Operating Systems2(CS3523)
Programming Assignment 4: The Jurassic Park Problem

Govinda Rohith Y
CS21BTECH11062

Low level design:

First the input parameters is read from "inp-params.txt" file and all those parameters are declared gloablly. Explanation of side functions which support various steps in execution are given below and after explanation of all functions the main design is explained below .

1. `main()` function:

   The two random number generator `distribution1` and `distribution2` are intiliased by given means λ_P and λ_C which are later used to generate exponentially distributed random numbers with given means. Now `class Dat` is created with 4 fields for storing request time `request_time`,entry time `entry_time`,exit time `exit_time` and car no `int carno`. For each thread and for each iteration. So a 2d array `Dat **req_dat;` is created globally of size P×K.To store entry time and entry time of each passenger into the museum two 1D arrays each of size P are created `string *entry_mu,*exit_mu;`. A boolean array `bool *is_avail_car;` is created to store status of each car which is used later. Now the function `museum_car()` is called which is solution to given problem. After all threads finish their execution the function metric_p1 and metric_p2 is called to print data required for graph plotting and now the allocated memory is freed. The car threads are killed forcefully to prevent infinite looping `pthread_kill(pid[i], SIGKILL);` this part is explained below in Main design.

2. `exponential_distribution<double> var_rand(int meaner)`

   At high level this function returns object which generates exponential distributed numbers when called whose mean is given as argument. To generate this first `random_device rd` is intiliased globally which is used to produce non deterministic random numbers. This is used to generate seed value for `mt19937 gen(rd())` which is very efficient pseudo-random number generator , by Mersenne twister algorithm , this is used later to generate random numbers. In the function `var_rand(int meaner)` `exponential_distribution<double>` object is intiliased according to given mean and this is returned. Two objects are intiliased `exponential_distribution<double> distribution1,distribution2` to generate two exponentially distributed random number when called each time (`distribution1(gen)`) whose means are $\lambda_P$andλ_C respectively. Note the gen is seed for mt19937 object which is intiliased before.

3. `void file_printer()`

   This function is to print output file as per given requirements. All the request,entry,exit times and car numbers stored in req_dat are read and printed into "output.txt" as per given instructions.

4. `std::string get_time()`

   This function returns current system time in HH:MM:SS.mmm (mmm is milliseconds) format as string. `using clock = system_clock` creates an alias for system_clock. clock::now gives the system time and assign to variable current_time_point and in next two steps it is converted into duration from epoch. Then the next expression (`duration_cast<milliseconds> (current_time_since_epoch).count() % 1000`) calculates the number of milliseconds that have elapsed since the current second began. Then the time is converted into string format HH.MM.SS.mmm (where m is milliseconds) and returned.

5. `void sleeper(double req_time)`

   This function is used to sleep the current thread for time in milliseconds which is given as argument `double req_time`. This is mainly implemented by using nanosleep() function. The parameter `req_time` is divided by 1000 to get time in milliseconds. `struct timespec tim` is intiliased with tim.tv_sec=(int)req_time/1 which is time in seconds and tim.tv_nsec=(req_time-(int)req_time/1)*1000000000 which is time in nanoseconds example if 2004.678 milliseconds is the parameter value then tim.tv_sec= 2 seconds and tim.tv_nsec=4678000 nanoseconds. Then nanoleep function is called which make sleep the current thread according to specified time.

6. `void metric_p1()`

   This function is used to calculate the average time taken by the passengers to complete their tour, which is helpful in graph plotting. A for loop is made for p passenger threads Inside the loop, the function extracts the entry time and exit time of the passenger i from the arrays `entry_mu` and `exit_mu`, calculates the time time difference between them in milliseconds. This gives the time spent in museum by a passenger and the average of all passenger is calculated accordingly.

7. `void metric_p2()`

   This function is used to calculate the average time taken by the cars to complete its tour, which is helpful in graph plotting. The function uses nested loops to iterate over each ride taken by each passenger.The entry and exit times of a car are stored as strings in a two-dimensional array 'req_dat', which is indexed using 'i' and 'j'.The entry and exit times are parsed to extract hours, minutes, seconds, and milliseconds using the substr() function and stoi() function to convert the extracted strings to integers.The time taken by a car is calculated by subtracting the entry time from the exit time and converting the result to milliseconds.The calculated time is added to the 'store' array at the index corresponding to the car number.The average time taken by a car is calculated by dividing the total time taken by all cars ('differ') by the total number of cars ('c').The result is printed to the console.

