Name: Vishal Vijay Devadiga

Roll Number: CS21BTECH11061

Code Flow

- Opens input file in main and get value of p(number of passenger threads), c(number of car threads), k(number of iterations of critical section), and average values of sleep function λ_1 and λ_2
- Create p threads that execute the passenger function
- Create c threads that execute the car function.
- Passenger waits for a car to be available, then enters the critical section and notes the car available and removes it from the queue. It enters the critical section of the car, rides it(sleep), then exits the car. The passenger sleeps, after which it either makes another requests till it finishes all iterations of critical section, or exits the muesem. The car sleeps until it becomes ready for another ride, and then adds itself to the queue
- Each car thread waits till a passenger start riding it, then waits till the passenger exits, then waits till it beomes available to passengers.
- Car threads exit by a global variable when all passengers threads exit.

Code Design

I have used:

- 1 semaphore for the number of cars available
- 1 semaphore to update the car queue atomically
- c(number of cars) semaphores to denote whether the car itself is busy or not.

All available cars are stored in a queue. If a car accepts a request, then it is atomically removed from the queue. After finishing the ride, it atomically enters the queue.

Cars keeps checking whether their semaphore is activated or not, if activated it then waits till it acquires its own semaphore(that is, the passenger finishes the ride), then sleeps for an amount of time and then adds itself to queue and deactivates its own semaphore and increases the number of cars semaphore by 1.

Passengers wait for the number of cars semaphore, then enters the critical section and notes the car available(from the queue) and removes it from the queue and acquires the semaphore of that car. It then sleeps, then releases the semaphore(that is, exits the car), then sleeps before making another request.

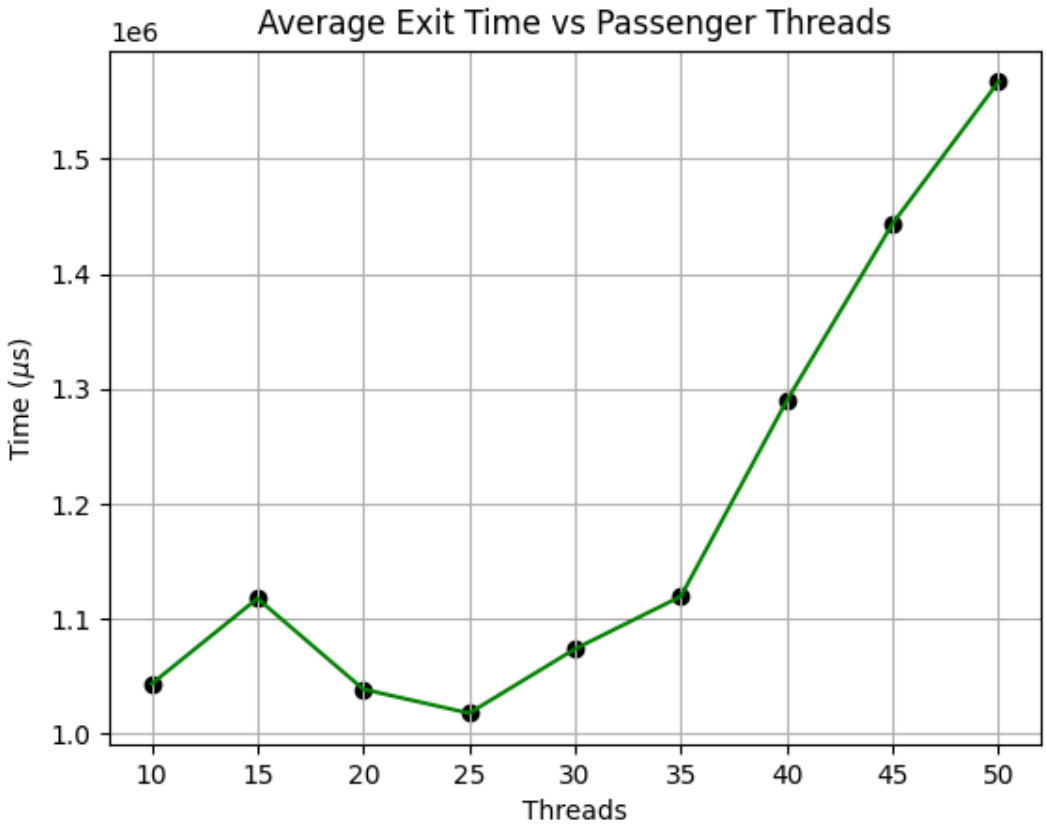
Plots

My system has the processor AMD Ryzen 7 5800u, which has 8 cores and 16 threads.

