The Jurassic Park Problem

Aaryan, CO21BTECH11001

Note: For the sake of saving time and also simulating that threads are performing some time consuming tasks, I have set the values of λ_P and λ_C as $\lambda_P = 1.0$, $\lambda_C = 2.0$, so that we can get output within a few minutes.

Working of Code:

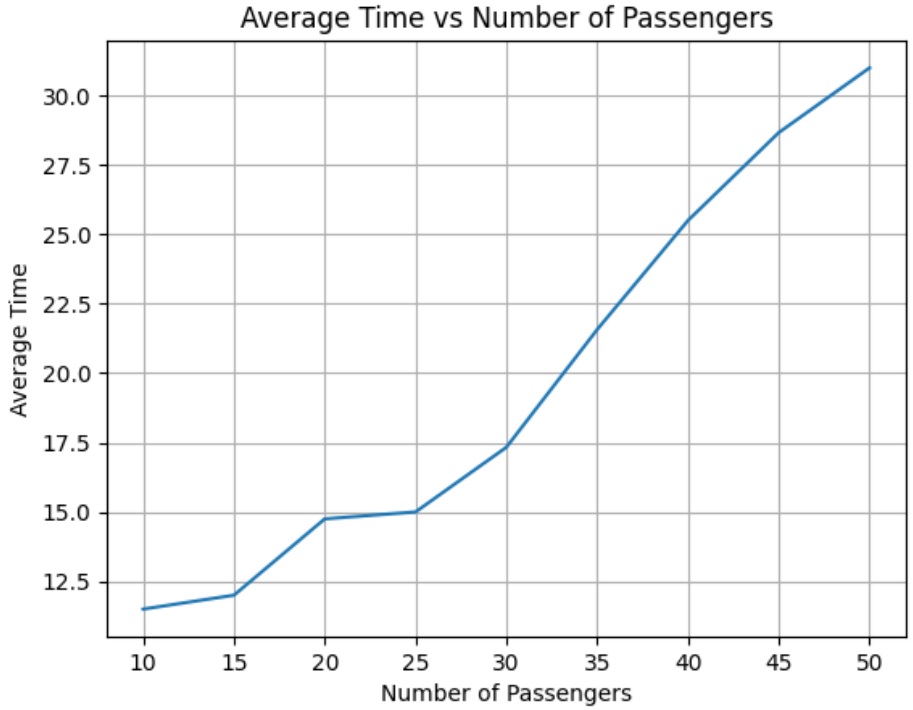
1. Let
    a. The number of passengers = `n`.
    b. The number of cars = `m`.
    c. Number of ride request by each passenger = `k`
2. A vector `car_buffer` stores the id of all cars which are available. It is initialized by adding all numbers in `[0,m)`.
   `_mutex` is a semaphore to ensure mutual exclusion during access to `car_buffer`.
   `_empty` is a semaphore to ensure that a passenger has to wait if no car is available.
3. `car_sem` is an array of semaphores corresponding to each car. Every element of the array is initialized to `0`.
4. `passenger_sem` is an array of semaphores corresponding to each passenger. Every element of the array is initialized to `0`.
5. Then, `m` threads representing each car and `n` threads representing each passenger are made.
6. Argument of thread functions of both car as well as passenger is their `id` in 0th index.
7. When the passenger enters it's thread function, it sleeps for a while (roam around), then make `k` requests as follows:
    a. Waits for a car to be available (using `_empty`).
    b. Waits for `car_buffer` to be available (using `_mutex`).
    c. Signals the car thread whose id is the last element of `car_buffer` (using `car_sem`) and it removes the last element from `car_buffer`.

    d. Wait for the ride to finish (using `passenger_sem`).
    e. If the request is not the last request, it sleeps before making the next request.
    f. If the request is the last request, the passenger exits the museum (function returns).
8. When the car enters it's thread function, it executes as follows in an infinite loop (which breaks when all request are fulfilled) :
    a. Wait until a thread signals (using `car_sem`)
    b. Sleep for random time (riding time).
    c. When, ride is finished, it signals the passenger who is riding (using `passenger_sem`).
    d. If all requests are not fulfilled, sleep for random time before accepting the next ride request.
    Then, it will add its id at the back of car_buffer by grabbing `_mutex`.
    e. If all requests are fulfilled, it comes out of the loop and the function returns.
9. An array `passengerInCar` of size `m` is made to keep track of the `id` of the passenger who is riding in `ith` car.
10. To keep track of the number of requests which are completed, a variable `totalRequestsCompleted` is used. A semaphore `_totalRequest` is used to increment `totalRequestsCompleted`.
11. While incrementing `totalRequestsCompleted`, if a passenger thread finds that all requests are completed (i.e., `totalRequestsCompleted == n*k`), it will signal all car threads so that the car threads can return from their function.
12. To find the average time taken by a passenger, a variable `totalPassengerTime` is used. It is incremented every time a passenger completes a ride.
13. To find the average time taken by a car, a variable `totalCarTime` is used. It is incremented by each car thread when there are no requests left.

Results:

1. **Average time taken by passengers vs number of passengers:**
When the number of cars and number of requests by each passenger is kept constant, the average time taken by passengers increases. Reason

for the same is that when the number of passengers increases, `car_buffer` becomes empty more frequently. Therefore, passenger threads have to wait longer.
Here is the plot for the same when `m` and `k` are fixed to `25` and `5` respectively.



2. **Average time taken by cars vs number of cars:**
   When the number of passengers and number of requests by each passenger is kept constant, the average time taken by car decreases. Reason for the same is that the request of more passengers can be fulfilled at the same time, therefore total rides to be given by one car decreases and function returns in lesser time.
   Here is the plot for the same when `n` and `k` are fixed to `50` and `3`

respectively.