1 Introduction

The goal of this lab is to implement the management of shared rail sections in a train simulation. The source code of the simulator and a program boilerplate for the student program are provided. The simulator is a multithreaded program that mocks up the behavior of trains on a rail network, leading to synchronization concerns. The student program must implement the management of at least a shared rail section, the synchronization between the trains at the station and an emergency stop system.

2 Analysis

In a concurrent system where multiple trains are operated by separate threads, special attention must be paid to synchronization to ensure that all entities can use the network safely without running into one another. Critical sections in this context include shared track sections and stations. The tracks, particularly the shared sections, act as shared resources. Without proper synchronization, both trains could attempt to occupy the same section simultaneously, leading to a collision in the simulation. Deadlocks are a significant risk. With two trains, this could occur if both attempt to enter the same shared section from opposite ends and wait indefinitely for the other to leave. Similar to shared sections, stations are points where special synchronization is needed. Both trains might need to use the station simultaneously.

To solve the critical sections challenges we are going to use both active waiting to ensure the access to only one train and mutual exclusion to ensure that the shared resources are accessed correctly.

3 Conception

3.1 Shared sections

To address station synchronization, we implemented a barrier mechanism. This ensures that trains can only leave the station when it is fully occupied. Additionally, we integrated a priority system within this barrier, giving precedence to the last train arriving at the station to depart first when the barrier is lifted. The synchronization barrier can be implemented using semaphores (which may also act as mutexes) and boolean flags.

3.2 Station

For managing the shared section, we established a synchronization logic which ensures that while one train occupies this critical section, the other train waits. As soon as the first train exits the section, the waiting train receives clearance to enter. This approach effectively prevents conflicts and ensures safe passage through the shared track section. Similarly to the shared sections, the synchronization logic can be implemented using semaphores and boolean flags.

4 Implementation

4.1 Routes

The different routes that are implemented in the application are generated by the `routeFactory` method in the `cppmain.cpp` file. This method takes a `RouteType` as a parameter and returns a `Route` object. The `RouteType` is an enum that defines the different routes that can be generated. The `Route` object is a structure that contains the parameters necessary to successfully drive a route, including the station contacts, one or more shared sections or the junctions that need to be set to drive the route. Altough the source code could be refactored to use a more object-oriented approach, it was decided that the current implementation was sufficient for the scope of this lab.

4.2 Program startup

In the main function, we initialize the model railway system, including setting up the model turnouts based on the selected route. Locomotives A and B are initialized with unique identifiers and predetermined speeds, and their starting positions are determined by the route configuration. Then, two threads launch an instance of the runnable locomotivebehavior class. In this class, the run() method is responsible for dictating the locomotive's actions, encompassing station stops, navigation through shared sections, and responses to track junctions.

4.3 Shared sections

Synchronization of shared track sections is a critical aspect of our implementation. The `access()` and `leave()` methods of the `Synchro` class ensure that only one train occupies a shared section at any given time, effectively preventing the risk of collisions.

The access to the shared section a rather classical active wait implementation using a semaphore. A boolean flag signals whether the section is free or occupied. If the section is occupied, the train must wait until the section is released by the other train. The critical section of the method is protected by a mutex to ensure that only one thread at a time my modify the shared variables. If the section is not free, the locomotive must stop before doing it's active wait.

```
1  void access(Locomotive& loco) override {
2      mutexSection.acquire();
3
4      if (!isSectionFree) {
5          loco.arreter();
6          otherIsWaiting = true;
7          mutexSection.release();
8          sectionSemaphore.acquire(); // Blockingly wait for the section to be free.
9          mutexSection.acquire();
10         otherIsWaiting = false;
11         loco.demarrer();
12     }
13
14     isSectionFree = false;
15     mutexSection.release();
16 }
```

Listing 1: The shared section access routine.

The release of the shared section is quite straightforward. The critical section is protected by a mutex to ensure that only one thread at a time my modify the variables and, if another train is waiting on the section, it is released.

```
1  void leave(Locomotive& loco) override {
2      mutexSection.acquire();
3      isSectionFree = true;
4      // Give access to the other loco if it is waiting.
5      if (otherIsWaiting) {
6          sectionSemaphore.release();
7      }
8      mutexSection.release();
9  }
```

Listing 2: The shared section exit routine.

