بسم الله الرحمن الرحيم

King Abdulaziz University – Faculty of Engineering - EE-463

Operating Systems

Operating Systems Project#B

Car 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 it has come, 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 needs two main 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.

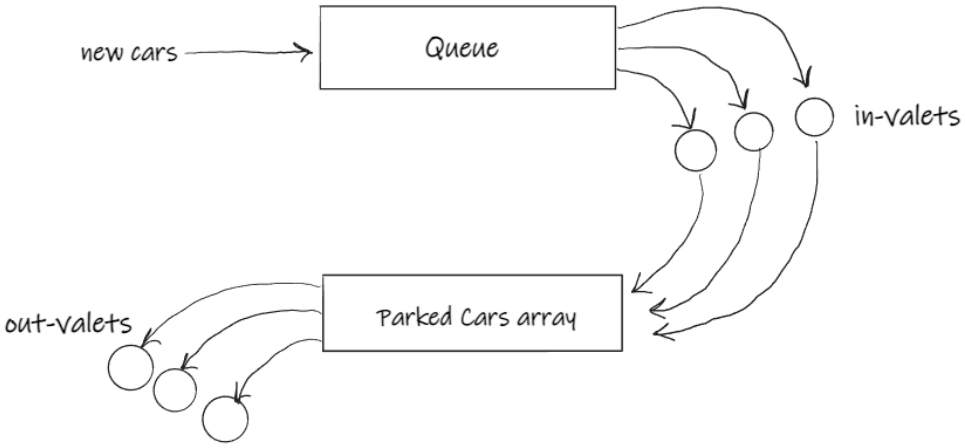

Figure 1 the overall design of the solution for the car parking simulator

As the two data structures the array and queue are shared between multiple threads then synchronization techniques are a most 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.

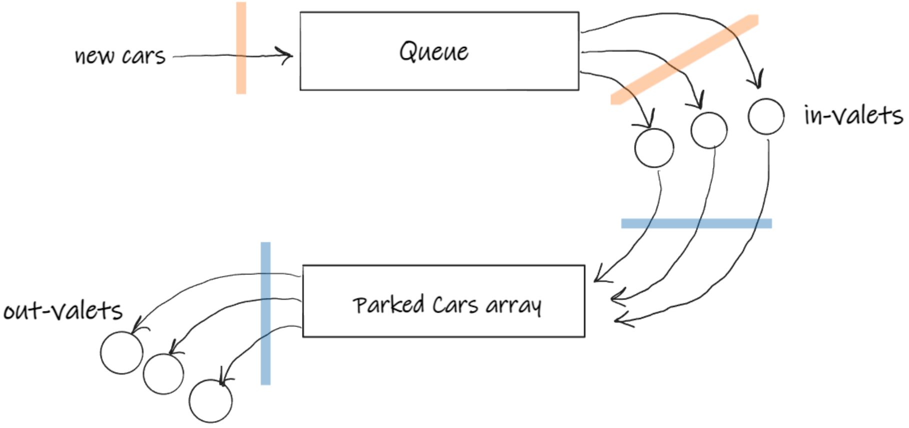

Figure 2 the areas where synchronization problems are expected to appear.

On the other side, the parked cars array is accessed by multiple in-valets where all of them is trying to insert new car in an empty slot. This needs a 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 a lock so the multiple threads of the out-valets can not access the same car at the same time. 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 and to signal when the a new car is parked so the out-valets can take care of it when its due time.

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.

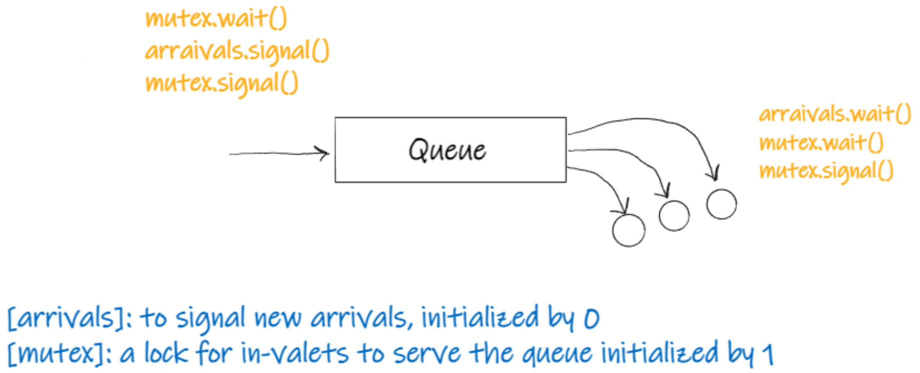
mutex.wait()
arraivals.signal()
mutex.signal()
Queue
arraivals.wait()
mutex.wait()
mutex.signal()
[arrivals]: to signal new arrivals, initialized by 0
[mutex]: a lock for in-valets to serve the queue initialized by 1

Figure 3 the synchronization solutions for the waiting queue

The main thread will generate new cars and then wait for empty slots in the queue if it is available then it will insert the car into the queue. After it inserts the new car, it will signal the arrivals 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**.

