CO21BTECH11002
Aayush Kumar
OS2 Programming Assignment 4

Global Variables:

pas_ids:- It stores the id of the passenger riding in i'th car. Say, car_id = 5, pas_ids[car_id] = 7, this means that the passenger with id 7 is riding in the car with id 5.

avl_cars:- It stores the ids of cars that are available to take passenger requests in the form of a queue. Each time a passenger requests a ride, the car that is in the front of avl_cars vector is assigned to the passenger. Each time a car completes its ride, it pushes its own id at the back of the vector.

The car_func( ) function:

This function simulates car rides. Each car thread has an infinite while loop which will make sure each car is active indefinitely until all the passenger requests are exhausted. Each car thread waits until the state of the car with the same id is changed from waiting (i.e. 0) to riding (i.e. 1) by a passenger thread. Everytime the state of the car is changed to riding, the thread checks if total no. of rides are already completed and returns if rides are completed.

```
if(total == p*k) {
    return;
}
```

Else, it notes the current time and prints it in the log file. Then it rides for some random amount of time. The ride is simulated using the usleep() function.

```
double t = exp1(random_number_generator)*1000000; // time in microseconds
usleep(t);
```

After the ride is completed, it prints the info in log files and signals the passenger that the ride is completed.

```
// signal passenger that ride is over
sem_post(&pas_state[pas_ids[car_id]]);
```

Also, after each ride completion, the process notes the time and updates the total ride time accordingly.

The car then waits for some random amount of time before taking another ride. Once the wait is complete, it pushes its id in the avl_cars vector and increases the count of available cars by signaling the empty_mutex semaphore.

```
// add car to queue of available cars
sem_wait(&avl_cars_mutex);
avl_cars.push_back(car_id);
// increment no. of cars waiting
sem_post(&empty_mutex);
sem_post(&avl_cars_mutex);
```

The pas_func( ) function:

This function simulates the requests made by the passengers. The function notes the current time and prints it into the log file. Then, each thread waits for some time before starting to make requests, corresponding to the wandering around of the passengers when they enter the museum. This is simulated using the uslee() function.

```
// wait for some random time before requesting a ride
exponential_distribution<double> exp2(1.0/lam_p);

double t = exp2(random_number_generator)*1000000; // time in microseconds
usleep(t);
```

Each passenger thread has a for loop that runs for k times corresponding to k requests made by each passenger.

Everytime a request is made, the function notes the current time and prints it in the log file. Then, the passengers wait if there are no available cars.

```
// wait for a car to be available
sem_wait(&empty_mutex);
```

Once a car becomes available, the passenger takes from the avl_cars queue and erases it from the queue. A message that Car ‘i’ has accepted passenger ‘j’ is printed in the log file.

```
// get the first available car
sem_wait(&avl_cars_mutex);
int car_id = avl_cars.front();

// print car accept request time in log file
fprintf(f_out,"Car %d accepts passenger %d's request\n", car_id, pas_id);

// remove car from queue of available cars
avl_cars.erase(avl_cars.begin());

// signal avl_cars_mutex so that other threads could modify avl_cars queue
sem_post(&avl_cars_mutex);
```

The passenger id riding the car is stored in the pas_ids vector at the index corresponding to the car’s id.

```
// store id of passenger travelling in car car_id
pas_ids[car_id] = pas_id;
```

The time is noted and printed in the log file. The passenger then signals the car to start the ride.

```
// signal car to start the ride
sem_post(&car_state[car_id]);
```

Now the passenger starts riding and waits until it receives a signal from the car thread about completion of the ride. Time is again noted and printed in the log file.

```
// wait for ride to end
sem_wait(&pas_state[pas_id]);
```

After each ride is completed, the value of the ‘total’ variable is incremented. The passenger then waits for some random amount of time before making another request.

After all the requests have been completed, i.e. the loop has run for k times, the passenger exits the museum. The exit time is noted and printed in the log file. The value of the ‘pas_total’ variable is updated accordingly.

```
// print exit time in log file
fprintf(f_out,"Passenger %d exit the museum at %d:%d:%d\n", pas_id, exit_now->tm_hour, exit_now->tm_min, exit_now->tm_sec);

// update pas_total
sem_wait(&pas_mutex);
pas_total += ExitTime - EnterTime;
sem_post(&pas_mutex);
```

Once all the rides have been completed, the states of all the cars are set to 1, as there can be some cars that may wait indefinitely but won’t get any requests from passengers since the rides are completed. When the car threads proceed further, they check that the total no. of rides has already been completed and hence, they return.

```
// check if all requests have been completed
// once all the rides are over, the car_state can be changed by any thread in any order as that does not affect the output
// hence, the update does not need to be mutually exclusive
if(total == p*k) {
    for(int i=0; i<n; i++) {
        sem_post(&car_state[i+1]);
    }
}
```