4.4 Station

To ensure orderly operations in the station, we implemented a barrier mechanism. This approach not only manages station occupancy but also gives priority to the last train that arrives at the station. The barrier uses a boolean status flag that indicates if the station is already occupied or not. The reading and writing to the flag are protected by a mutex. If the current thread is not the last trained arrived in the section, it will blockingly wait for the second train to arrive using a semaphore. Once the second train has arrived, the related thread takes a 5 second sleep and then gives itself priority on the shared section by calling the `access()` method and setting its priority to 0 (highest priority). Once the priority is set the arriving thread releases the other thread, at which point both locomotives can resume their routine.

```
1        void stopAtStation(Locomotive& loco) override {
2
3            // If the station is empty, stop the loco and set the station to occupied.
4            mutexStation.acquire();
5            if (!isStationOccupied) {
6                isStationOccupied = true;
7                mutexStation.release();
8                loco.arreter();
9
10               // Wait to be released by the last arrived loco.
11               stationSemaphore.acquire();
12
13               // Set the priority
14               loco.priority = 1;
15               loco.demarrer();
16           } else {
17               // If the station is occupied, wait for 5 seconds and have the last
18               // arrived loco start first by already giving it access to the shared section.
19               isStationOccupied = false;
20               loco.arreter();
21               PcoThread::usleep(5e6);
22
23               // Give access to the shared section before releasing other locos so
24               // that we're sure it can't be acquired by another loco.
25               access(loco);
26
27               // Release the semaphore to signal the first loco that it may continue.
28               stationSemaphore.release();
29
30               // Set the loco to the highest priority and restart it.
31               // Note: this is useful in the LocoBehavior::run() while loop in order to avoid the
32               // prioritized loco trying to acquire the shared section again.
33               loco.priority = 0;
34               loco.demarrer();
35               mutexStation.release();
36           }
37       }
```

Listing 3: The shared section station synchronization routine.

4.5 Emergency stop

An emergency stop functionality is implemented. This function can be invoked to immediately halt both locomotives, setting their speeds to zero and ensuring an instantaneous stop.

```
1  void emergency_stop() {
2      locoA.arreter();
3      locoB.arreter();
4      locoA.fixerVitesse(0);
5      locoB.fixerVitesse(0);
6  }
```

Listing 4: The emergenty stop routine.

4.6 Program exit

The simulation runs continuously with the locomotives checking if the shared sections can be accessed and – if not – waiting to be able to access it. Only the activation of the emergency stop brings the operation to a halt, but without closing the window. The simulation is gracefully terminated upon the activation of the *close* button, ensuring the proper shutdown of all threads and the safe state of the train model system.

5 Tests

5.1 Shared section

The testing of the train simulation system is conducted in a series of different routes to ensure the robustness and reliability of the implementation. These tests were designed to cover a wide range of scenarios, each introducing different complexities and challenges.

1. **Provided train route:** Initially, we tested the system using the provided default train route. This served as a baseline to verify the fundamental functionality of our implementation, including basic movement of the trains and basic synchronization in shared sections and at stations. The related configuration is `RouteName::ROUTE_1`.
2. **Route with only a station and a shared section:** We tested a minimalist route that included only a station and a shared section. This scenario allowed us to test that a train behaves correctly, even when it must handle both the shared section and the station at once. The related configuration is `RouteName::ROUTE_2`.
3. **Route with a shared section right before the station:** Another scenario placed the shared section right before the station. This setup was aimed to test the system's handling of the synchronization mechanism when trains transition quickly from a shared section to a station stop. The related configuration is `RouteName::ROUTE_3`.
4. **Route with a shared section separated from the Station:** A route where the shared section was separated from the station by a few blocks was also tested. This scenario was aimed at evaluating the system under a more spread-out configuration, ensuring that our synchronization logic remained effective. The related configuration is `RouteName::ROUTE_4`.
5. **Route with two shared sections:** Going further, we introduced a route featuring two shared sections. This more complex scenario tested the system's capability to manage multiple critical sections simultaneously, ensuring that each train could access and leave these sections and without conflicts or deadlocks. The related configuration is `RouteName::ROUTE_5`.

5.2 Station synchronization

We validated that the last train to enter the station was the first one to leave the station. Furthermore, we've validated that the locomotive that has to wait in the station doesn't hiccup while starting (it must start and stop to wait to access the shared section in a smooth fasion).

Emergency stop

We tested the emergency stop in different scenarios with the inertia of the model taken into account:

- Just before the station. Inertia makes the train enter the station, which correctly triggers the `stopAtStation` routine, but the trains stay still even though being allowed to leave.
- Just before the critical section warning contact. Inertia makes the train acquire the section and correctly sets the boolean flag indicating that the section is busy. The train does not restart as a result of acquiring the section.
- When only one train is the station.
- When both trains are outside the station.
- When both trains are outside the critical section.

6 Conclusion

In conclusion, the train simulation system successfully demonstrates the application of concurrent programming principles to a complex, real-world scenario. By employing semaphores and synchronization techniques, we effectively managed shared resources and ensured safe, conflict-free operation of the two trains on a single track system. The diverse testing scenarios further validate the robustness of the proposed implementation.

