Concurrent programming Lab 1 documentation

Anton Eliasson Gustafsson and Christian Nilsson

Our solution for lab 1, Trainspotting, was implemented using an inner class which implemented Runnable. We used this solution to be able to send 3 different parameters which we felt was needed; speed of the trains, traiId and a boolean indicating if it was the train going up or down in the start. These 3 variables was used during the start of the run() method to make sure the threads controlled the right train.

In our implementation, a train can only acquire a semaphore on active sensors and depends on which direction it has.

In the reversed way, a train can only release a semaphore on inactive blocks. Also here, a train will release depending on both the sensor and the train's direction. This decision is based on the fact that in some cases, a train with much higher speed could otherwise hit a lower-speeding-train's tail. However, this could only happen in a few cases, but for consistency in our code it felt better to do so in all of them.Having release on inactive blocks seems therefore safer.

Two methods predefined and executed inside run() :

```
public void trainWait(Semaphore s, int trainId, int speed)
```

This method is used is some cases when a train hits a position next to a critical region. A train will set it's speed to 0 and wait for a semaphore to be acquired. If the train access the semaphore, it's speed will be set to whatever it had before. Note that if the semaphore is available just as a train tries to access it, the train will appear as if the speed was not set to 0 at all!

However the method was not used in cases when a train could go another alternative route. In those cases we used tryAcquire() instead (see in our code).

```
public int stationStop(int trainId, int speed)
```

This method is used whenever a train arrives to a train station. The train's speed is set to 0, then the thread on which the trains runs on sleeps for a moment. After this the train reverses it's direction and speed.

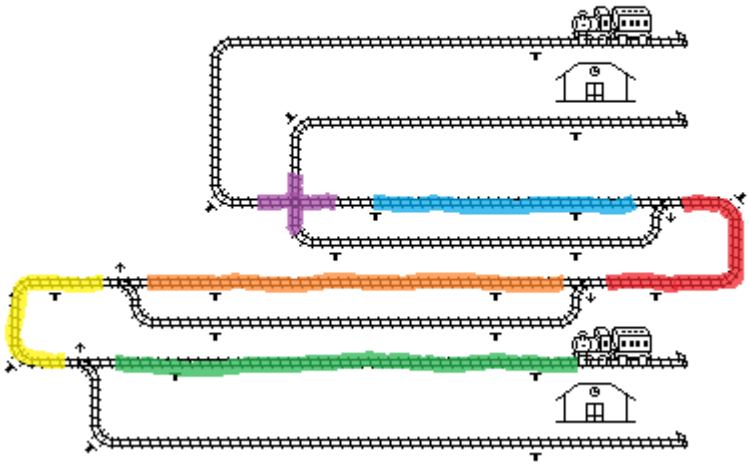
We placed the sensors 2 steps away from all the switches and crossings to give the system enough time to stop the trains before they reach the critical areas. We added a sensor at each end of the tracks to make the station stops possible.

We have 6 critical regions (semaphores) on in our solution. ToppKorsning is the purple intersection where the train have to wait until the other one have fully crossed it. topTop is the blue critical region for choosing the path for the upper station stop. topRight is the red upper single track section. middleMid is the orange section for choosing upper or lower path of the middle track. bottomLeft is the yellow bottom left single track section. bottomDefault is the green for choosing track for the lower station stop.

Depending on the direction a train has, the train tries to access a critical region (acquires a semaphore) when a sensor just before a critical region is active.

The switches are switched depending on direction, active sensors and whether or not a train access a semaphore.

As stated above, semaphores are released on inactive sensors. In some cases just outside the critical region, but sometimes even inside (at the end of) a critical region. It does not cause any problem though. For instance, if a train comes from the top going through topTop, it will only release topTop after it access topRight.



The reason for the max speed is that the train takes to long to stop from the sensors so they stop on critical parts of the track. 15 as max speed showed no collisions, however 20 did. Maybe we could adjust the max speed to 19 – 16. But in that case we need to test a lot of cases. Therefore 15, which is the minimum max speed seemed as the safest choice. We tested our solution by running the trains at different speeds and seeing if they crashed or not. In some cases we placed the trains on different start positions in order to see how they behaved at specific critical regions.