King Abdulaziz University

Faculty of Engineering

Electrical and Computer Engineering Department

Operating Systems (EE463)

Team4

Project B:

Design Document

Submission date: 09/05/2022

Member	Name	University ID
1	**Ibrahim Alfaris**	**1855080**
2	**Mohammed Almaafi**	**1845137**
3	**Osamah Badughaish**	**1855398**

Instructor:

Dr. Abdulghani M.Al-Qasimi

Table of Contents:

1. Design Choices:

This section will shade light on the design choices that are made to be justified. The design of this project depends on queueing and serving. Since we have a random number generator generates n number of cars in each iteration, these cars should be enqueued in a FIFO queue. Therefore, the first coming car will be the first served one.

Incidentally, serving or dequeuing the car process is done using in-valet workers. These workers will be created as separate threads will check periodically for the availability of parking cars. In addition, the car will be parked in the available slots, these slots will be represented using a HPIFO Queue (Highest Priority In First Out), which is a priority queue assigns the highest priority for the car that has less staying time (remaining time to leave). Thus, as we mentioned later, the in-valet threads will check the availability of the slots which is a vacancy in this priority queue. However, we will get back to talk about the out-valet threads. These threads will check for the cars which have completed their staying time, and it is the time for them to leave. Hence, these threads will empty the parking slot, as well as updating and recording the related states and data regarding these cars. However, each car has a counter declares its remaining time in the park. Therefore, a monitor thread is used to update the remaining time for each car periodically. Nevertheless, these queues are being shared among multiple threads. Hence, we should avoid race conditions when we want to add, delete, or show the queues graphically. There for, each queue will have its own mutex lock to avoid race condition and prevent more than one thread to manipulate the same queue. Furthermore, since the priority queue is used for the parking slots, it will be shared between both in-valet and out-valet threads. Where the in-valet threads will check if there is an available slot and to park a new car in it, and out-valet threads will make sure that the slots are not empty, then no need to get inside. Hence, two counting semaphores will be employed to satisfy this condition. Eventually, we reach to the end of our discussion of the design in general. Let us summarize the used locks and semaphore in this project:

1- A mutex lock for the FIFO queue, for incoming cars. Named “Qlock”.
2- A mutex lock for the HPIFO queue, for parking slots, named “PQlock”.
3- A counting semaphore indicates if the car park has empty slots. Named “empty” and initialized to the number of parking slots.
4- A counting semaphore indicates if the car park has ready-to-left cars. Named “full” and initialized to Zero.

2. Main Algorithms Design:

In this subsection, we will be exhibiting the main used algorithms in this project. However, this section is updated and is not the same as the progress report. it contains the brief explanation of the main threads, and how do they apply the design choices justified above.

2.1 Main Thread:

The main thread job is to generate new cars and put them into the arrivals FIFO queue. Hence, it takes the waiting queue lock, puts the generated cars in it, then releases the lock. The algorithm is shown below:

```
while(!stop){
        Qlock.acquire();                    // acquire the waiting queue lock
        put_the_generated_cars_in_waiting_queue();
        Qlock.release();                    // release the waiting queue lock
}
```

2.2 In-valet Threads:

This thread simulates the in-valet behavior, where its job is to take a car from the waiting queue, then put it to the parking slot. Hence, it acquires the waiting queue lock, take the first arrived car, then releases the lock. Then, it waits the empty semaphore, if it is available, it will acquire the parking slots lock, put the car in the park, then releases the lock. And it will not send a signal to the full semaphore because this will cause a deadlock. Sending the signal to the full semaphore will be done in the monitor thread later. The algorithm is shown below:

```
while(1){
        // 1. taking the car from arrivals queue
        Qlock.acquire();          // acquire the waiting queue lock
        serveCar = Qserve();      // take the first arrived car from the waiting queue
        Qlock.release();          // release the waiting queue lock
        // 2. putting the car into parking slots
        empty.wait();             // wait for an empty slot
        PQlock.acquire();         // acquire the parking slots lock
        PQenqueue(*car);          // put the car in the slot
        PQlock.release();         // release the parking slots lock
}
```

2.3 Out-valet Threads:

This thread simulates the out-valet behavior, where its job is to take the cars that have reached there leaving time and let them leave the parking slot. Thus, it will wait the full semaphore to be available. Then, it will acquire the parking slots lock. Next, it will validate that the car in the peak of the priority queue has reached its time. After that, if it is reached its time, it will signal the empty semaphore after removing the car from the parking. Eventually, it will release the parking slots lock. The algorithm is shown next:

```
while(1){
        full.wait();             // wait for signal indicates a car wants to leave
        PQlock.acuire();         // acquire the parking slots lock.
        serveCar = PQpeek();     // check the car that wants to leave

        if(serveCar->ltm <= 0){  // if it is really its time has come
                PQserve();       // remove it from the parking slots
                Empty.signal()   // signal the semaphore that indicates vacant slots
        }
        PQlock.release();        // release the parking slots lock
}
```

2.4 The Monitor Thread:

This thread prints out the run-time simulation states periodically, such us each slot and which car is parked in. Also, it checks and updates the leave time for the cars in the parking slots. Moreover, when a car wants to leave it will signal the full semaphore. However, in the progress report we did not do the monitor thread and we were signaling the full semaphore in the in-valet thread, which caused a deadlock. Therefore, this thread will acquire the parking slots lock, update the leaving time of each car, signal the full semaphore if the car has ended its parking time. Then, it will release the lock before leaving. The algorithm is shown below:

```
while(1){
        PQlock.acquire();                // acquire the parking slots lock
        Print_cars_in_slots();           // print each slot and the id of the car in it
        Print_num_of_parking_cars();     // print the No. of currently parking cars
        // updating the leave time for each car
        for (int i = 0; i < psize; i++) {
                update_remaining_time_of_car(i);  // update remaining time of this car.
                // if a car wants to leave, signal the full semaphore
                if (parkings[i]->ltm <=0){
                        full.signal();
                }
        }
        PQlock.release();        // release the parking slots lock.
}
```