Note that **all measurements of time are in microseconds(μs)**

Values for $\lambda_1 = 100ms$ and $\lambda_2 = 80ms$

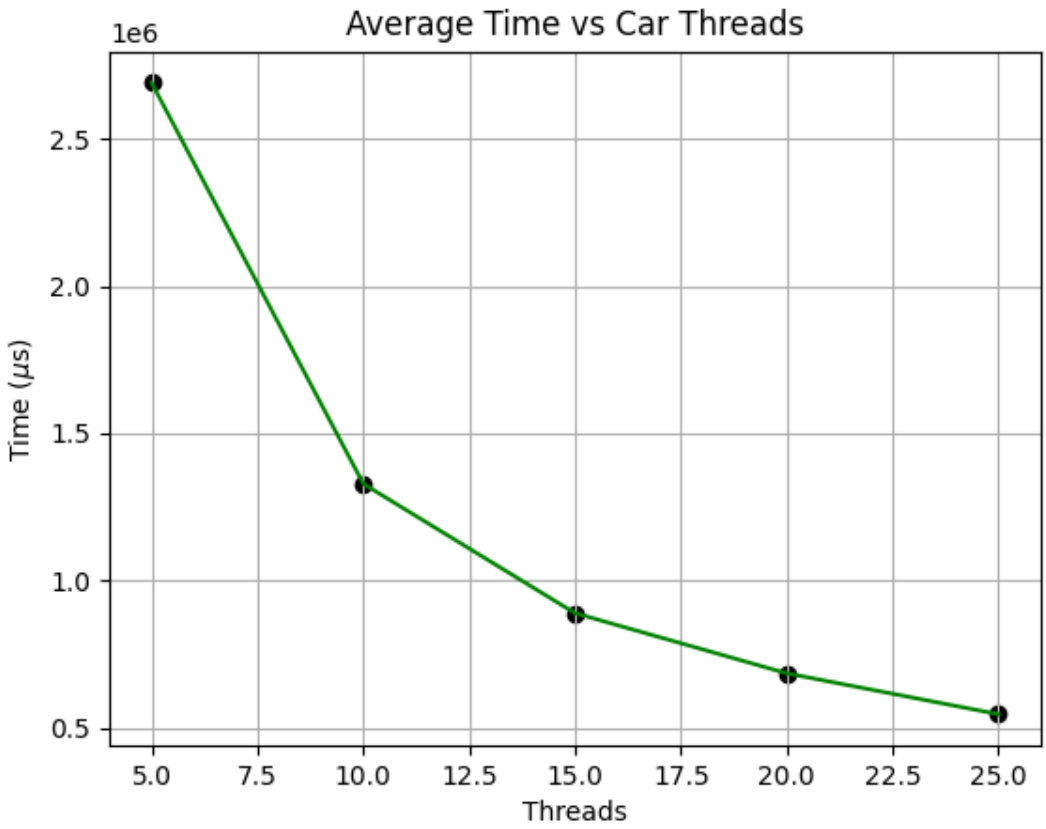
Graph for average tour time vs number of passengers



Passengers	T_1	T_2	T_3	T_4	T_5	T_{avg}
10	1030318.5	1065106.25	1021844.25	1036369.25	1063564.5	1043440.5
15	1050259.25	1163379.5	1042262.75	1113018.25	1219832.0	1117750.35
20	1041794.5	1044239.0	1044101.25	1056809.5	1006992.75	1038787.4
25	1034715.75	989124.75	1024693.0	977038.0	1064147.25	1017943.75
30	1065543.0	1068556.25	1091145.5	1049350.75	1094714.0	1073861.9
35	1120940.0	1098153.0	1110831.5	1139602.25	1128363.25	1119578.0
40	1331763.0	1267009.75	1258496.5	1302656.0	1288635.75	1289712.2
45	1521172.75	1386702.75	1462408.25	1414832.5	1433865.25	1443796.3
50	1606602.0	1577183.75	1600372.0	1528276.0	1523267.5	1567140.25

When number of passengers is less than the number of cars, then average tour time is similar(does not change), as a car is almost always available for a passenger to ride. However, when number of passengers is more than the number of cars, then a passenger will have to wait until a car is available, and this time increases the more the difference is between the number of cars and number of passengers.

Graph for average ride time vs number of cars



Cars	T_1	T_2	T_3	T_4	T_5	T_{avg}
5	2794613.4	2625635.4	2837873.2	2487146.6	2704377.6	2689929.24
10	1326925.4	1174888.8	1400616.4	1466848.3	1273012.6	1328458.3
15	894728.27	918376.0	830432.33	888365.53	914152.93	889211.01
20	652122.85	700253.8	661392.25	707459.7	703522.65	684950.25
25	528618.68	514607.36	565992.76	568878.92	552459.2	546111.38

Here, we find there is an inverse relationship between average ride time and number of cars, which is logical as the total ride time of the passengers is constant in the tests, and this time is being shared between the cars.