The main( ) function:

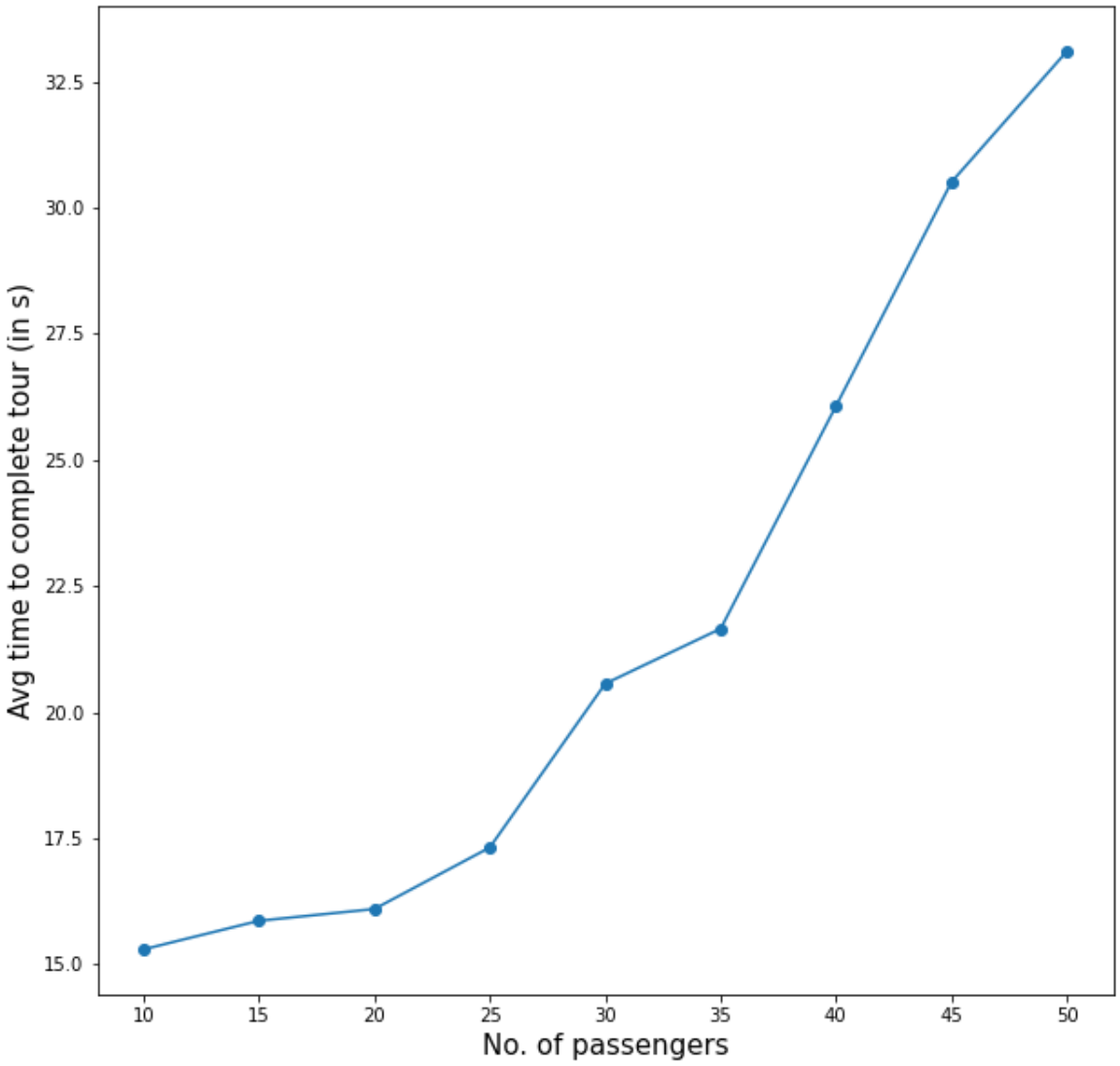
We start by reading the value of n - no. of cars, p - no. of passengers, lam_c - the parameter for exponential wait time of cars, lam_p - the parameter for exponential wait time of passengers and k - no. of requests made by each passenger. Then the vectors and the semaphores are initialized to required values. Then, we create n car threads and p passenger threads and run them. Then we wait for the threads to finish execution and on them. Finally, average time for cars and passengers are calculated and printed.

