بسم الله الرحمن الرحيم

King Abdulaziz University – Faculty of Engineering - EE-463

Operating Systems

Operating Systems Project#B

Car Park Simulator

Name	ID
Hesham Taha Banafa	**1742275**
Muhannad Saeed Alghamdi	**1846525**

The Design of the Solution

The overall design starts with the main thread reading the command line inputs and initialize the car park simulator. The main thread then gets into an infinite loop where the new cars will be generated randomly based on the poison distribution. The new cars then will be inserted in a waiting queue. The waiting queue can be accessed by the in-valets which are represented by a thread-pool to take the cars from the queue and then park them in the parking area. The parking area is represented by an array of cars if the parking slots are empty then it will be set to null. On the other hand, the out-valets which are represented by another thread-pool can access the parking area to find the car that is ready to exit, to take it out to the client and set that parking slot to null. The overall design of the solution is shown in **Fig.1**. The solution requires two data structures, as follows:

1) **Queue of waiting cars** represents the cars waiting for the in-valets.
2) **Array of parked cars** represents the parking area where each element in the array is considered as a parking slot.

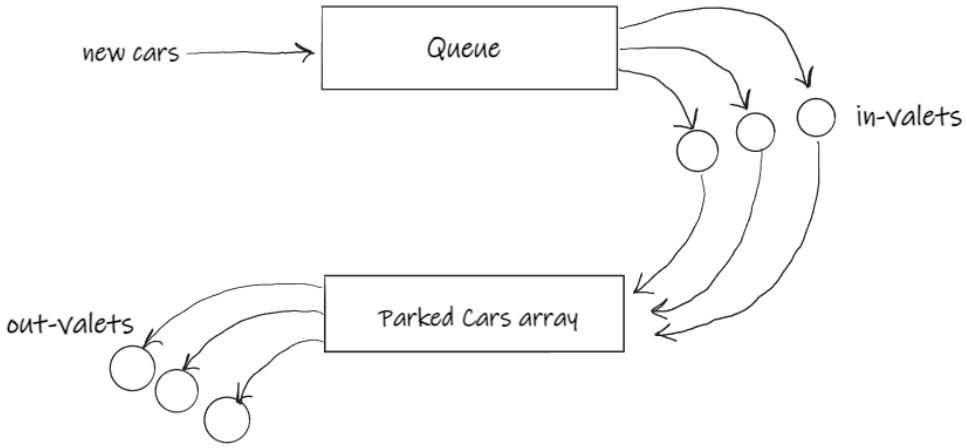

Figure 1 the overall design of the solution for the car parking simulator

As the two data structures, array and queue are shared between multiple threads, then synchronization techniques are required for our solution. **Fig.2** shows where exactly we are expected to have synchronization problems. The problem of synchronization between the multiple threads of the in-valets so they do not access the queue at the same time. Another problem with the queue is similar to the bounded buffer problem between the main thread which will generate the new cars and the in-valets which will serve the cars. So, the need for waiting for arrival cars and signaling when a new car is arrived at the queue, and the need of waiting for empty slot in the queue before a new car is inserted is essential in this solution.

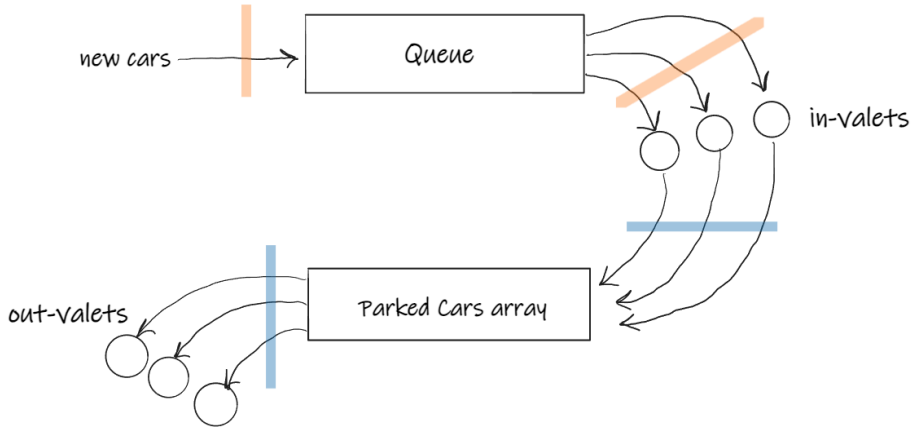

Figure 2 the areas where synchronization problems are expected to appear.

