Meilstein Omega

Antoine Scheffold 2556024
Roman Tabachnikov 2565209

July 2018

1 Introduction

Our implementation revolves around two new classes; the `Train` which is an extensions of the `Thread` class to simulate their "concurrent" nature and the `TrainService` (i.e. "Fahrdienstleitung") and which controls and manages the `Train`'s movement.

Only other modification we implemented was a `Lock` on both sub-classes of `Position`; `Location` and `Connection`. By Locking a `Position` for a certain thread, this allows us to implement the realty of never heaving two`Train`s on the same track, or on the same station at the same time. Additionally, on `Location` we added a `getParking()` and `freeParking()` synchronized methods to access the "shared" parking variable.

2 Execution

We start our Simulator with an initialization of our `TrainService` and all the `Train`s which take part in this particular `problem` instance, after which we let them run and wait for their termination with `.join()` on the current `main` method, allowing the waiting Simulator to finish the program.

2.1 train logic

When a `Train` wants to go somewhere, it would first ask the `map` for the shortest route, after which it would inquire our `TrainService` for a reservation to drive on this same route. If the `TrainService` denies - the `Train` would then ask for a reason; which `Position` is the cause of the problem (i.e. already locked). With the new information it can query the `map` (using the "avoid" parameter) to find an alternative route to its destination, without those "problematic" elements. If such route doesn't exist, the `Train` would look for the closest Parking place on the most direct route, ask the `TrainService` to reserve parking there and in a case of a decline, repeatedly ask him to reserve until it will get the reservation. This loop will repeat itself until the `Train` has reached its given destination.

2.2 reserving

The reservation method on the `TrainService` can be run by multiple Threads. The `TrainService` will try to lock all the `Position`s that are part of the given route using the tryLock(). To avoid a "back and forth" Problem, we first sort the given route's elements by their ids. This is done to give all reservations the same order; which in turn means that there will always be one `Train` that gets its reservation approved. If any of the components are already "taken", the `TrainService` would unlock everything that he locked thus far and return a negative answer to the `Train`.

2.3 driving

On a positive answer, our`Train`can finally proceed to drive through our approved route; Starting from the first Connection; it would figure the right direction based on its current `Location` and call leave() on that `Location` - thus the `TrainService` would unlock the given `Position`. The same happens with the `Connection` between our current `Location` and the next one, on which, when we're done with travel() over the connection, call arrive() and proceed to update our current location. This is repeated until the `Train` reaches this route's final destination. This way of locking on reservation and unlocking on pass-through lets up keep a "healthy" 1-to-1 ratio of locks to unlocks (i.e. we avoid locking twice). On a leave() we not only unlock the corresponding Position, but more importantly let any waiting `Train` a sign for any waiting trains (this is further explained in section 2.6). We decided to use Condition.signalAll() instead of a individual signal() because different `Train`s could be waiting for different elements, meaning we better let them all try to reserve again.

2.4 no reservation - no drive

The "disappointed" `Train` that got an unsuccessful reservation in the last phase, would ask `TrainService` for a list of positions that were the reason for the disapproval. This is done by iterating over the route again and writing the locked Positions into a list, which is then used by `Train`to ask the map for an alternative route. If such route exists the `Train` would then try again to reserve the new route. This whole ordeal is repeated until the map can't find any route, that doesn't include the "avoid" list. In this case the `Train` goes into the next mode.

2.5 "looking for parking"

The `Train` would query map for the shortest route to its final destination and the `Train` would then look for possible parking spots on this route. The `Train` would iterate over the list, checking from stating `Location` to the destination if it's possible to park there. In-case that it's a parking place, it would check if there are still free parking places left (stations have infinite). This would happen though the monitor that we implemented in `Location` to avoid data-races on this variable. Worst case scenario is that no parking is available on route and our `Train` would get its final destination as its assigned parking place. As per instructions, when a `Train` parks on a non-station parking spot, we would give pause() to recorder. Similarly any departure from a parking spot which is not a station would require a resume() to be given to recorder.

2.6 "waiting for parking-route"

Like it was described in out "train logic", when a `Train` gets a parking reservation it has to reserve the route to it as-well. When it can't get such reservation (done the same way as before, through `TrainService`), it would have to wait. By implementing a condition that listens for one of the maps elements to be unlocked by a passing `Train`, our `Train` would wake up and try to reserve the same route to the aforementioned parking spot. If the route is still blocked (one of the elements is still locked) it would go into await() state again, until next signal. Like before this Condition forces us to use locks here to avoid breaking. When a `Train` finally gets a route reserved, it would follow the same driving procedure as before.

2.7 "repeat until finish"

Our `Train`s would repeat this whole ordeal until hey arrive at their final destination. At which point they would notify recorder with finish() and terminate; thus resolving the join() called upon in Simulator.

3 Final words on Concurrency

Our model deals with two problems; the route and parking reservation. The first one is solved by implementing locks on every single `Position` which are unlocked when a `Train` passes through them. Based on the notion that our implementation is valid; we can assume that no dataraces over those route elements are possible. All the operations over those elements happen with `TrainService`'s assistance. The use of tryLock() avoids deadlocks over the locks and because our `Train`s follow strictly their assigned route we know the unlock them according to our model. the second problem deals with the variables of parking-spots on "parkable" locations. To avoid multiple `Train`s accessing this variable we implemented synchronized methods to work as monitors. Based on the assumption that our implementation is valid; we can assume that this variable will be accessed by only one `Train` at a time - thus avoiding dataraces altogether.

Figure 1: Sincerely, Your Train-managers; Antoine and Roman