REPORT

ANALYSIS

Low-Level Design Of The Code

This code simulates a museum ride where passengers make requests to ride on cars. There are p passengers and c cars. Passengers can make k requests and a car can carry only one passenger at a time. Each car takes some specific time to load another passenger, once it has loaded a passenger. Also, a passenger can make ride requests only after a specific time period. The low-level design of the code can be explained as follows:

1. **Libraries**: The code uses several libraries such as <iostream>, <pthread.h>, <ctime>, <chrono>, <atomic>, <random>, <unistd.h>, <cstdlib>, <cstring>, <math.h>, <semaphore.h>, and the standard namespace.
2. **Global Variables** : The code defines several global variables such as p, c, lp, lc, k, check, check1, time_taken_p, time_taken_c, *car_ride, and semaphores *S, *mutex, *filePointer, along with doubles t1, t2, timetaken_ta, timetaken_tb, timetaken_tc, timetaken_td, timetaken_te, time_passengers, time_cars, and bool done. These variables are used to store the data related to the number of passengers, cars, maintenance and ride time, the number of requests, the rides taken, the clock times, and the completion status.
3. **Ride Function**: The function void *ride(void *arg) defines the ride procedure for each car thread. It stores the corresponding number of the car thread as the car's id. Then it generates an exponential time (t2) for the car to take a new passenger request. And then the ride function enters a loop where it checks if the car is busy or not. This is done using a global array allocated on the heap named car_ride, which keeps into account the status of the car, i,e, whether it's idle or not. So it checks if the specific car id is busy or available. If the car is busy, it waits for time t2 and updates its status on the car_ride array as free. The car threads keep checking the status of the car until all the passenger threads join and terminate. It uses the semaphore mutex (binary semaphore) to synchronize the access to the shared variable car_ride.
4. **Routine Function:** The function void *routine(void *arg) defines the procedure for each passenger thread. It stores the corresponding number of the passenger thread as the passenger's id. It generates an exponential random time (t1) for the passenger to make a new ride request. t1 represents the time each passenger takes before making a new request. It then enters a loop, which allows each passenger thread to make k requests. For each request, it waits for the semaphore S to become available, indicating that a car is free. The passenger can only make a ride request when there is an available car. After acquiring the semaphore, the function enters another loop that iterates over the total number of ride requests possible and then uses a rule to check the car's availability correspondingly. The function then waits on the semaphore mutex to synchronize the

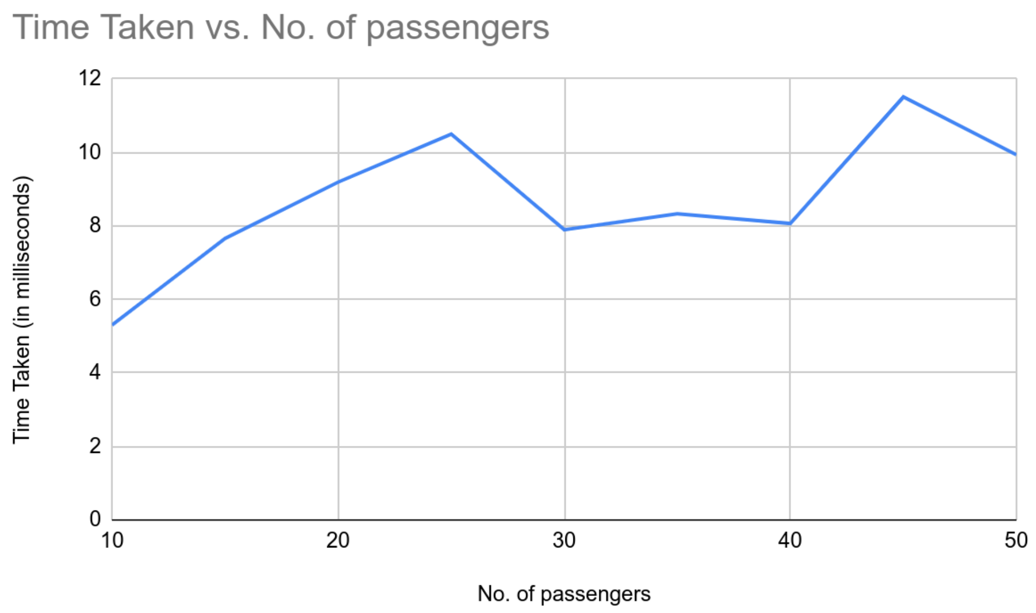
access to the shared variable car_ride. [The value 0 represents an available car, and 1 represents an occupied car.] The function checks if the car selected is available. If the car is available, the function sets the value of car_ride[j] to 1 to indicate that the car is occupied. In order to avoid the inaccuracies of multi-threading programming, we create an additional while loop where the passenger thread waits for the car ride to get over. As soon as the passenger finishes its ride, it waits for time period t1 before making another request and releases the semaphore.

