Train Journey Manager - Project Report

Name: Giovanni
Surname: Foletto
Registration Number: 218065
Email Address: giovanni.foletto@studenti.unitn.it
Delivery Date: February 7, 2025

---

1. Compilation and Execution Instructions

Compilation: To compile the file, the only action required is to launch make from the project directory.
The executable file will be created (built) with the name train_manager.

Execution: To execute the file, you can launch ./train_manager MAP1 or ./train_manager MAP2.
The two files (MAP1.txt and MAP2.txt must be present in the current file directory, or the program will fail with an error message and the execution will not be continued. The only thing that should be inserted is the name MAP1 or MAP2 since the program automatically adds .txt at the end of the filename.
The parser of the maps is calculated on the examples given and there are no guidelines against inserting files with different specifications. In that case, the program will not stop, but rather (probably) throw a SIGFAUL during execution. More on that in the Design and Implementation.

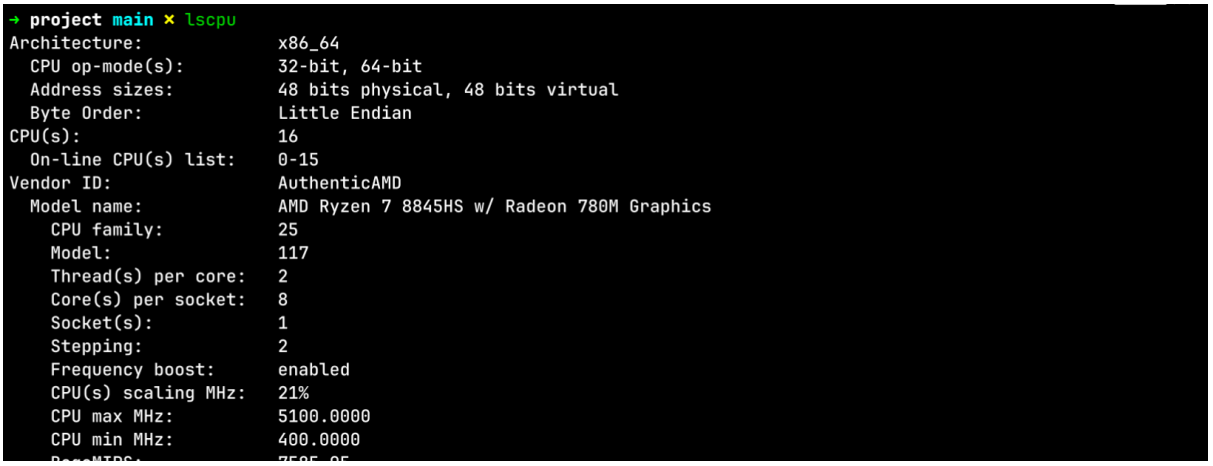
2. Target System

The whole project is developed with these specifications:

- **Operating System:** Fedora release 41 (Forty-One)
- **Kernel-Version:** Linux fedora 6.12.11-200.fc41.x86_64
- **Hardware Specifications:** AMD Ryzen 7 8845HS w/ Radeon 780M Graphics
- **Required Libraries:** the whole project uses only POSIX interfaces (sem_t and forks instead of pthreads)

More information:

```
→ project main × uname -a
Linux fedora 6.12.11-200.fc41.x86_64 #1 SMP PREEMPT_DYNAMIC Fri Jan 24 04:59:58 UTC 2025 x86_64 GNU/Linux
→ project main × cat /etc/redhat-release
Fedora release 41 (Forty One)
```



Libraries used:

- #include <fcntl.h>
- #include <semaphore.h>
- #include <stdio.h>
- #include <stdlib.h>
- #include <string.h>
- #include <sys/ipc.h>
- #include <sys/mman.h>
- #include <sys/msg.h>
- #include <sys/shm.h>
- #include <sys/types.h>
- #include <sys/wait.h>
- #include <time.h>
- #include <unistd.h>

3. Design and Implementation

The program is entirely included in the file main.c. The functionality required is implementable in around 250 LoC, so dividing it into small files, although feasible, is quite unproductive. The result will be a lot more code to handle the fact that information is in different files. In particular, there are problems when declaring sem_t in the train_manager.h, the linker complaint about a “Violation of ODR”. This ODR rule is from the C98 standard and is inserted to avoid confusion while allocating shared resources in the headers file and using them in the main. I think that the problem should be solved by implementing all like a “library” in which the main is responsible only for getting the argv values inside the run_library function.

The entire system architecture is thought out as a keep-it-simple solution, so the data flows gracefully around all the moving parts, and is easy and fast to add/remove functionality while allowing fast development. In this case, I worked out adding a feature at a time and having that tested as simply as possible before continuing.

