can be used or they can expire. Tickets are valid for multiple trips, depending on the particular agreement between the bus operator and several tourist agencies; it is up to the bus driver to decide when a ticket is "used" as passengers leave the bus after taking all possible trips that the ticket allows. Tickets can also "expire" as passengers leave the bus, meaning that the ticket was not used for all its possible trips but it cannot be used further because the agreement is not valid past the current date and time.

Tickets already used and expired tickets cannot be used again. Most passengers travel in groups, so operations always take a set of tickets and should perform the same operation on all tickets if possible, or fail without changing any ticket or adding any passenger to any bus.

All buses belong to a particular depot, and cannot be reassigned to different repos. Buses have a maximum number of passengers they can take at any given time, which cannot ever be exceeded.

Changes between Assignment 4 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 Result<Boolean> boardBusAsync (Bus b, Set tickets);

  abstract Result<Boolean> transferTicketsAsync(Bus from, Bus to, Set tickets);

  abstract Result<Boolean> useTicketAsync (Bus b, Set tickets);

  abstract Result<Boolean> expireTicketAsync (Bus b, Set tickets);

  abstract Result<Set> getTicketsAsync ();

  abstract Result<Set> getTicketsAsync (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.
- **Asynchronous methods**: Each method described above has an asynchronous version
    - The asynchronous methods should return as fast as possible
    - The asynchronous methods return a Result object, which the caller can then use to retrieve the result of the operation
    - The asynchronous methods do not wait for the operation to be performed
- ~~**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.~~
            - ~~However, all those tickets should be on the list after preceding operations and before later operations~~

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:

1. **getTickets** operations never list more tickets than a bus' capacity
2. **getTickets** operations never list more items for the whole depot than the sum of the capacity of all the buses.
3. Adding issued tickets to a bus successfully results in those tickets being listed in later **getTickets** operations.
4. Once a ticket is used or expires, that ticket is not listed in later **getTickets** operations.
5. ~~Each ticket is listed in one bus at most by **getTickets** operations.~~
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).~~
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.
8. ~~It is not possible to observe partial results of any operation. Each operation either happens completely, or not at all.~~
9. ~~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.~~
10. ~~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.~~
11. Transfer operations cannot lose any of the tickets being moved. Eventually, all the tickets will be either in the destination bus (success) or in the source bus (fail).

Concurrency Requirements

In this assignment, you are provided with an implementation of `Bus` that has the following properties:

- The provided bus already has a set contents to contain all the tickets in the bus that are in circulation
- Each bus is associated with its own worker thread
- Each bus's contents can be modified only by the worker thread of that wallet
    - If any other thread attempts to add/remove tickets from a bus, the code throws an exception that will cause all tests to fail
- Asynchronous methods must preserve order: If a thread adds a ticket to a bus and then uses it, via method calls `boardBusAsync` followed by `useTicketsAsync`, then both results should (eventually) be true
- Your implementation cannot have data-races or use busy-waiting
- Asynchronous operations on different buses must all make progress in parallel.
    - For instance an async transfer from B1 to B2 and another from B3 to B4 and an async get on bus B5 should all make progress in parallel

Bonus Extra Functionality

Besides implementing all of the above, you can claim a 10% bonus by ensuring minimizing the execution time of method **Depot.getTickets()** according to Amdahl's law.

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");
    }
}
```

You should also implement your own result, returned by the asynchronous operations.

Due Date and Resubmission Policy

This assignment is due on **April 5 2025** (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%** for undergraduate students, and **50%** for graduate students. You can resubmit your assignment

until **April 12 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 `assignment4` 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%.

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 do you avoid busy-waiting in your implementation?
2. Is it correct to modify `Bus.contents` using an operation that **does not** create an inter-thread happens-before relationship with reading it? Why?
3. How do you ensure that a failing transfer operation does not result in tickets being lost?
4. (bonus) What steps did you take to ensure minimize the execution time of Depot.getTickets() according to Amdahl's law?

A solution that handles an interrupted exception incorrectly will result in a 0% grade for all questions.

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. Speeding up the video is considered cheating, please refer to the syllabus for the penalties associated with academic integrity issues.

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 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.