5. **Main function():** First, we take in inputs from an input file, i.e, the values for p, c, lc, lp and k. Then we allocate memory to the semaphores S and mutex. We also dynamically allocate memory to the shared variable car_ride. And then we open the output file to log in the details and the passenger and car threads are created. Then using for loops we join the threads. After the passenger threads join and terminate we update the value of boolean variable done to true. And lastly, we free all the memory that was allocated on the heap and close the output file.

Output

In the routine function, we take note of the times when the passenger thread enters the museum, when the passenger thread makes a ride request, when the car accepts the passengers' which ride request , when the car ride is finished and when the passenger thread exits the museum. And all this information is printed onto an output file. But since the threads work concurrently in nature hence the time log gets jumbled up. So in the output file, we can see the time when the passenger enters the museum, the time when the passenger makes which number of request and all the relevant information. In the end we also note the average time taken by the passenger to complete the trip and the average time the car takes to make the trip.

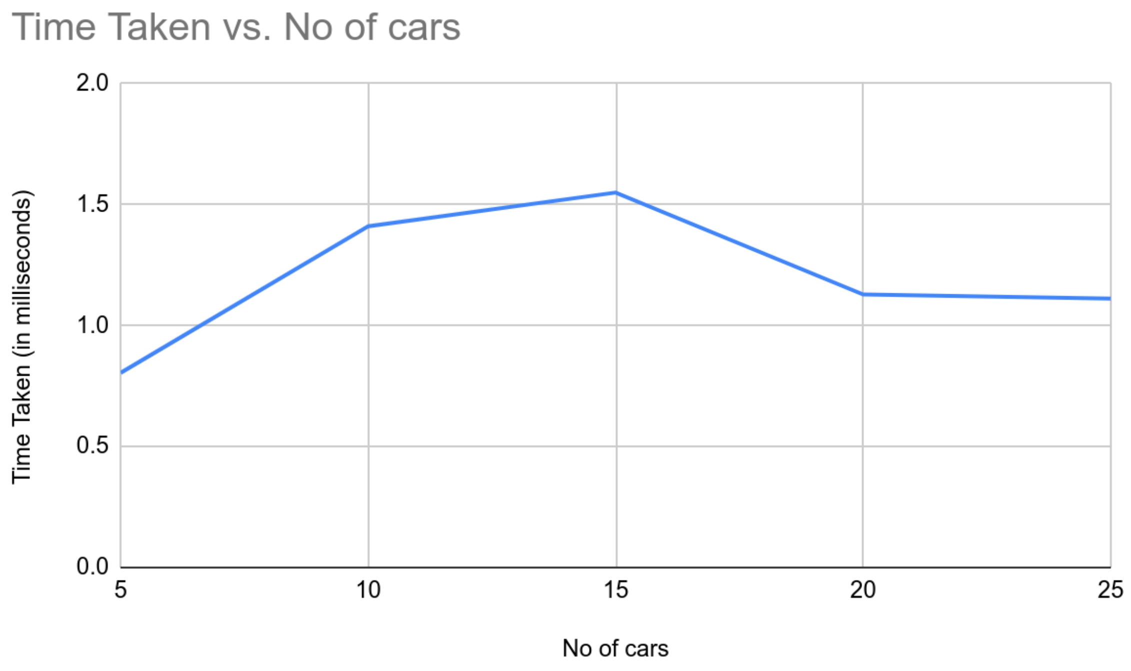
PLOT 1 :



From this graph, we can see that as the number of passenger threads increases while the number of car threads remain constant, the average time taken by the passenger to around the museum increases. This may happen due to resource contention and synchronisation overhead. As the number of passenger threads increases, more threads will compete for access to the shared data structures, such as the car_ride array and the count variable.It causes passenger threads to get blocked while waiting for a car to become available, as a result, the overhead associated with thread synchronization using semaphores increases, which causes an increase in the time taken.

Also, additionally, we observe a fluctuation in the average time taken when the number of passenger threads is between 25 to 45. This can be explained the model, the architecture of the computer core used while running the program and the code written to implement the given problem statement. Additionally, when the number of passenger threads increases, the number of context switches between threads also increases. Context switching is the process of saving the state of one thread and restoring the state of another thread to enable multitasking. This can cause an additional overhead associated with context switching.

PLOT 2:



From this graph, we can see that as the number of car threads increases while the number of passenger threads remains constant, the time taken by the car threads to finish the ride almost remains constant because the time taken by the car to finish the rise does not depend on the number of the car threads. But we observe very small fluctuations in the time taken and this is due to as the number of car threads increases, it results in an increased overhead associated with managing these threads (which is again very small).

IMPORTANT OBSERVATION:

It is important to find the optimal balance between the number of passenger and car threads to maximize the efficiency of the system. This can be achieved through careful monitoring and tuning of the system and may require adjusting other system parameters such as the duration of the safari ride or the number of cars that can be held simultaneously.