TRAIN JOURNEY MANAGER

REPORT

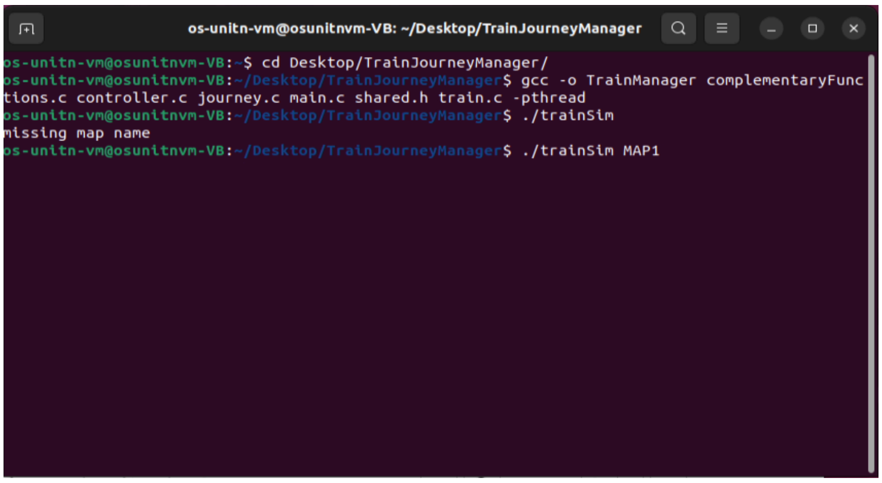
DAVID BRUGNARA
david.brugnara@studenti.unitn.it
mat. 235236

Operating Systems A.A. 2023/2024
I.C.E. Engineering

UNIVERSITY
OF TRENTO

HOW TO COMPILE THE PROJECT

Once you have extracted the folder containing all the source codes, you can proceed to compile the program. Enter the folder "TrainJoruneyManager", where we need to compile and link all different codes together. Invoke the command gcc, with the tag -o you can specify a custom name for the executable, for example TrainManger. Next we're going to list all 6 codes, including their extensions. Once all files have been listed, it's crucial to add the tag -pthread, to ensure that the linker will find the correct library.



```
os-unitn-vm@osunitnvm-VB:~$ cd Desktop/TrainJourneyManager/
os-unitn-vm@osunitnvm-VB:~/Desktop/TrainJourneyManager$ gcc -o TrainManager complementaryFunc
tions.c controller.c journey.c main.c shared.h train.c -pthread
os-unitn-vm@osunitnvm-VB:~/Desktop/TrainJourneyManager$ ./trainSim
missing map name
os-unitn-vm@osunitnvm-VB:~/Desktop/TrainJourneyManager$ ./trainSim MAP1
```

Now that the executable is ready, simply use the command ./ followed by the desired name to execute it. Don't forget to add the MAP-CODE, as of now, only MAP1 and MAP2 exist.

TESTED ENVIRONMENTS

The heavy lifting has been done working on a amd-based WINDOWS 11 machine (16 cores, 32GB of RAM), using a C++ version of the program, not using any OOP features, to simplify the conversion to C.
Once the program has been proven stable, it has been imported to a **simulated** UBUNTU (22.04.3) environment on virtualBox for further testing. The virtual machine has been given 8 virtual cores to work with and 12GB of RAM.

DESIGN AND IMPLEMENTATION

This project is explicitly focused on threads, so the first element to discuss is what kind of threads have been used. The main options at disposal were POSIX threads, forks and OpenMP. Forks would have introduced too much overhead while OpenMP's high level of abstraction might have back-fired, so the industry standard POSIX threads were chosen for their relatively-easy management and communication capabilities.

The program has been split into 6 files, here's a quick rundown on each file's purpose.