Although not implement, the synchronization techniques used in this lab may easily be extended to accommodate shared sections with an indefinite amount of entry and exit paths.

Appendix

A Train routes

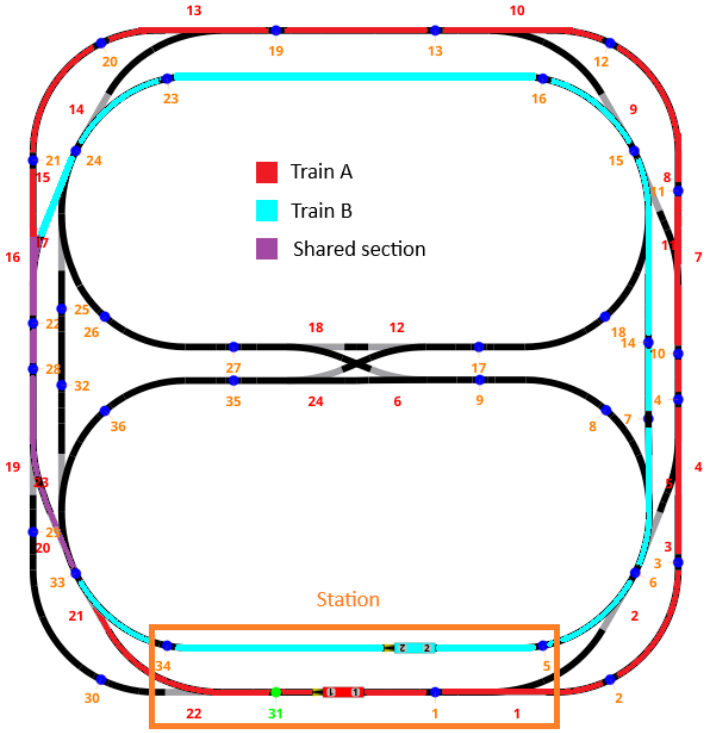

Figure 1: Train route 1