On the other side, the parked cars array is accessed by multiple in-valets where all of them are trying to insert new cars in an empty slot. This needs a write lock to prevent the multiple threads of the in-valets to access the parked cars at same time and write to same location by mistake. One additional problem with the parked cars array is that it is accessed also by the out-valets where the search for the car which suppose to leave and then they take it out to the client. This will need the same write lock so the multiple threads of the out-valets and in-valets can not access the same car at the same time, and protect the car park array. The write lock for protecting the park array is called "writer", and is also shared with CarPark GUI program section for proper display and safe read. And as we have both the in-valets and the out-valets accessing the parked cars array we need to signal when a car is served by the out-valets to indicate an empty parking slot, rather then becoming busy waiting while trying to add new cars to the park.

After the previous discussion about the potential problems that we are expected to have, then we need provide solutions to deal with these potential problems. **Fig.3** shows the synchronization techniques used with the waiting queue and how it is accessed by the multiple threads of the in-valets and the main thread to add and serve the coming cars.

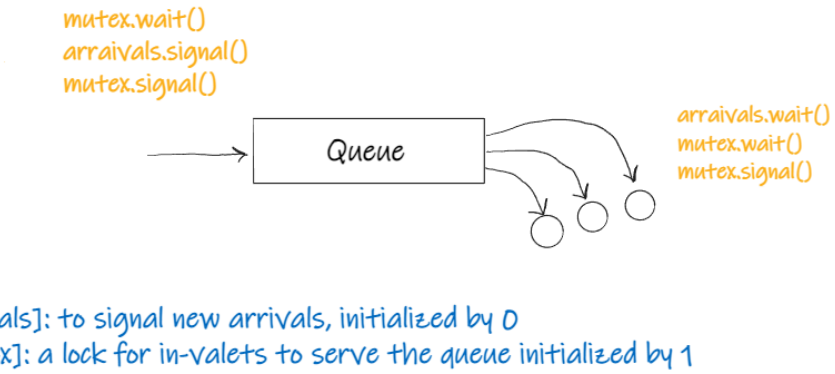

Figure 3 the synchronization solutions for the waiting queue

The main thread will generate new cars and attempt to enqueue them in the queue if there is room available, otherwise the car is turned away and recorded as a turn away. After the new car is inserted into the queue, it will signal the arrivals semaphore as a new car has arrived at the queue. The in-valets then will be notified and start serving the car after it acquires the lock after that it will notify the main thread by signal the empty1 semaphore to let it know that there is an available empty slot in the queue. Then it releases the lock by signaling the mutex. On the other hand, the parked cars array with its synchronization solution are shown in **Fig.4**.