1. **main.c**; as the name suggests, it is the body of the program, in charge of launching the controller and the journey process, initializing shared variables and synchronization tools.
2. **shared.h**; this is a header file, used to remove clutter from main and, import libraries and group all functions and variables used across the program.
3. **controller.c**; its purpose is simple, initialize all track segments, launch the train processes and wait for them to finish.
4. **train.c**; this is the first file to contain multiple functions, first we have the main body of the train thread, which the train uses to communicate with the journey thread to gain access to its itinerary, then calls the second function. A function whose job is to get the train moving from one segment to the next, while checking the availability of each segment. All of this could have been done inside the main train body, but separating the two makes for a clearer code.
5. **journey.c**; its job is short and simple, but crucial to the program. Deliver to each train its itinerary.
6. **complementaryFunctions.c**; stores 2 functions with marginal roles to the project, one is only used once by journey to read the .csv file containing the map while the other function provides the current time whenever a train process wants to track its progress in the log file.

The most error-inducing part of a programming using threads is synching access to shared memory, in this program there are 2 instances of concurrent access. The Train-Journey communication, where the train threads need to regulate access to the shared itinerary, second is the access to the segment status variable, in order to check whether or not the train has the right to advance closer to its destination. Let's dive deeper in both instances.

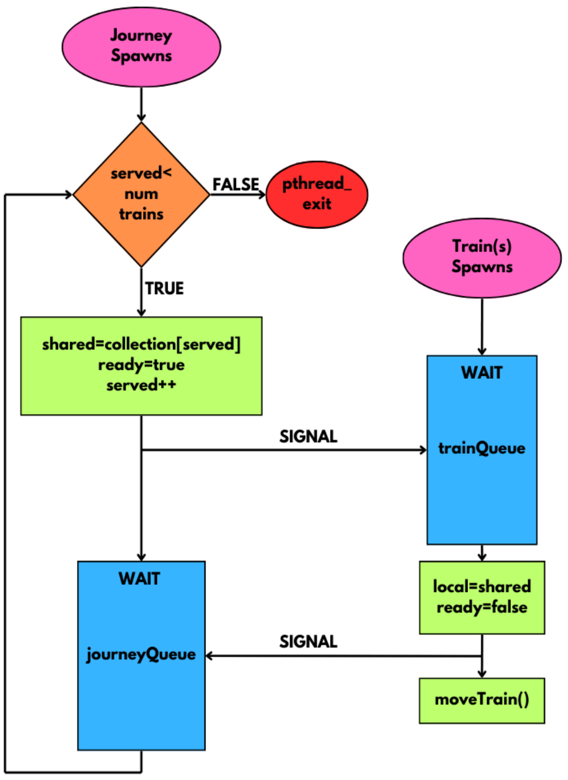
Journey-Train communication

As established earlier, each train needs to communicate with *Journey* in order to get a copy of its itinerary, the underlying idea is to use a shared variable ***shared***, where for each train *Journey* loads one itinerary, waits for a train to copy it before loading the next one. This idea has been implemented with the use of one mutex, ***comm_mutex***, 2 conditions used to create a special queue, a ***ready*** flag used to materially check on the conditions and 1 progress counter called ***served.***

Since unregulated access to these variables is bound to create inconsistency problems we need to establish that only one process at the time is able to read and modify these mentioned variables.

Where the initial design required 3 queues and 2 flags to grant data consistency with some tinkering a lighter version was made possible.

served is initialized as 0, while ***ready*** as *false*. So when the trains get launched they will all enter the train queue while accessing the mutex lock, where they will stay sleeping (soft lock release) until Journey is done setting up the shared variable and will wake up a train thread with the command *signal*. Journey himself will go to idle in its queue until the just awakened train is done copying the itinerary in its local memory. Once the train thread has its own copy of the itinerary stored and has set the flag back to its original state, it's free to leave the critical section, but not before waking up Journey with a signal.