Each process is generated by forking, and then it is implemented a message-passing technique using sys/msg.h. Blocking tracks happen using POSIX semaphores and shared memory is using the default POSIX API, instead the System V.

System Architecture

The program consists of (in code the name of the functions in the code):

1. **Controller Process: main**
    - Creates and manages shared memory for track segments.
    - initialize the trains_data by loading information from textfiles.
    - Spawns five train processes.
    - Keep track of all processes generated and initialize a waitpid for each one, collecting the resulting return code.
2. **Journey Process: journey_process**
    - Get information about routes from load_routes (while dynamically creating filenames for input using only the initial filename without the .txt extension).
    - Opens message queues to pass messages and then creates N messages for N number of trains.
    - The messages passed contain all the information gotten from load_routes, including:
        i. itinerary: a list of values that represent the number of tracks to pass into in order
        ii. num_steps: the number of sectors the train has to move into to arrive at the destination
        iii. start/dest: information about the starting and ending station's name
3. **Train Threads (TRAIN_THR): train_process**
    - This function gets as its first thing the message provided by the journey process. It waits if the message is not already present or fails if it cannot get it.
    - It opens the shared memory location, using the SHM_NAME common information, and mapping the information in its own running memory space using mmap
    - Create or open the file to log information into creating the filename dynamically using the (really useful) snprintf function (allows to use of common description information of the printf function to insert data inside a string).
    - It starts the journey, by requesting a lock on the first segment it has to move into, and then unlocking the current segment it is on (except in the starting phase, in which the train is in a station, that is allowed to have multiple trains in it)
        i. This process uses sem_wait allowing the train to completely stop in the current segment while it is occupied. This allowed a more efficient method instead of using sem_trywait and continuously pulling. Instead, this function will return zero only when the semaphore can be taken.
        ii. The logging information is a bit tricky, solved using a fprintf on the log file descriptor, and then evaluating each component of the string from the current train state (current_segment and message struct from the message receive process). The other information are gotten from the time and localtime functions, with a bit of a hacky way of displaying the months (a simple array of strings compiled manually).
        iii. After the train has acquired the next segment, we can free the last used, using sem_post
    - For the current implementation, the train maintains the lock on the last segment if it has arrived at the station. So we can now unlock it after the route is completed.
    - After all actions are completed, we must only clean up the shared memory used, “unsubscribing” to that (munmap) and then closing the file descriptor. These two functions are why I choose to use POSIX shared-memory API since I find a lot more interesting handling shared memory like a file (more in a Linux-way-of-doing thing) instead of the pointer to the memory location used by SystemV.

Synchronization & Parallelism

To handle synchronization and parallelism I added to the file semaphores supports and shared memory. As I was saying before, each train to enter a segment must require a semaphore to be locked. When this action happens, then the train can sit in the “shared segment” (represented by a shared memory segment) and change the value of that as “occupied”.

There is no method of synchronizing the train at the departure as requested. This is difficult with forking processes. This will be simple to create using pthread and its functionality of pthread_barrier.
The deadlocks are avoided using semaphores. Since the type allows only atomic operations, all the complexity against deadlocking is externalized to the semaphore library instead of being implemented internally in this project.

4. Execution and Results

Sample Execution Output

Upon execution, the train_manager process output on the stdout only these information:

```
➜ project main × ./train_manager MAP1
Train 4 reached its destination.
Train 3 reached its destination.
Train 1 reached its destination.
Train 2 reached its destination.
Train 5 reached its destination.
➜ project main ×
```

In the current directory have been created N files (as N is the number of trains) that contain something like this:

```
➜ project main × cat T1.log
[Current: S2], [Next: MA5], February 07, 2025 18:21:10
[Current: MA5], [Next: MA6], February 07, 2025 18:21:12
[Current: MA6], [Next: MA7], February 07, 2025 18:21:14
[Current: MA7], [Next: MA3], February 07, 2025 18:21:16
[Current: MA3], [Next: MA8], February 07, 2025 18:21:18
[Current: MA8], [Next: S6], February 07, 2025 18:21:20
[Current: S6], [Next: --], February 07, 2025 18:21:20
```

Observations & Issues Resolved

The concurrency is handled by using shared memory and semaphores, allowing smooth concurrency capabilities for the program. Deadlocks are avoided using the semaphore POSIX implementation that allows for atomic operations thus avoiding deadlocks.
The performance of the programs is as good as possible since the only time the processes have to wait is the one while waiting for the semaphore to unlock.

Conclusion

This project successfully simulates a train journey management system, handling concurrent train movements efficiently using process synchronization techniques. Further optimizations can include dynamic segment reallocation strategies and priority-based train scheduling.