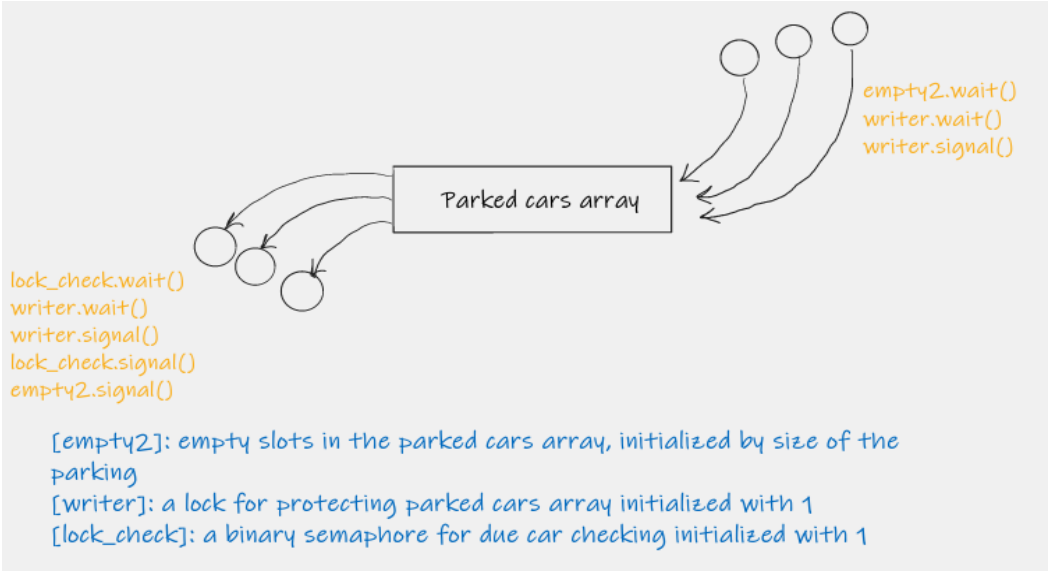

Figure 4 the synchronization techniques used for the parked cars array.

The in-valets start by waiting for empty slots in the parking area, if there is a parking slot then it acquires the lock so no other in-valet is accessing the parked cars array at the same time. It then releases the writer lock to let other in-valets in. On the out-valets side, valets check the car park array for due car. If a car is due, it is removed from the data structures using the same critical section for coherency. We note in our design, it is forced that only one valet checks the parked cars at any one moment by using the "lock_check" semaphore. After a car is due, out-valet signals the empty semaphore so in-valets can bring more cars.

Pseudo-code for the solution

In this section, we will discuss the pseudo-code for the main three parts of this solution which are more likely to have synchronization problems discussed above, the three main parts are:

I) The main loop (generating new cars)
II) The in-valets
III) The out-valets

We will discuss each section, with its pseudo code used to solve the synchronization problems which were discussed in the previous section and how the implementation of the different solutions discussed above can be used together in order to solve the above problems.

I) **The main loop:** generating new cars

```
main loop:
if (q.isFull())
then
rf++
continue
else
pk++
mutex.wait()
q.Qenqueue(car)
mutex.signal()
```

II) **The in-valets:** serving the in-coming cars

```
arrivals.wait()
if (turn != thread_id) continue // Simulate a RR for even work
turn++; if (turn == num_in_valets) turn = 0;
mutex.wait()
Q.serve()
mutex.signal()
delta = time() - car.atm
sqw_mutex.aquire()
sqw += delta
sqw_mutex.release()
in_held_mutex.wait()
nm++
in_held_mutex.signal()
empty2.wait()
writer.wait()
oc++
//find NULL locations in the parking array
// replace it with the new car
writer.signal()
in_held_mutex.wait()
nm--;
in_held_mutex.signal()
```

III) **The out-valets:** serving the out-coming cars

```
lock_check.wait()
// Check if turn (same as in-valets)
car = NULL
while (car == NULL) do
writer.wait()
//loop through parking slots
//find car that its due is now
//remove the car
empty2.signal()
delta = time() - car.ptm
writer.signal()
done /* While */
lock_check.signal()
spt_mutex.aquire()
spt += delta
spt_mutex.release()
```

IV) Monitor thread

```
ut = oc / psize
ut_privous = 0
while (1)
ut_now = oc / psize
ut = (ut_privious + ut_now) / 2
ut_previous = ut_now

updateStats()
show()
printStats()
```

Documentation of the used functions

I) monitor

```
/**
   Entry point for monitor thread , 15/4/2022
   @param Pointer to thread arguments array
   @returns void
   @author Muhannad Al-Ghamdi
*/
void *run_monitor(void *args);

/**
   Calculate car-park cumulative utilization , 15/4/2022
   @param  double ptr  varible of pervious calculation
   @param  double ptr to utilization variable
   @return void
   @author Muhannad Al-Ghamdi
   @precondition N/A
*/
void calc_utilization(double *ut_previous, double *ut);

/**
   Print current status of the carpark, 15/4/2022
   @param void
   @return void
   @author Muhannad Al-Ghamdi
*/
void print_stats();
```