Main Design:

The `main` function calls the function `void museum_car()`. This function creates and intiliases "P+C" binary semaphores in form of arrays `sem_t *sem_ex_p,*sem_ex_c`. These will be useful later. Each of these semaphores is dedicated to each thread. Two semaphores `sem_c and sem_bin` are created and intiliased. All the semaphores are created globally. The semaphore `sem_c` is intiliased to number of cars "C" which is useful for passenger thread to wait if there are no available cars. The binary semaphore `sem_bin` is used particulary to protect boolean array `bool * is_avail_car`. Then after semaphore creation "P" passenger threads and "C" car threads are created. Each passenger thread calls the function `each_pas(arg)` with thread numbers(passenger

number) as its argument. Each car thread calls the function `each_car(arg)` with thread number (car number) as its argument.

Each passenger thread first gets its passenger number (thread number) from argument. Then `sem_wait(sem_ex_p[t_no])` is done to represent the passenger's entry into the museum. Note that this step is useful because if this step is not done then we have to call `sem_wait(sem_ex_p[t_no])` two times first time to make semaphore 0 and second time to put that passenger thread to sleep or wait because of that step we can just call `sem_wait(sem_ex_p[t_no])` once to put passenger thread to waiting. Then museum entry time is recorded into array `entry_mu[t_no]` .Next a for loop is made for 'K' iterations for each iteration the passenger thread generates a roaming time temp using a pre-defined probability distribution, `distribution1(gen)`. The function then calls sleeper(temp) to simulate the passenger roaming for temp milliseconds. Now the request time is recorded into `req_dat[i][j].request_time` (for thread i and iteration j).After that, the passenger thread requests a car by waiting for the sem_c semaphore. Once the semaphore is available(indicating a car is available), the passenger thread waits for the sem_bin semaphore to read and modify `is_avail_car` and finds the first available car by checking the `is_avail_car` array. If a car is found, the passenger thread reserves the car by setting the is_avail_car array entry to the passenger's thread number t_no and breaks from the loop. At this point of time the passenger thread got a available car for it's riding and the car number is stored in variable `'j'` and also recorded into `req_dat[i][j].carno` . And the passenger thread records the entry time into data structure `req_dat[i][j].entry_time` . Now the passenger thread signals that car 'j' by using semaphore `sem_post(&sem_ex_c[j]);` and the passenger thread puts itself to waiting for the ride to be over (This riding is done in car thread).

Similiarly to passenger thread, in car thread `sem_wait(&sem_ex_c[t_no]);` is done to represent the car is available. Now a infinite loop (which will be terminated later by `pthread_kill(thread_no)`) is run. At start of this loop each car waits for signal from any passenger `sem_wait(&sem_ex_c[t_no]);`. After receiving signal from passenger the car thread simulates the roaming time by calling the function `sleeper(temp)` and the time temp is generated by `distribution2(gen)`. After roaming the car thread signals the passenger thread `sem_post(&sem_ex_p[is_avail_car[t_no]]);` so that passenger thread can continue and car waits for another passenger signal. After riding is done the passenger thread records the exit time into `req_dat[i][j].exit_time` and sets `is_avail_car[j]=-1; sem_post(&sem_c);` which indicates the releasing of resources.

Graph Analysis

Note for below graphs the parameters λ_P and λ_C are taken as 100 milliseconds.