Results:

Taking lam_c = 1 and lam_p = 2, we get the following results:

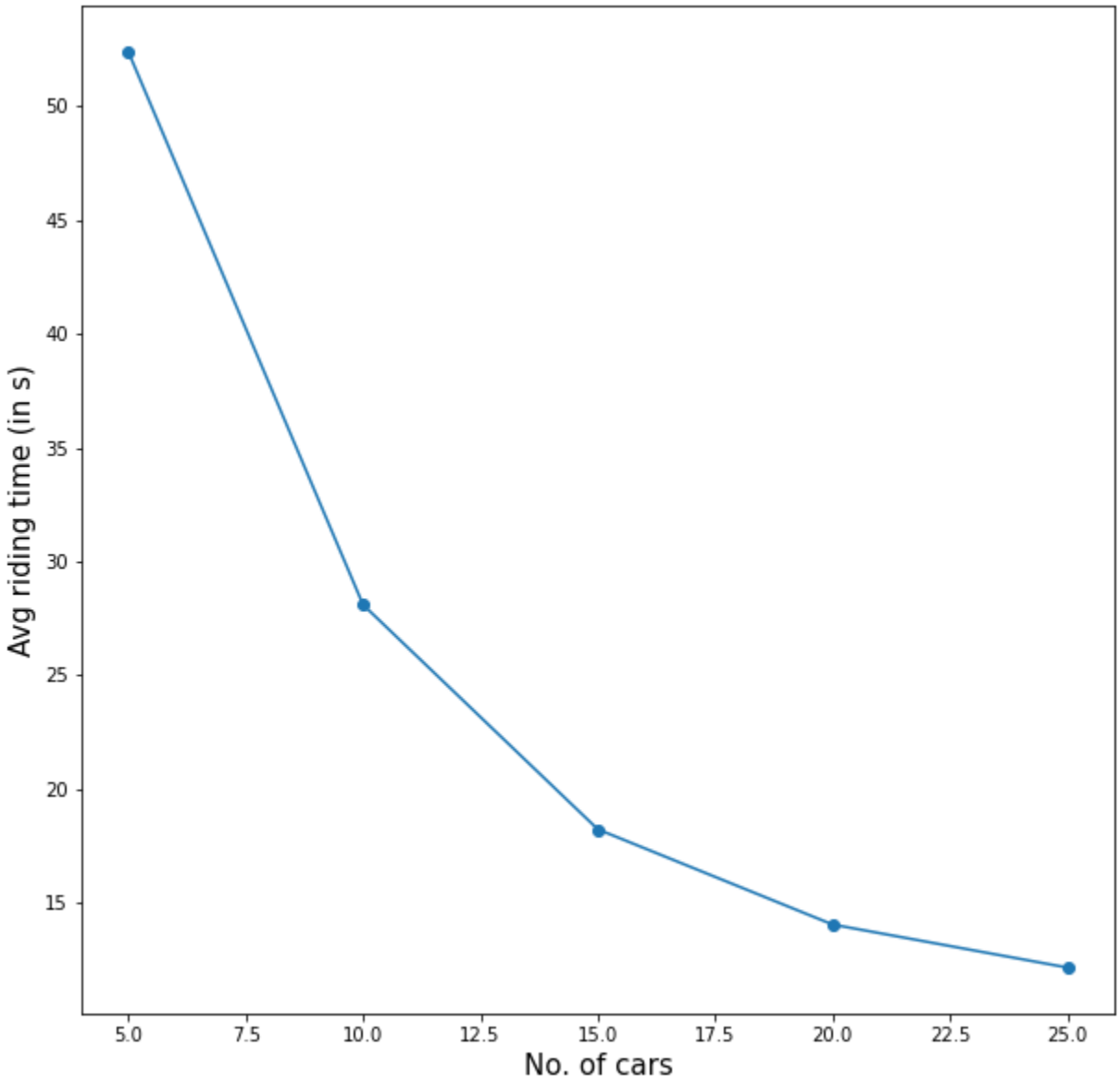
Average time to complete tour (in s) vs no. of passengers (keeping number of cars = 25, no. of requests = 5):

Passengers	10	15	20	25	30	35	40	45	50
Avg Time	15.3	15.867	16.1	17.32	20.567	21.657	26.075	30.511	33.104



Average riding time (in s) vs no. of cars (keeping no. of passengers = 50, no. of requests = 3) :

Cars	5	10	15	20	25
Avg Time	52.4	28.1	18.233	14.05	12.16



Conclusion:

1. When the no. of cars is fixed, on increasing the no. of passengers, the average time to complete the tour increases as each passenger will have to wait for more time for other passengers to complete the rides and free the cars.
2. In the initial stages, on increasing the no. of passengers, the increase is not much since each passenger wanders around for some time before making another request. During this time, other passengers free some of the cars. Hence, other passengers do not have to wait for a long time before boarding the cars. But when the no. of passengers is high, after wandering around for some time, passengers will have to wait additional time, thus affecting the average waiting time.
3. When no. of passengers is fixed, on increasing the no. of cars, the average riding time of the cars decreases as each car will have to complete less no. of requests.
4. The decrease in average riding time is significant in initial stages but saturates near the end. This is because when the cars are high in no. the cars will have to wait longer for passengers to make requests.