Assignment 4 Report

Tanmay Goyal
AI20BTECH11021

Program Design:

The program begins with reading from the input file ‘inp-params.txt’ and getting the values of the number of passenger threads, number of car threads, λ_p, λ_c and the number of requests per passenger. If the file cannot be opened, we return an error.

We then define two classes to store the parameters to be passed to the threads: passenger_params; for the passengers and car_params; for the cars. Passenger_params consist of the following attributes: The passenger_index, the number of requests per passenger, the total number of cars, λ_p , and the pointer to the output file. Car_params consists of the index of the car, λ_c , the total number of rides, which is simply the product of the number of passengers and the number of requests per passenger, and the pointer to the output file.

The basic design is as follows: We have the following semaphores:

1) emptyCars – (counting) initialized to the total number of cars.
2) passengerLines- (binary) -> passenger at the front of the waiting line holds it.
3) totalNumberRides – (binary) -> used to update the total number of rides which is a shared variable.
4) occupiedCar[i] – (array of binary) -> tells whether the car i is occupied or not.
5) carLineUpdate- (binary) -> helps each car update the double ended queue of the line of cars.

We have an array called carStatus, which tells the status of each car: -5 if it is unavailable, -1 if it is waiting for a passenger, and >0 if it has a passenger. We also have a double queue called carLine which tells us which car is at the front to pick up a passenger. There is also an atomic<int> global variable to hold the total number of rides done.

The code runs as follows:

The passenger arrives at the museum. It wanders around for a random time, which is sampled from an exponential distribution with parameter λ_p. Once it is done roaming, it gets into line by waiting on the passengerLines semaphore. Once it acquires the passengerLines semaphore, it waits on the emptyCars semaphore. Once it acquires this semaphore, it means a car is waiting for a passenger. It shall acquire the carLineUpdate semaphore, get the details of the car at the front of the deque, remove the car from the front of the deque and release carLineUpdate. It then updates the carStatus of the particular car to its own number, acquires the occupiedCar semaphore for the particular car, and releases passengerLines for the next passenger in line to find its car. The passenger thread then busy waits till the ride is completed. This is indicated by the status of the car turning to -5 again (the car thread shall update its status to -5 once its done). Then, it shall increment the number of rides done, output to the log, and release the occupiedCar semaphore. The purpose of occupiedCar is to keep the car till the passenger is done incrementing the number of rides and outputting to the log, else in certain cases, the car is already available to the next passenger, which results in mismatch of number of rides done, resulting in the program never terminating. After all the rides for a passenger is done, it exits the museum.

On the car front, each car takes a break for a random time, sampled from an exponential distribution with parameter λ_c. It shall then update its status to -1, signal to emptyCars that it’s available to take a passenger, push itself in the queue of carLine, and signal to occupiedCar. The car thread then busy waits till it does not get a passenger. This is indicated by its status changing to >0 (which is done by the passenger thread). The car rides for a random amount of time between 1 and 5 seconds and signals the end of the ride by updating it’s status to -5. It then waits on occupiedCar semaphore till the passenger signals to it after it is done updating the total number of rides and outputting to the log. Once it acquires the occupiedCar semaphore, it again takes a break, and releases the semaphore only when it is ready to take a passenger again. We keep the car threads going till the number of rides done is not equal to the expected number of total rides. We ensure that we have a while loop to allow the thread to break out in case the condition is met. Also, the other place where the car thread can get stuck is while it’s waiting for a passenger. So, while busy waiting, we also allow the thread to keep an eye on the condition. If the condition is fulfilled, then we allow the thread to break out of their respective loops.