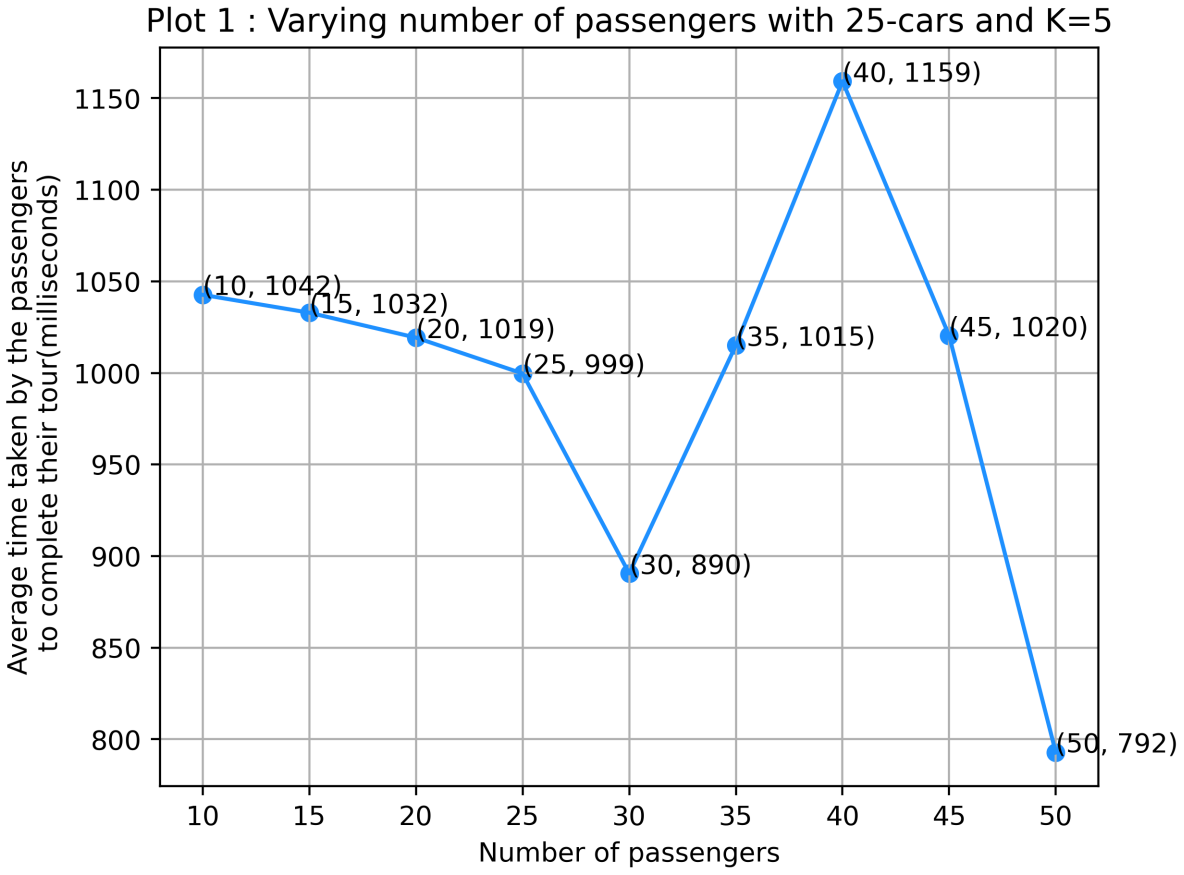

Figure 1: Plot between average time taken by the passengers to complete their tour vs number of passengers,by fixing number of cars=25 and K=5

From the above Figure 1 the overall average time taken by the passengers to complete their tour is almost lies in the range (800-1000) ms. When number of passengers are increased there are no significant change in average time. The average time is decreasing upto 30 passengers and reach the local minimum at 30. This is obvious as the number of cars are 25 and total number of passengers<25 so the average tour time decreases. When passengers count is >30 the average tour time increases because as number of cars are less compared to passengers the passengers have to wait for a car which increases the tour time so the average increases.

Anomalies

One anomaly is that when passenger count >40 we expect the average time to increase but it decreases because of the fact that number of passengers are increased but the waiting time almost remains same to that of when passenger count >30. This is because there are atmost two people each car has to serve (taking average analysis) for each iteration in both cases (passenger count>30 and passenger count>40) so the waiting time almost remains same.