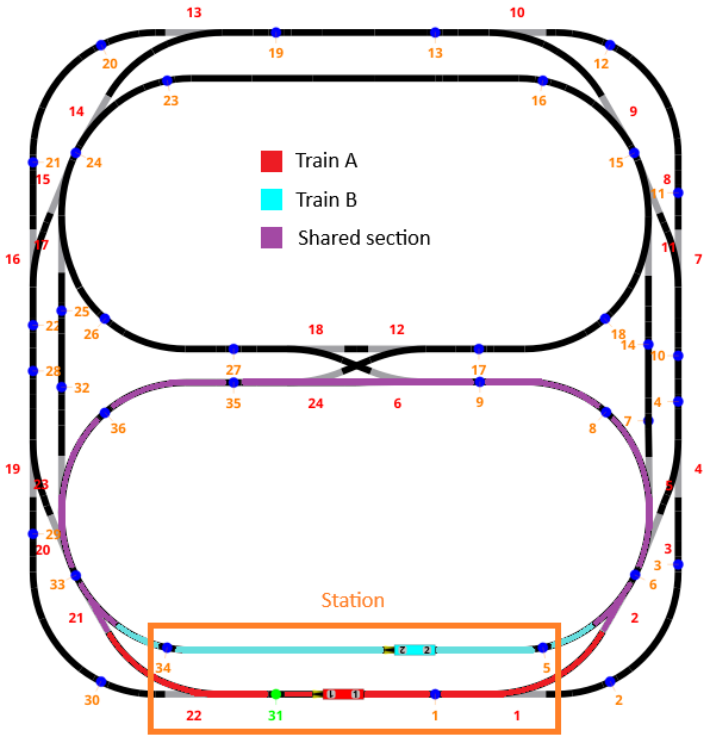

Figure 2: Train route 2

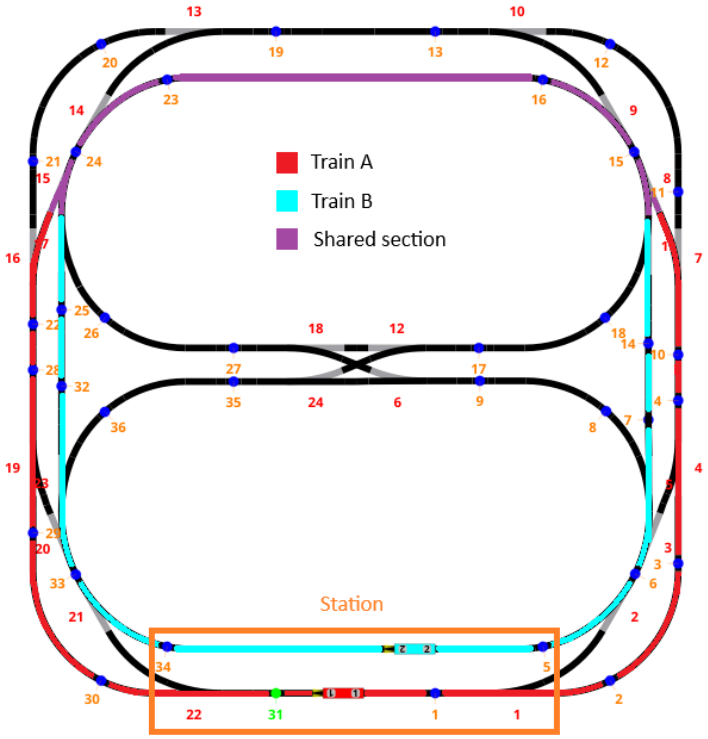

Figure 3: Train route 3

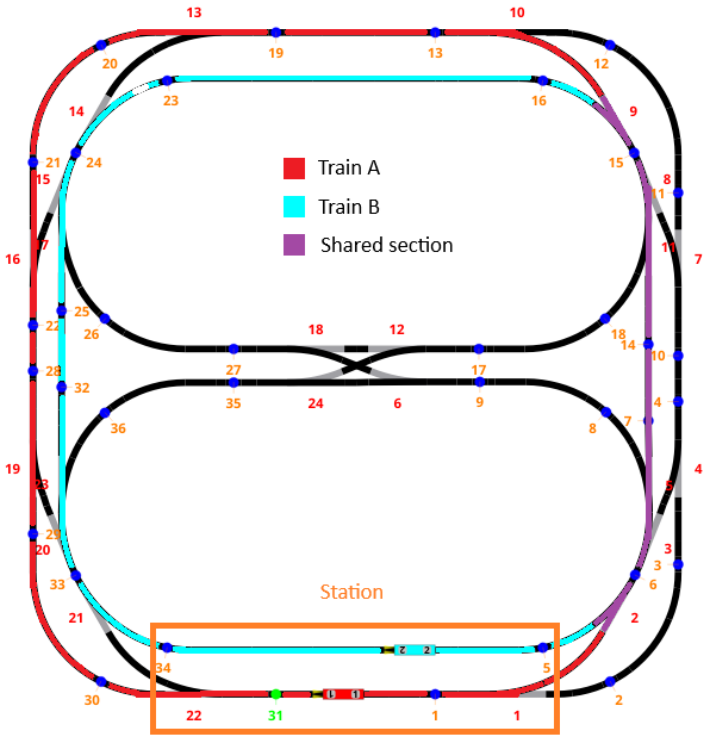

Figure 4: Train route 4

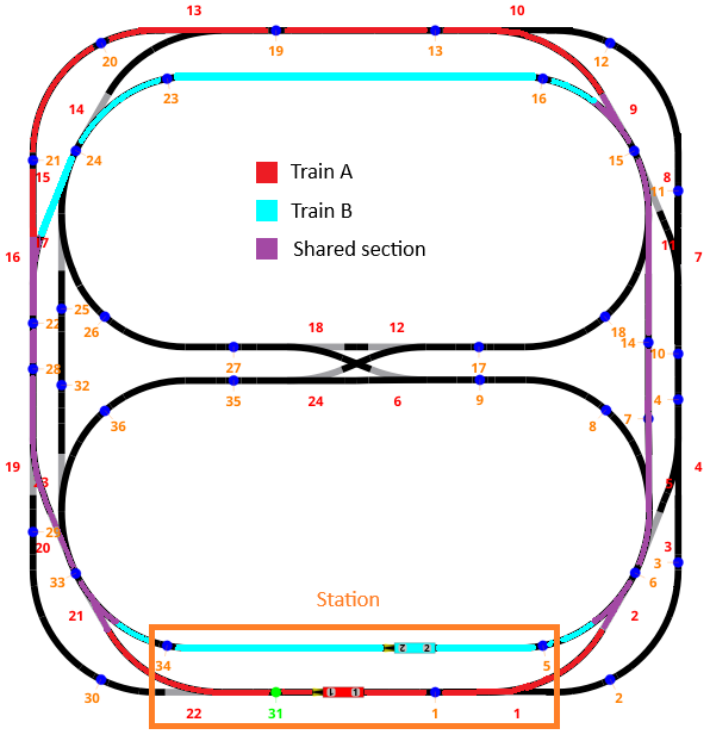

Figure 5: Train route 5