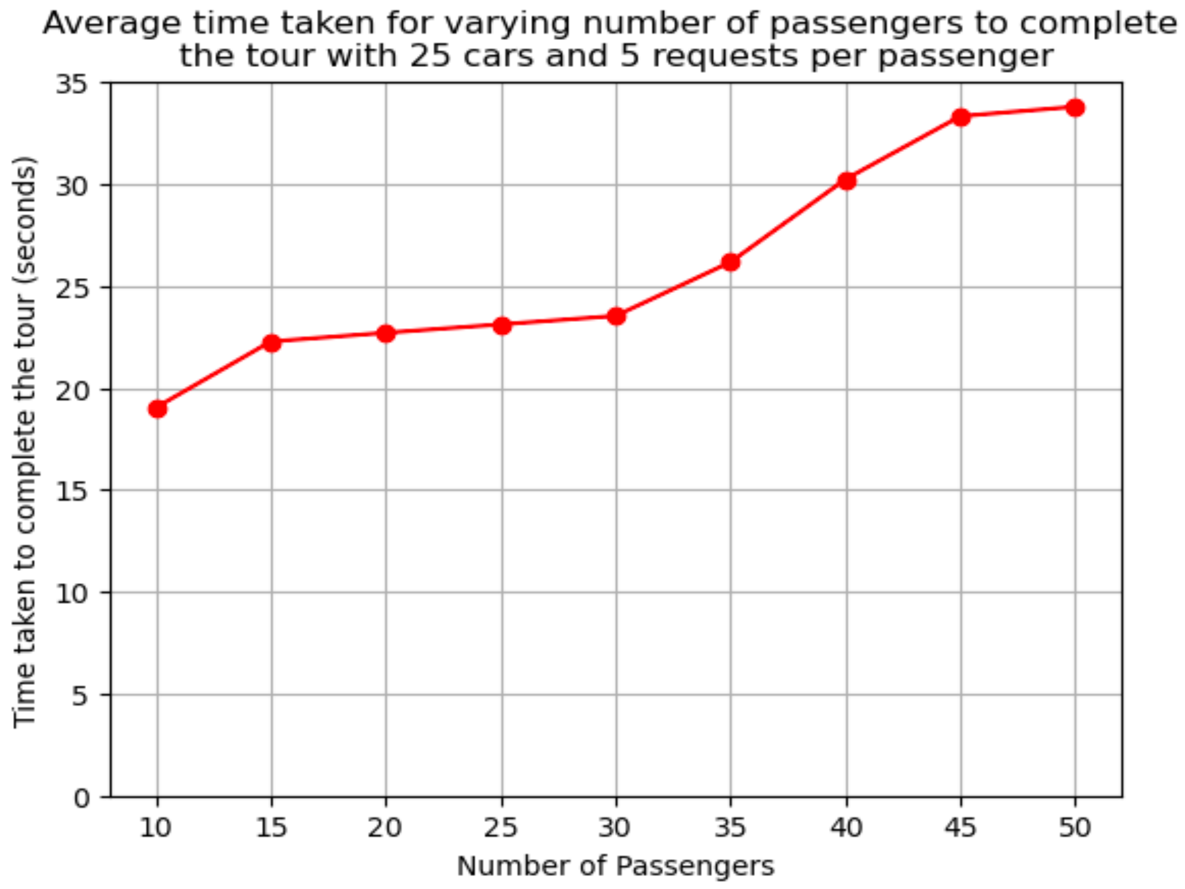
After all the rides are done, we join the car and passenger threads, and output the resulting time to the output log and the terminal for convenience. We also make sure to free all the memory on the heap, i.e. allocated through malloc, and destroy all semaphores.

Analysis:

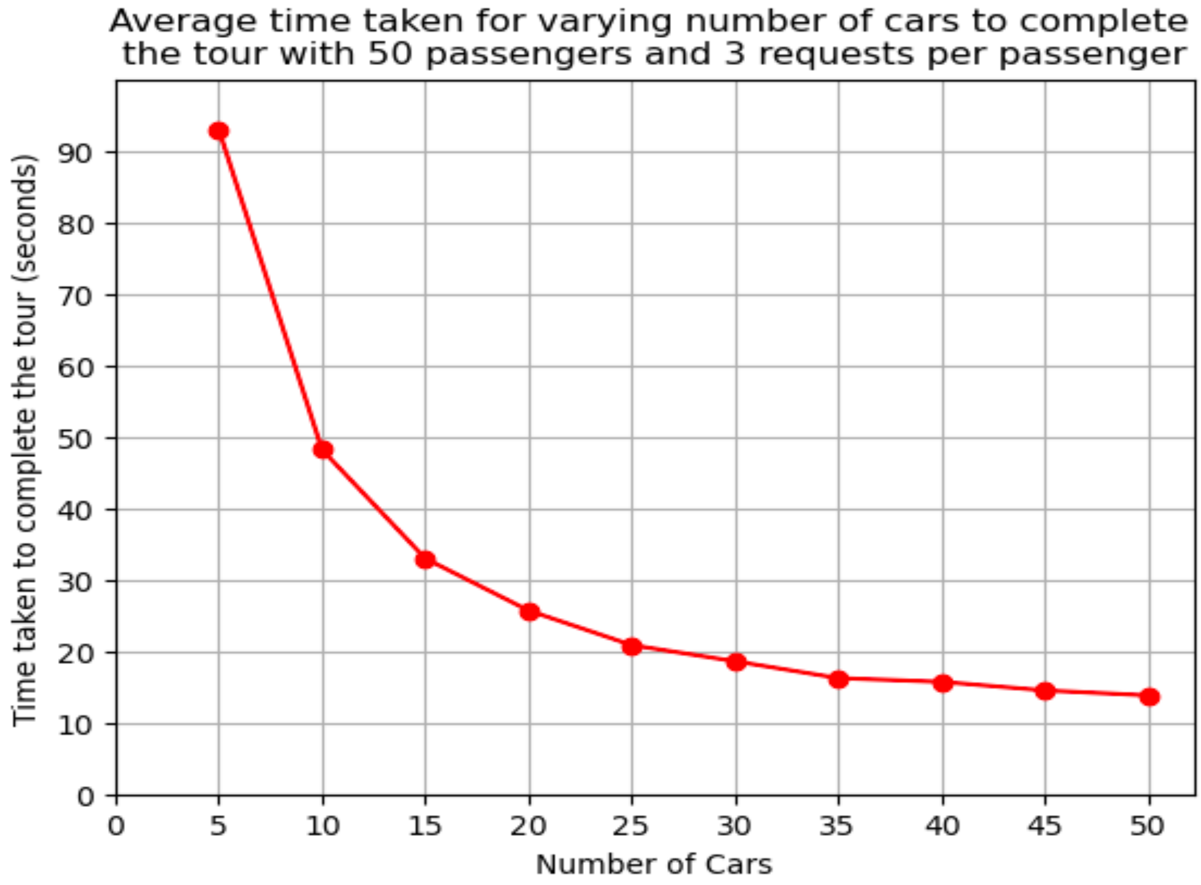
In both cases of analysis, we keep λ_p and λ_c fixed to 10 and 15 respectively to ensure uniformity in analysis. We also make sure each car ride is at random between 1 and 5 seconds.

In the first case, we keep the number of cars fixed to 25 and the number of requests per passenger fixed to 5. We vary the number of passengers from 10 to 50. We find that the increase in the beginning is not very significant because of the number of free cars. That is as the number of passengers increase to 25, all the cars begin getting utilized, and hence, there is not a significant increase in time, since none of the passengers must wait. However, after the number of passengers cross 25, there is a steep increase in the time required to complete the tour since passengers have to begin waiting for free cars.

For the second case, we keep the number of passengers fixed to 50 and the number of requests per passenger fixed to 3. We vary the number of cars from 5 to 50. We find that as the number of cars increase, the average time for the tour to end reduces. This is because as the number of cars increase, the number of passengers having to wait for a free car decrease. The decrease is steep in the beginning and reduces as the number of cars further increase. It is minimum when the number of cars is equal to the number of passengers.



Average time taken for varying number of passengers to complete the tour with 25 cars and 5 requests per passenger
Time taken to complete the tour (seconds)
Number of Passengers
0
5
10
15
20
25
30
35
10
15
20
25
30
35
40
45
50



Average time taken for varying number of cars to complete the tour with 50 passengers and 3 requests per passenger
Time taken to complete the tour (seconds)
Number of Cars
0
10
20
30
40
50
60
70
80
90
0
5
10
15
20
25
30
35
40
45
50