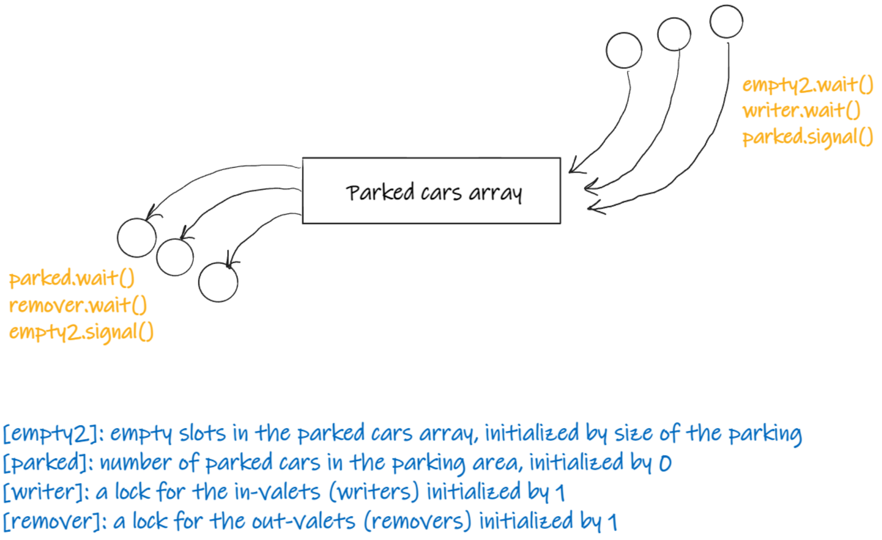
empty2.wait()
writer.wait()
parked.signal()
Parked cars array
parked.wait()
remover.wait()
empty2.signal()
[empty2]: empty slots in the parked cars array, initialized by size of the parking
[parked]: number of parked cars in the parking area, initialized by 0
[writer]: a lock for the in-valets (writers) initialized by 1
[remover]: a lock for the out-valets (removers) initialized by 1

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. After parking a car it signals the parked semaphore to let the out-valets know that there is a new parked car. It then releases the writer lock to let other in-valets in. On the out-valets side it starts by waiting for parked cars then it acquires the lock for remover to ensure no other out-valets are in and then it removes the car that its due time is now. After that is signals the empty semaphore so in-valets can bring more cars and then it releases the lock to let other out-valets in.

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++
else
pk++
mutex.wait()
q.Qenqueue(car)
mutex.signal()
```

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

```
arrivals.wait()
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
parked.signal()
writer.signal()
in_held_mutex.wait()
nm--;
in_held_mutex.signal()
```

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

```
parker.wait()
remover.wait()
//loop through parking slots
//find car that its due is now
//remove the car
delta = time() - car.ptm
spt_mutex.aquire()
spt += delta
spt_mutex.release()
empty2.signal()
remover.signal()
```

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

EE463 Operating Systems Term Project-B Memo 04/20/22

Hesham T. Banafa
Muhannad Al-Ghamdi

Current state summary:

The car-park is operating correctly in terms of generating new arrivals based on the provided RNG from the main loop. Currently, the main loop generates, allocates and initialize incoming cars, then later added to the Car-park queue if preconditions checkout. We have also implemented in-valets code for thread management and synchronization of shared data of the process. The in-valets wait on semaphore initialized to 0 named ‘arrivals’ accordingly with it’s purpose. It is noted that the semaphore implementation used allow the threads to sleep until the semaphore is ‘posted’ or ‘signaled’. In-valets currently take arriving cars from the queue and park them in empty slots.

Finished:

Queue implementation is complete and tested to operate correctly.
Code is split into files for isolation (main, monitor, in-valets, out-valets).
Main loop generating incoming cars.
Monitor calculates some statistics, updateStats, print to terminal, and re-draw GUI window.
GUI working with queue.
CLI arguments parsed correctly according to Project document.
Makefile updated to compile correctly.

Work in progress:

SIGINT and SIGTERM handling: graceful thread termination and cleanup and print final car-park report to stdout.

Out-valets: initial code as per design.

Testing for deadlocks: theory and testing.

Problems during work:

C language tool chain does not allow multiple includes. Having circular includes produces this compile-time error. It was later found that it is common practice to use ‘include guards’. The following illustrates the pre-proccessor directives for include guards.

```
#ifndef HEADFILENAME_H
#define HEADFILENAME_H
{{ Header content, varibale and function decelerations }}
#endif
```

During compile-time, the per-procseesor checks if the compile-time symbol HEADFILENAME_H is defined to decied whether include the whole header again. It serves the purpose of “include once” or

```
#pragma once
```

Falsely implemented CarInit when it is already implemented in CarPark.o. ‘ld’ output mentions the CarInit is defined multiple times.

EE463 Operating Systems Term Project-B Memo 04/20/22

Hesham T. Banafa
Muhannad Al-Ghamdi

Not Started:
Out-valet functionality.

Calculations of all statistics.

Proper thread termination is good practice, despite the fact that the OS (Linux) frees all the resources held by the starting process after terminaion. Graceful termination of the program can be done in 2 common methods. The first method is to send pthread_cancel() to each thread. Ideally, the thread should terminate immediately. However, if the thread unsets ‘cancel state’ with pthread_setcanelstate(), a cancel request is then queued until the thread re-enables cancellation.

Moreover, pthread_setcanceltype() alllow to set asynchronous cancel where the thraed is canceled at any point, which may introduce data corruption of shared data is used. Another type is deferred, where the thread is canceled when it reaches a cancel point, by calling pthraed_testcancel() when ever is safe.

The second method, which is more reliable in our testing is to use a shared flag. Since threads by default share data with the parent thread, we can make the thread check the value of a global flag to indicate when it should call pthread_exit().