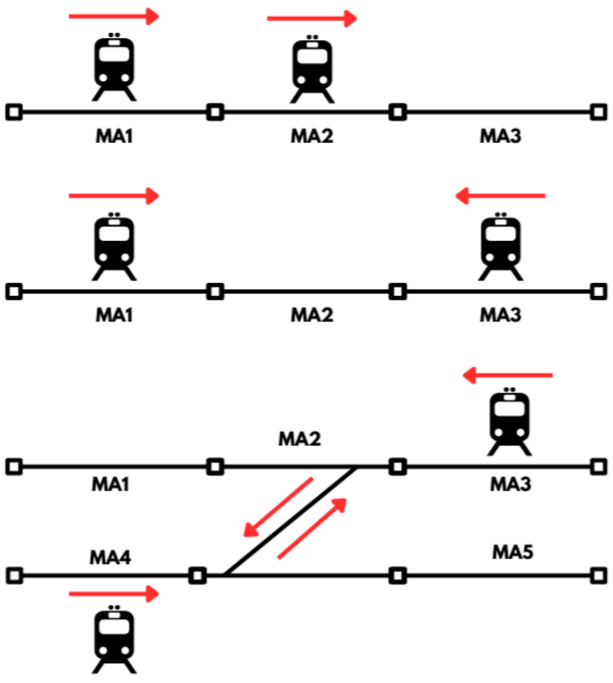
Segment availability

We know that there are 16 individual track segments, and each segment has its own binary (0/1) status associated to. Obviously each train has to check the status of the segment in order to proceed, but what if two trains want to read the status of the same segment? A data race occurs, so the logical answer is to implement a mechanism to secure data access. The two most intuitive solutions are a single lock to protect the whole status array, or an array of locks, one for each element of the array, so that they could be locked individually. This program uses an array of locks. With the current size of maps and number of trains, the two options grant virtually the same performance, but if we were to increase the amount of trains, a single lock would introduce an intolerable amount of contention and as result waiting times would skyrocket.
More specifically in the implemented locking system, deadlock and starvation are prevented by the trains always releasing the lock immediately after reading or writing its value, and always before sleeping. Each train keeps track of 2 indexes, one for the current segment and one for the next segment. Before freeing the current segment, the train will always need to be granted access to the next, thus "owning" two different segments at once for a short period of time. This makes for an overall secure system.

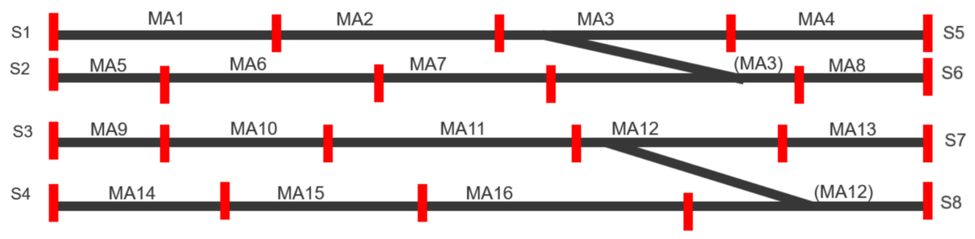
PROGRAM EXECUTION

Now that the trickiest parts of the code are clarified, and the rest of the code is explained in the source code using comments, it's time to look at the end results.
As established earlier, deadlock on data is prevented, but this does not prevent trains from blocking the tracks. There are 3 types of collision possible.

- Type 1 consists of 2 trains heading the same direction, collision will be avoided by the system, with the second train always going to sleep after seeing the next segment occupied, this will drag on for the entire track. Both trains will arrive at their destination.
- Type 2, is essentially impossible to manifest with a well planned map, which takes in consideration the structure of the railroad.
- Type 3, is a fatal collision, where 2 trains are blocking each other with no way out and should be avoided with careful map planning.



Let's discuss MAP2, which is more complex than MAP1. There's fundamentally 2 knots that will cause delays. It's clear that the railway is divided in 2 disjointed sections, the upper one and the lower one. In MAP2, the lower one gets populated by T2 and T3, who start from S3 and S4 respectively, both moving towards S8. This will play out in 2 ways, depending on computational time either T2 or T3 will have access to MA12, shortly after it will be freed and ready to be used again.