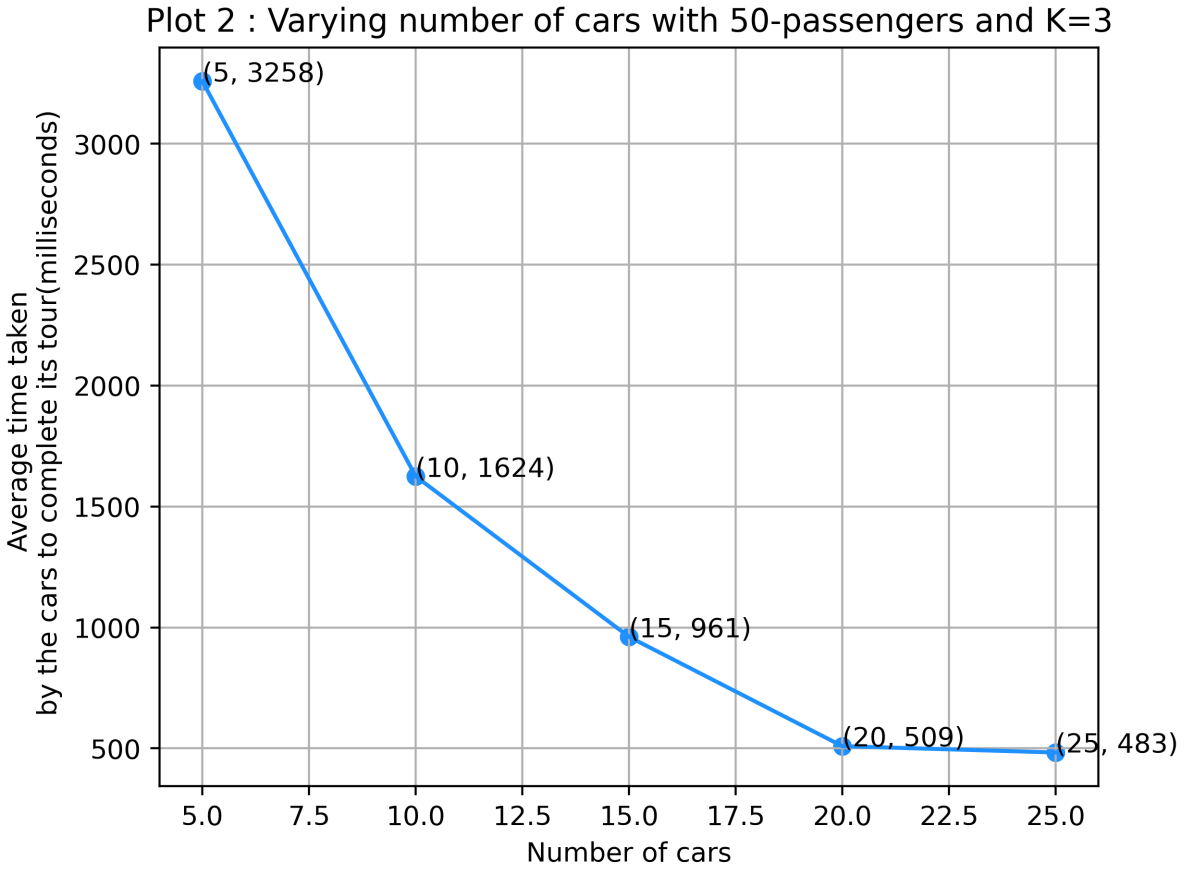

Figure 2: Plot between the average time taken by the cars to complete its tour vs number of cars ,by fixing number of passengers=50 and K=3

From the above Figure 2 the average time taken by the car to complete its tour decrease as number of cars increases. This is obvious because of the fact that, as there are more number of cars available all the cars can complete its tour fastly as the work (picking up passengers) is shared so the average decreases. But as number of cars are increased the time becomes decreasingly constant as because of more number of cars available the work sharing reduces average time such that it becomes a constant. This graph is perfect example for the fact that number of cars should be comparable to passenger for efficient use of work sharing.

Analysis of algorithm:

For a given input P,C,λ_P,λ_C. The space complexity is O(3×P×K+2×P). Time complexity can be calculated by assuming each thread sleeps for approximately λ_P amount of time to simulate museum viewing section and λ_C amount of time for roaming section so for total of P threads and each thread with K iterations takes $<$ O($(\lambda_P + \lambda_K)$×P×k). That "less than" is occuring because of the fact that, the all threads are executing in parellel so the time may decrease.