COSI 131a: Fall 2019

Programming Assignment 2

Due date for task 1 is November 4th and task 2 November 11th

Overview

Your task is to implement a simulation of tunnels and vehicles using Java Threads. There will be few tunnels, but many vehicles. There will be several constraints on how many vehicles of each type a tunnel can contain at any given time. After you implement this primary system, you will implement a higher-level tunnel controller that will schedule access to the tunnels based on a priority system.

The project is split into 2 required tasks. Task 1 requires you to use Java default synchronized methods and busy waiting to prevent race conditions. Task 2 requires you to build on your synchronized code and add Java condition variables to avoid busy-waiting and add a scheduler.

Tunnel Description

Any implementation of the Tunnel interface must satisfy the following properties:

- Each tunnel has only one lane, so at any given time all vehicles must be traveling in the same direction.
- Only three cars may be inside a tunnel at any given time.
- Only one sled may be inside a tunnel at any given time.
- Cars and sleds cannot share a tunnel.

As you know from learning about concurrency, without proper synchronization, these constraints could be violated since a thread may secure permission to enter a tunnel, then be interrupted before it can actually enter, at which point another thread may also secure permission to enter the tunnel and do so. When the first thread to get permission to enter the tunnel does so, one of the invariants may be violated (e.g., maybe a sled and car will be in the tunnel at the same time). Part of your job in this assignment is to use synchronization to prevent such race conditions from occurring regardless of scheduling.

To implement your solutions, you will need to use the code base provided by us, described below. This code base includes a test harness that allows us (and you) to test the correctness of your solutions. You will be therefore instructed to follow certain conventions in your code and will be not allowed to modify certain classes. For example, your code will not be starting or joining client threads. Instead, the threads will be started by the test harness. Please follow the instructions carefully. You will need to understand how the entire provided code works to complete your assignment.

Provided Code

The code for this assignment is divided into the following packages

cs131.pa2.Abstract

The `Abstract` package contains the abstract classes and interfaces that your program must implement. You are not allowed to modify any of the files within this package

cs131.pa2.Abstract.Log

The `Abstract.Log` package contains classes used to record the actions of the car. These classes are used by the test packages to evaluate that constraints have not been violated. The log is not a means for passing messages. The only reference to Log classes in your code should be in the `ConcreteFactory.createNewPriorityScheduler` method. You are not allowed to modify any of the files within this package.

cs131.pa2.Test

You may not edit any of the test classes, but we highly encourage you to study them for your understanding and to aid in discovering why a test is failing.

cs131.pa2.CarsTunnels

The `CarsTunnels` package contains a few mostly empty classes that implement the classes found in the Abstract package. All of the code pertaining to your solution of this assignment must go in this package. You are free to add, edit, rename, and delete files in this package in the course of implementing your solutions.

API Documentations
You will want to inspect the java files yourself, but to get you started we include the API docs generated with javadoc from the files we provide.

Task 1: The Basic Tunnel

Implementation

You are provided with a class called `BasicTunnel` that implements the interface `Tunnel`, and you must implement functionality to carry out the following tasks:

- Enforce the `Tunnel` Restrictions described above.
- Use the Java `synchronized` keyword to prevent race conditions (without introducing deadlock).

When this task is complete your code should pass all tests included in `BehaviorTest`, and `SimulationTest`. Please make sure to run the tests several times as a race condition problem can appear in only some of the runs. We provide the `TIMES` variable inside each test

class that specifies how many times to run each test. It is set to 1 by default but feel free to change it to how many times you want to run each test.

You will need to provide JavaDoc documentation for any new classes, methods, or members you create, and you may be penalized if you fail to do so.

Task 2a: The Priority Scheduler

You have now successfully programmed `BasicTunnel` which enforces entry criteria and prevents race conditions. However, there are two problems with the design of `BasicTunnel`:

- While no tunnel is available to a `Vehicle`, that `Vehicle` busy-waits (loops over all tunnels repeatedly inside of its `run()` method).
- There is no way of prioritizing important vehicles (say, an ambulance) over less-important vehicles (say, an ice cream truck).

Priority Defined

- Priority is a number between 0 and 4 (inclusive).
- A higher number means higher priority.

Implementation

You are provided with a class called `PriorityScheduler` (in `PriorityScheduler.java`) that implements `Tunnel` and you must add functionality that controls access to a collection of Tunnels in order to implement the priority scheduling policy described above. The Tunnels "behind" the `PriorityScheduler` will be `BasicTunnels`. `BasicTunnel` should be the same class you implemented for the first part of this project. `PriorityScheduler` will carry out the following tasks:

- Keep references to `BasicTunnels` as private member variables inside `PriorityScheduler`.
- When a `Vehicle` calls `tryToEnter(vehicle)` on a `PriorityScheduler` instance, the following general steps should be executed:
    - If `vehicle.getPriority()` is less than the highest priority from among all the other vehicles waiting to enter a tunnel (i.e., there is another vehicle with higher priority)
    - Or there are no other vehicles with higher priority but there are no tunnels into which the vehicle can currently enter,
    - Then the vehicle thread must wait;
    - Otherwise the vehicle successfully enters one of the tunnels.
- When a `Vehicle v` exits the scheduler by calling `exitTunnel` on a `PriorityScheduler` instance, the scheduler must call `exitTunnel(v)` on the appropriate tunnel from the collection of tunnels managed by the scheduler (remember, you may **not** modify `Vehicle` or any of the other classes provided to you, or `BasicTunnel`, so you must solve this problem within `PriorityScheduler`). After a vehicle has successfully exited a tunnel, the waiting vehicles should be signaled to retry

to enter a tunnel. **Note that the vehicles with highest priority should be allowed to enter.**
- Use condition variables to avoid busy waiting when the car cannot find a tunnel to enter. Make sure the use of the condition variables is safe.

