Programming_Assign_4 - Jurassic Park Problem
Es21btech11003

Code Implementation and Low level Design of Code

Global parameters

1. **FILE *thread_fp** - *Pointer to output file*
2. **Input parameter** - *int P, C, k; double lambdaP, lambdaC;*
3. **int Day_end = 0** - *Used for tracking termination of passenger threads*
4. **The semaphores** :

    sem_t mute - *used as a mutex lock - for critical section parts of the code*

    sem_t passenger_sem - *for tracking passenger requests*

    sem_t car_sem - *keeps track of no. of cars active*

    sem_t *in_car - *tracks passenger threads that are in cars*

5. **queue<int> passenger_queue** - ride request queue

Main Implementation

1. File input from file named - ***"inp-params.txt"*** , takes input parameters as space separated values .
2. **Initializing Semaphores** -

```
// Initialize the semaphores
sem_init(&mute, 0, 1);
sem_init(&passenger_sem, 0, 0);
sem_init(&car_sem, 0, C);
in_car = (sem_t *)malloc(sizeof(sem_t) * P);
for (int i = 0; i < P; i++) {
  sem_init(&in_car[i], 0, 0);
}
```

3. **Thread creation** - using thread library , P passenger threads , C car threads are created .
4. Waiting and joining of threads - at first waits for passenger threads to terminate , then change **Day_end variable** to 1 leading to termination of car threads.
5. Destroying Semaphores

Passenger thread function Implementation

1. Each passenger thread enter the museum - the time is printed in outfile
2. Loop for simulating cycles of passengers
3. Initial wait_time to simulate passenger wondering in the museum
4. Passenger request for a ride - time is printed in the outfile
5. Request is pushed to the passenger_queue , this push is made mutually exclusive using mute semaphore.

```
   sem_wait(&mute);
   // Adding the Passenger request to queue
   passenger_queue.push(id);
   sem_post(&mute);
```

6. car_sem semaphore check if a car is available or not , allows only C no. of passenger thread to execute further
7. Passenger sem semaphore signals car threads about the passenger request.

```
//acquiring a car if available , else waiting
   sem_wait(&car_sem);
   //Signaling car thread about passenger request
   sem_post(&passenger_sem);
```

8. Ride starts - time is printed in the outfile
9. In_car sem waits for the passenger to complete the tour before further execution of passenger thread
10. Ride ends - time is printed in the out file
11. car_sem frees the acquired car

```
// freeing the acquired car
   sem_post(&car_sem);
```

Car thread Implementation

1. Inside a while(true) loop , each car thread waits for passenger requests .
2. Using passenger_sem semaphore , each thread waits for a passenger request to be invoked from passenger threads .
3. Now each Car thread grants the first request that is in the passenger queue . access to passenger queue is made mutually exclusive using mute sem

```
sem_wait(&mute);
   // handling Passenger request in front of Queue
   if ((passenger_queue.empty() || passenger_queue.front() == 0) && Day_end == 0) {
       sem_post(&mute);
       continue;
   }
   int passenger_id = passenger_queue.front();
   passenger_queue.pop();
   fprintf(thread_fp, "Car %d accepts passenger %d\n", id, passenger_id);
   sem_post(&mute);
```

4. Now each thread waits to simulate a ride time using wait_time(lambdaC) .
5. Car thread finishes the ride - printed in the out file
6. in_car sem signals waiting passenger thread to continue execution.

Output analysis

Let us take an example with input parameters as :
P = 10 // no. of passengers
C = 3 // no. of cars
lambdaP = 1
lambdaC = 2
K = 3 // No of cycles

First 30 lines of the output are as follows :

Passenger 2 enter the Museum at 0.000152 s // at first each thread enter the museum
Passenger 1 enter the Museum at 0.000181 s
Passenger 3 enter the Museum at 0.000190 s
… … …
Passenger 9 request a ride at 0.008287 s // First ride request - Passenger thread 9
passenger 9 starts riding at 0.008303 s // request accepted by Car 1
Car 1 accepts passenger 9
Passenger 7 request a ride at 0.035681 s
passenger 7 starts riding at 0.035700 s
Car 2 accepts passenger 7
Passenger 10 request a ride at 0.069805 s // Passenger 10 requests after 7
passenger 10 starts riding at 0.069843 s // Starts it's ride
Car 3 accepts passenger 10 // all cars are busy now
Passenger 1 request a ride at 0.141361 s // Passenger 1 next in queue after 10
Passenger 3 request a ride at 0.247462 s // then passenger 3
Passenger 2 request a ride at 0.614049 s
Passenger 6 request a ride at 0.733200 s
Passenger 8 request a ride at 0.754885 s
Passenger 4 request a ride at 1.136264 s
Car 3 finishes passenger 10's tour
passenger 10 finishes riding at 1.568401 s // Passenger 10 finishes
passenger 1 starts riding at 1.568468 s // passenger 1 frees Car 3
Car 3 accepts passenger 1 // Car 3 accepts next request in queue
Car 1 finishes passenger 9's tour // Passenger 9 finishes it's first tour
passenger 9 finishes riding at 2.330055 s
passenger 3 starts riding at 2.330090 s
Car 1 accepts passenger 3
Car 1 finishes passenger 3's tour
passenger 3 finishes riding at 2.427466 s
passenger 2 starts riding at 2.427502 s

As We can see here, at first each thread enters the museum. A passenger thread makes a car request , if a car is available , request is accepted, else the request is added to the queue and as soon as a car is available the request in front of the queue is accepted .

