# Operating Systems 2

## Ishaan Jain CO21BTECH11006

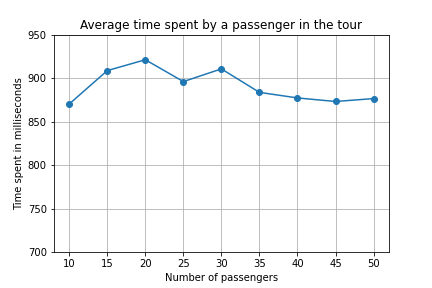
Programming Assignment 4

**Program Design**

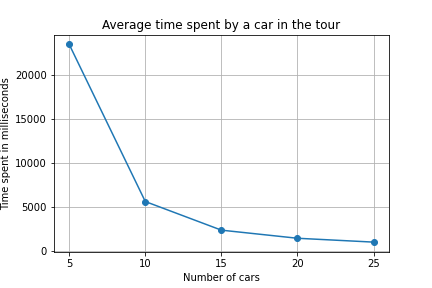
* The program take input from the file “inp\_params.txt” and reads the following:
  + P, the number of passengers.
  + C, the number of cars.
  + , the parameter for the exponential wait between successive ride requests made by the passenger
  + , the parameter for the exponential wait between successive 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 .*
  + 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 . 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 .
  + 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 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

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

This data was taken with the value o

**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
* 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.