OPERATING SYSTEMS

MUTUAL EXCLUSION

Description

A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex.

Mutex comes into the picture when two threads work on the same data at the same time. It acts as a lock and is the most basic synchronization tool. When a thread tries to acquire a mutex, it gains the mutex if it is available, otherwise the thread is set to sleep condition. Mutual exclusion reduces latency and busy-waits using queuing and context switches. Mutex can be enforced at both the hardware and software levels.
Disabling interrupts for the smallest number of instructions is the best way to enforce mutex at the kernel level and prevent the corruption of shared data structures. If multiple processors share the same memory, a flag is set to enable and disable the resource acquisition based on availability. The busy-wait mechanism enforces mutex in the software areas. This is furnished with algorithms such as Dekker's algorithm, the black-white bakery algorithm, Szymanski's algorithm, Peterson's algorithm and Lamport's bakery algorithm.

Major Functions

We have decided to improve the train efficiency by automating not just the trains but also the
passengers. From now on, passengers will be robots. Each robot and each train is controlled by a thread. You have been hired to write synchronization functions that will guarantee orderly loading of trains.

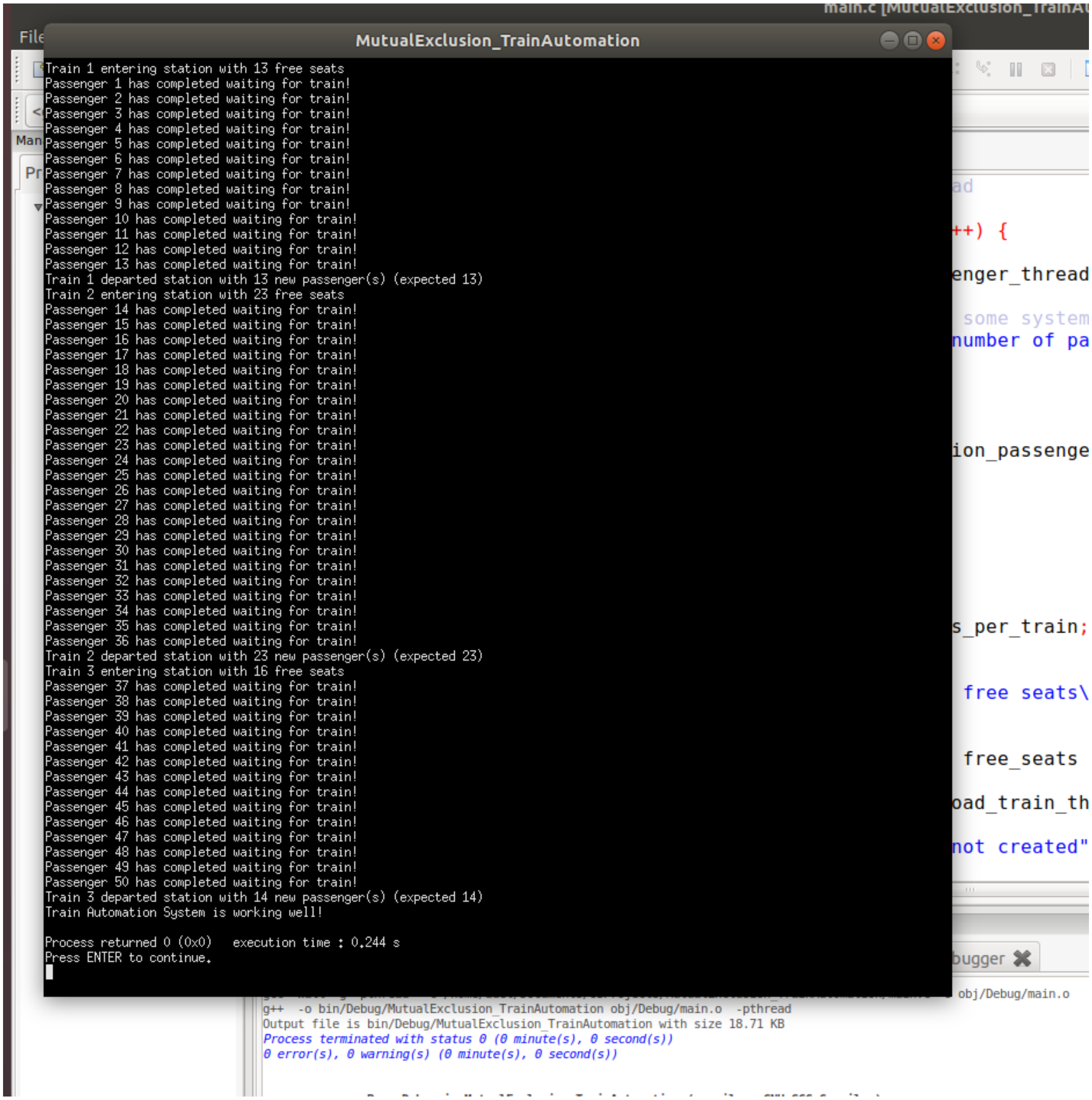
Organization of Code

Main Functions

The following part of the code composed into the following functions:

1. station_load_train(struct station *station, int count) : where count indicates how many seats are available on the train. The function must not return until the train is satisfactorily loaded (all passengers are in their seats, and either the train is full or all waiting passengers have boarded).
2. station_wait_for_train(struct station *station): where implementing the matrix multiplication using threads as described into part (a).Each created thread is responsible for calculating an element in the result matrix. That's why we are creating threads inside the second loop [when retrieving an element in the matrix]. We create a new threat using pthread_create function to compute a matrix element, and after finishing all computations we join all these threads created by their IDs by pthread_join function.
3. station_on_board(struct station *station): Once the passenger is seated, it will call the function to let the train know that it's on board.

Sample Runs & Screenshots



Thank You