In the shown example : The car request are in order -

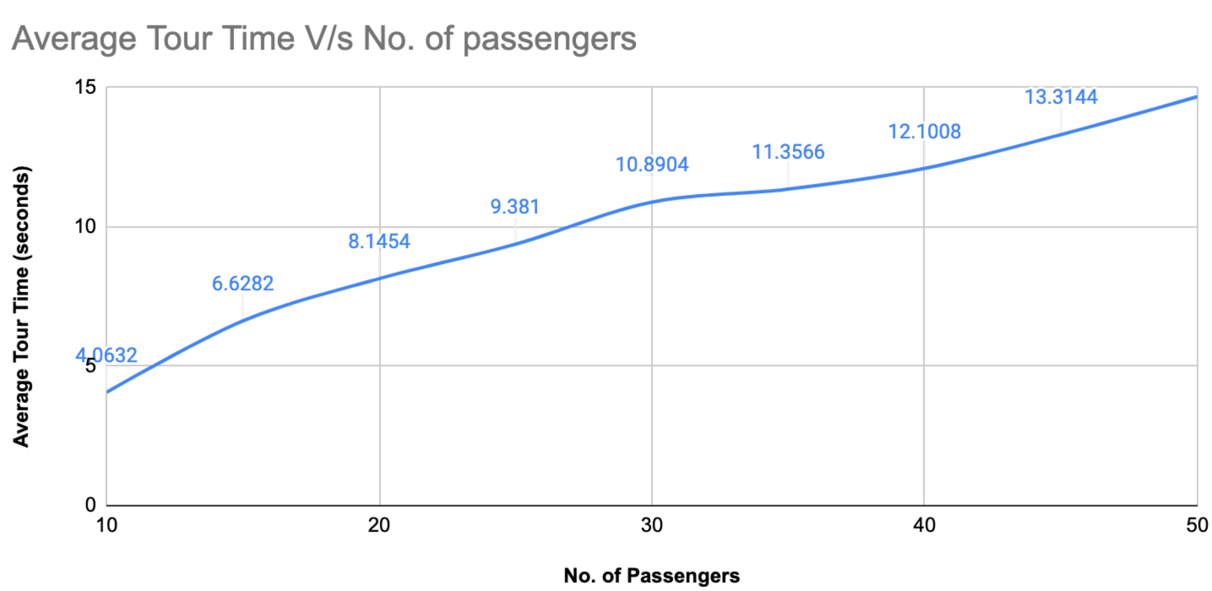
1. Passenger 9 - accepted by Car 1
2. Passenger 7 - accepted by Car 2
3. Passenger 10 - accepted by Car 3
4. Passenger 1 - waites for car to get available
5. Passenger 10 finishes its tour - Frees Car 3
6. Car 3 accept Passenger 1’s request as it was in front of the queue
7. After passenger 1 , Passenger 3 was in the queue
8. Passenger 9 finishes its tour - Frees Car 1
9. Car 1 accepts Passenger 3’s request
10. And henceforth …

Plots

Average Tour Time V/s No. of passengers

No. of Passengers	Average tour time(seconds)					
	Iteration 1	iteration 2	iteration 3	iteration 4	iteration 5	Average time
10	4.856	4.204	4.649	3.496	3.111	4.0632
15	6.906	6.112	6.914	6.426	6.783	6.6282
20	8.314	8.684	8.136	7.943	7.65	8.1454
25	9.332	9.596	9.311	9.673	8.993	9.381
30	11.15	10.506	11.487	11.296	10.013	10.8904
35	11.873	11.324	11.214	10.372	12	11.3566
40	12.925	11.354	11.856	12.33	12.039	12.1008
45	13.765	14.052	13.603	12.424	12.728	13.3144
50	15.605	14.954	14.56	15.214	13.047	14.676

Here as we can see , As the No. of passengers increases the time taken for each passenger to complete all its cycle also increases .
Reasoning - as the no. of passengers increases, availability of cars decreases hence time taken increases , Car threads are more often already occupied when requested so each passenger is required to wait more time.



Average Tour Time V/s No. of cars

No. of Cars	Average tour time(seconds)					
	Iteration 1	iteration 2	iteration 3	iteration 4	iteration 5	Average time
5	35.542	34.329	32.395	36.195	34.694	34.631
10	29.586	29.485	27.194	30.684	30.382	29.4662
15	23.39	24.295	22.596	23.052	25.596	23.7858
20	15.395	14.863	14.695	16.984	13.541	15.0956
25	7.395	7.204	9.321	8.205	8.993	8.2236

Here as we can see , As the no. of cars increases the time taken for each passenger to complete all its cycle also decreases .
Reasoning - as the no. of cars increase, the availability of cars also increases hence time taken per passenger decreases , Car threads are more often already available when requested so each passenger and less wait time is required .