II) in-valets

```
/**
   Initialize in-valet thread-pool, 15/4/2022
   @param Number of valets to start
   @returns 0 if successful, 1 if failed
   @author Muhannad Al-Ghamdi
*/
int init_in_valets(int number_valets);

/**
   Entry point for in-valet thread , 15/4/2022
   @param Pointer to thread arguments array
   @returns void
   @author Muhannad Al-Ghamdi
*/
void *run_in_valet(void *args);
```

III) Out-valets

```
#include <pthread.h>

/**
   Initialize out-valet thread-pool, 15/4/2022
   @param Number of valets to start
   @returns 0 if successful, 1 if failed
   @author Hesham T. Banafa
*/
int init_in_valets(int number_valets);

/**
   Entry point for out-valet thread, 15/4/2022
   @param Pointer to thread arguments array
   @returns void
   @author Hesham T. Banafa
*/
void *run_in_valet(void *args);
```

Time scheduling and task allocation

NO.	Task	Member	Due data
Phase One: Design the solution			
1	Discuss the project requirements	Muhannad - Hesham	12 Apr. 22
2	Discuss different potential solutions	Muhannad - Hesham	
3	Choose a doable solution	Muhannad - Hesham	
4	Deep discussion of the selected solution	Muhannad - Hesham	
5	Find potential problems and their solution	Muhannad - Hesham	
6	Fix the overall design of the solution	Muhannad - Hesham	
7	Progress report write-up	Muhannad - Hesham	19 Apr. 22
Phase Two: Implement the solution			
8	Implementation of the Queue	Hesham	16 Apr. 22
9	Implementation of the in-valets thread handler	Muhannad	24 Apr. 22
10	Implementation of the out-valets thread handler	Hesham	24 Apr. 22
11	Implementation of the main thread	Muhannad - Hesham	19 Apr. 22
12	Implementation of the monitor thread	Muhannad	25 Apr. 22
Part Three: Testing and validation			
13	Testing and validation	Muhannad - Hesham	29 Apr. 22

Testing and validation

Elaborate testing for edge cases of synchronization deadlocks and memory leaks were conducted to ensure correctness of designed solution. Moreover, testing for longer periods of time to detect cases of starvation indicated some concerns. In terms of dynamic memory allocation and deallocation, a great rule of thumb is to ensure every frequent allocation call, has a corresponding free call. In this project, a frequent allocation is done during the generation of new cars into the park. Cars go out of existence in 2 places, either in attempting to join a full queue, near the allocation. Or at after finishing the park period. In **Fig.5**, we show that after a long running simulation, the top program shows the memory usage to be constant in our testing.

Figure 5: Memory leak testing using top UNIX program

Furthermore, testing for deadlocks in the early design showed a deadlock is possible when a “parked” lock was used to wake the out-valets. It is a fundamental flaw since there is a case where no cars are due, the last empty slot is filled, and the out-valets will never get signalled. Since out-valets are only signalled. This issue is addressed in the modified code above.

It was observed during the testing that due to OS scheduling and the nature of POSIX semaphores not enforcing a FIFO in signalling sleeping threads, that some valets take up to 50% the cars of others. It in actuality not a critical issue, and can not be explicitly defined as starvation. But a basic turn based approach (round robin) is implemented to evenly distribute the work. We note this does not block during the full operation, it only blocks at the start. And by block we mean yield to others.

Future improvements

There is much to improve from our final design, to address time complexity. A major improvement to the out-valets is to implement a priority-queue to provide out-valets with a constant time O(1) search for due cars by peeking the top of the queue. Rather than the current basic solution of a linear search, which is O(n) on average. A priority queue was implemented but was never integrated fully into the simulation program.

Another area to improve is to make the search empty slots also constant time by having a simple linked list or a queue, with nodes of information of an empty slot. This should structure is changed during adding a new car and removing from the park. We also do not seek through this structure.