Operating Systems 2

Ishaan Jain CO21BTECH11006

Programming Assignment 4

Program Design

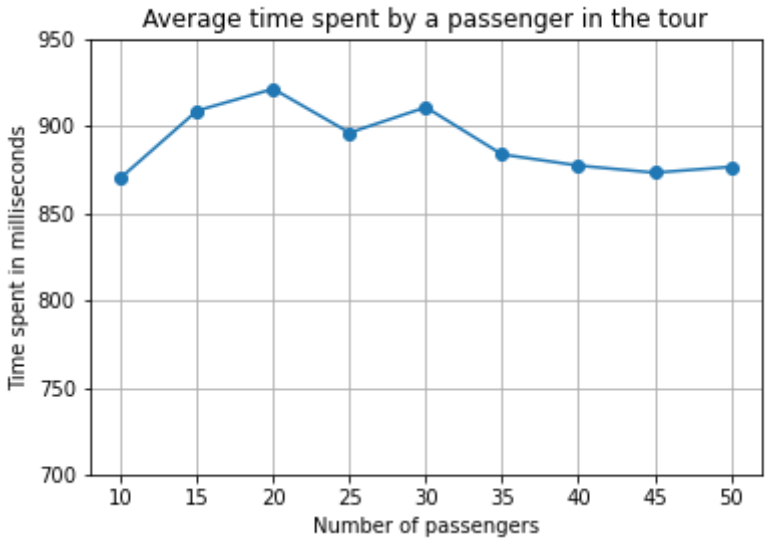
- The program take input from the file "inp_params.txt" and reads the following:
  - P, the number of passengers.
  - C, the number of cars.
  - λ_P, the parameter for the exponential wait between successive ride requests made by the passenger
  - λ_C, the parameter for the exponential wait between successive ride requests accepted by the cars
  - K, the number of rides each passenger takes.
- The program then creates the required number of passenger and car threads and calls the functions car_thread() and passenger_thread()
- The parameters are initialized globally.
- Semaphores and Mutexes used:
  - *log_mutex:* used to safely print the log in the output file.
  - *que_mutex*: used to lock the queue before pushing/popping elements.
  - *request*: mutex used to regulate the requests made by the passengers.
  - *passenger_done*: used to signal if a passenger is done or not.
  - *car_available*: used to regulate the number of available cars.
  - car_mutex: it is an array of semaphores of size P. It helps communicate between car and passenger threads.
- *passenger_thread()*
  - To simulate the waiting of the passenger before requesting any ride, the function *"sleeps" the threads for some random time which was given by* using C++ library *std::default_random_generator* and *std::exponential_distribution by using the parameter* λ_P.
  - We keep logging the messages by *log_message()* function. That function uses *log_mutex* to protect the message from being overwritten.
  - It then generates a for loop to simulate the *k* rides.
  - It then gets the system time for logging a message for a ride request.
  - It then goes to *wait()* for semaphore, which is greater than 0, so it passes it.
  - The ID of the passenger is pushed into a queue, the *request* semaphore is signaled and the *car_mutex[id]* is put to wait, these 2 steps put the control from *passenger_thread* to *car_thread.*
  - After returning to o *passenger_thread*, the *car_available* semaphore is signaled to make the car available again when the ride ends.
  - There is a delay in two successive rides for a passenger, which is simulated by the same random number generator as above and the thread is put to sleep.
  - Outside the while loop, the *wait()* is called upon *passenger_done* indicating that the thread has completed *k* rides
- *car_thread()*
  - It gets into a while loop which terminates when the total number of rides is $P \times k$. It is counted by an *atomic_int* after every ride.

- This function initially is blocked by *request* semaphore which is signaled by the *passenger_thread.*
- The passengers who requested a ride were in a queue, those are allotted cars on FCFS.
- *sleep()* is used to simulate the ride delay. The delay time is also taken from the random number generator in the exponential distribution, but this time the parameter is λ_C.
- The finish of the ride is logged into the output file.
- The *car_mutex[the riding passenger]* signaled, indicating that the ride was over.
- The *rides_completed* variable is incremented as a ride is completed.
- The *passenger_done* is signaled.

Structure of Program

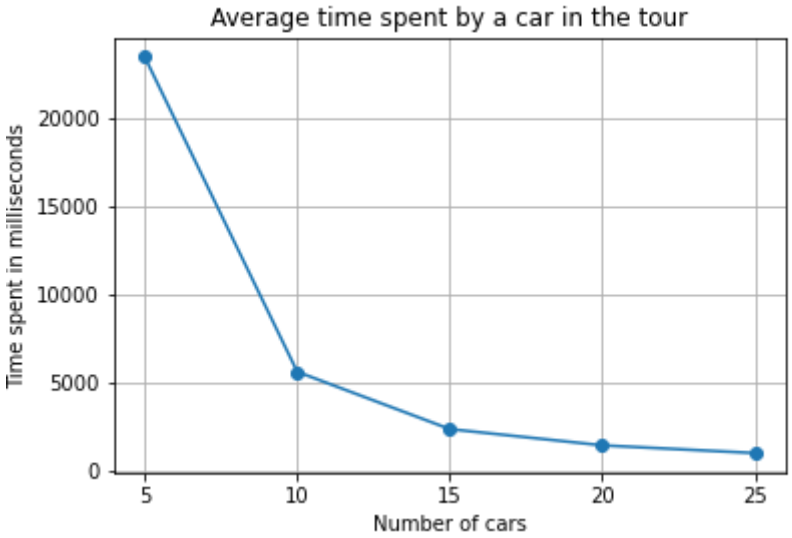
- We have a class
  - *combination // to store the rider and car combo for logging*
    - *rider*
    - *car*
- Queue *pass_id*
  - This acts like a waiting queue for the passengers.
- Queue *riding*
  - This keeps a track of the combination of the rider and car having a tour.
- Global Variables
  - Int *P, C, k* taken from the input file
  - double λ_P, λ_C also taken from input file.
  - *atomic_int rides_completed:* to keep track of the number of rides.
  - Semaphores and mutexes as defined above.
  - *ofstream log_file:* to generate the output file and print data in it.

Comparison between Number of Passengers and Average Time to complete the tour



This data was taken with the value of $\lambda_P = 10,\ \lambda_c = 10,\ k = 5\ and\ C = 25$

Comparison between Number of Cars and Average Time to complete the tour



This data was taken with the value o $\lambda_P = 10$, $\lambda_c = 10$, $k = 3$ and $P = 50$

Note:

- The processes have been generated at random using *std::default_random_generator* and *std::exponential_distribution*
- There can be situations where there might be anomalies while running multiple threads and running them multiple times.
- The results also depend on the values of λ_P, λ_C
- All the outputs generated in the stats file were copied and pasted into a excel to generate a graph and is generated by NumPy and matplotlib
- Due to some starvation or deadlock, my *car_threads()* were not terminating. Though the passenger threads were terminated successfully, that output was used to generate the graph.