When this task is complete your code should pass the tests `PrioritySchedulerTest.CarEnter`, `PrioritySchedulerTest.SledEnter` and `PrioritySchedulerTest.Priority` included with this assignment. Please make sure to run the tests several times as a race condition problem can appear in only some of the runs. We provide the `TIMES` variable inside each test class that specifies how many times to run each test. It is set to 1 by default but feel free to change it to how many times you want to run each test.

You will need to provide JavaDoc documentation for any new classes, methods, or members you create, and you may be penalized if you fail to do so.

Task 2b: The Preemptive Priority Scheduler

This task is to modify your scheduler to be preemptive. In order to do this, you are allowed to modify the `Vehicle.doWhileInTunnel` method and add any required members in the `Vehicle` class, as well as the `Tunnel` class. You many include new class members and methods, but you should not change the way log entries are entered in the `Log`. The new scheduler class is `PreemptivePriorityScheduler`. We suggest that you run your design idea by the TAs before you start coding this task.

Preemption Rules

- The highest priority value (4) will be reserved for ambulances.
- An ambulance may share a tunnel with any other vehicles (including a sled) **except** another ambulance (i.e., the only time an ambulance must wait is if there is already an ambulance in all tunnels).
- Any time an ambulance wants to enter a tunnel, any vehicles in that tunnel must immediately pull over and wait for the ambulance to completely pass through the tunnel. That is, the vehicle cannot make progress toward the end of the tunnel while the Ambulance is in the tunnel.
- Once an ambulance leaves the tunnel, vehicles in the tunnel start making progress again.

Implementation

- Modify `Vehicle.doWhileInTunnel` as follows:
  - Use Java condition variables to avoid busy-waiting while an ambulance is in a tunnel.
  - Use `await(timeout)` where the `timeout` is the same as the parameter to `Thread.sleep` in the implementation provided to you.
  - Think carefully about what object to lock (remember, a synchronized method locks "this"). You may modify the `Vehicle` constructor (or add a new field and

setter/getter) if you wish for the purpose of referencing another object to synchronize on.

- Here is an example of how `Vehicles` wait on an ambulance:
  - Say a vehicle waits in a tunnel for 100 ms, and has currently been waiting for 50 ms
  - When it is notified that an ambulance is entering its tunnel, it waits indefinitely for the ambulance to leave the tunnel
  - When the ambulance notifies it that it has left the tunnel, the vehicle begins its timeout wait again, using whatever time was remaining on the clock from when it was woken up the first time by the ambulance. In other words, the vehicle calls await(50).

Tests

You will have to pass the `PreemptivePriority, PreemptivePriorityManyAmb, PreemptivePriorityManyTunnels` and `AmbulanceEnter` tests from `PrioritySchedulerTest` JUnit class. Please make sure to run the tests several times as a race condition problem can appear in only some of the runs. We provide the `TIMES` variable inside each test class that specifies how many times to run each test. It is set to 1 by default but feel free to change it to how many times you want to run each test. Your code must satisfy the described conditions to get full credits. That is, simply passing the tests does not give you full points. Please make sure to discuss your solution with the TAs before submitting.

You will need to provide JavaDoc documentation for any new classes, methods, or members you create, and you may be penalized if you fail to do so.

Hints

`instanceof` operator

You might find the instanceof operator helpful. instanceof can tell you if an object can be downcast from a parent type into a child type. Typically, we don't use it because there are better techniques (polymorphism leverages the type system to do the work for you), but for this problem it will probably help you out.

Here is an example:

```
public class TestInstanceof {

      public static class Parent {}
      public static class Child1 extends Parent {}
      public static class Child2 extends Parent {}
```

```
	public static void main(String[] args) {
		Parent p1 = new Child1();
		Parent p2 = new Child2();

		if(p1 instanceof Child1) {
			System.out.println("p1 is instance of Child1");
		}
		if(p1 instanceof Child2) {
			System.out.println("p1 is instance of Child2");
		}
		if(p2 instanceof Child1) {
			System.out.println("p2 is instance of Child1");
		}
		if(p2 instanceof Child2){
			System.out.println("p2 is instance of Child2");
		}

	}
}
```

This outputs:

```
p1 is instance of Child1
p2 is instance of Child2
```

No “main” method

Note that your only point of entry in the code we provide is through the JUnit tests, which will set up the environment in which your client threads will run. Your tasks for this assignment do not include writing a main method. Rather, you must rely on your understanding of busy waiting/condition variables, and the JUnit tests to know if you have completed the assignment.

Submission

Please export your project together with javadocs as described on LATTE and submit as a zip file. You should not submit source files individually.

Important Note about Disallowed Java Tools

In PA1, you were instructed to consider using a synchronized class provided by Java for inter-thread communication (`LinkedBlockingQueue`) to solve the problem. For this project, that is **not allowed**; you may not use **any** synchronized data structure included in the Java API. You must write your own (using the "synchronized" keyword). Of course, you can and should use non-synchronized data structures in the Java standard library. You can consult the API docs to see if a data structure is synchronized.
You also **may not** use the thread priority methods provided by Java (e.g., you may not use

`Thread.setPriority`).
Finally, you should solve this assignment by using Java 8 and Junit 5. No external packages are allowed, unless you got permission from the TAs.