The upper section is more brittle, MA3 will be used by 3 trains, T1 while going in a straight line from S2 to S6, T4 while going from S6 to S1 and T5 while going in a straight line from S5 to S1. Due to MA3's position on the map, it's clear that T4 and T5 have a favorable position to reach MA3. In most of the executions MA3 will be disputed between the two, and T3 will only arrive when both T4 and T5 have already left MA3. But sometimes (around 1/16 executions), due to computational timing, T1 will reach and lock MA3 while T4 is still inside MA8. Effectively a type3 collision, manifesting in a fatal collision. This leads us to a question.

Is the rail system intrinsically flawed?

The rules which the train system has been built upon are clear;

- All trains leave the starting station at the same time
- Trains cannot deviate from the route specified in the MAP
- A segment can only host a single train at the time
- Each segment has a status variable
- If multiple trains want to access the same segment, only one will be able to enter, the rest have to wait

Regarding the last rule, in the implementation details we can read; "when permission is denied, does nothing and goes to step C", where step C says "sleeps 2 seconds".

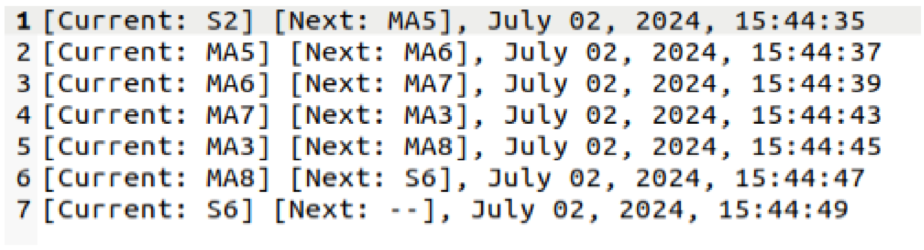
Collisions are fundamentally very similar to deadlocks, but to be clear, in the previous example, T4 and T1 can still periodically check each other's segment status, **there's no data deadlock**

Possible solutions

By ignoring the rules we could implement a priority ranking, asking access to multiple segments at once, placing “safe zones” where trains can wait out without actively occupying the segment, giving the train the agency required to move back to its previous segment, or implement a queue, where if 3 threads try to access the segment, the first one will be able to occupy it, and the second one will be able to reserve it for himself next. But all these solutions would be contradictory to the rules the train has been given, where the access to a segment is a cycle of single requests, separated by 2 seconds intervals, with no long term planning, and no deviations from the assigned itinerary.

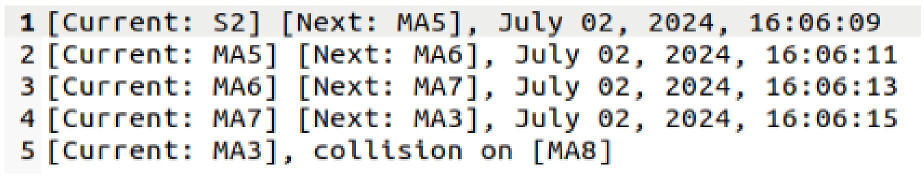
A collision detection mechanism able to detect and terminate colliding trains has been added.

T1 when no collision occurs,



```
[Current: S2] [Next: MA5], July 02, 2024, 15:44:35
[Current: MA5] [Next: MA6], July 02, 2024, 15:44:37
[Current: MA6] [Next: MA7], July 02, 2024, 15:44:39
[Current: MA7] [Next: MA3], July 02, 2024, 15:44:43
[Current: MA3] [Next: MA8], July 02, 2024, 15:44:45
[Current: MA8] [Next: S6], July 02, 2024, 15:44:47
[Current: S6] [Next: --], July 02, 2024, 15:44:49
```

T1 when collision occurs,



```
[Current: S2] [Next: MA5], July 02, 2024, 16:06:09
[Current: MA5] [Next: MA6], July 02, 2024, 16:06:11
[Current: MA6] [Next: MA7], July 02, 2024, 16:06:13
[Current: MA7] [Next: MA3], July 02, 2024, 16:06:15
[Current: MA3], collision on [MA8]
```