EE 463	**Operating Systems**	Date: 21/06/1444
Winter 2023	**Term Project**	14/01/2023
Dr. A. M. Al-Qasimi		**Due: 15/02/2023**

Introduction:

In this project, you will be developing a simulator application program that illustrates two forms of IPC that you have studied so far: shared memory and signals; also, multi-threading, thread-pools, mutual exclusion and several synchronization techniques. You are asked to write a simple, **car-park** simulator with graphical visualization under the Linux operating system. To simplify your work, we have provided a modest graphical visualization library with a set of functions that you can call from your simulator code.

To offer VIP service to their clients, the car-park management of a 5-star hotel decided to employ valets to handle the parking and/or un-parking process of their client cars. In their view, this would make the car-park service as responsive to their VIP clients as possible by minimizing the average client-waiting-time. A client arrives at the front door of the hotel, he/she waits for available valet, possibly in a queue; valets acquire the client's car, record how long he/she wish to stay, and drive it to the car-park, possibly waiting for an available slot before they can park the car; when parked, they record the slot number, the current time and the intended leaving time. Cars stay parked until their leaving time. Valets periodically check which parked car is due to be un-parked; fetch that car from the car-park and deliver it to the client at the front door of the hotel; recording the actual time it stayed parked.

Objectives:

This simulator has to solve some of the problems you studied in the operating systems course, but it is easier to understand, modify and test simulation code rather than modifying and testing the code of a real operating system. This is especially true when the simulator provides graphical output. In this project you are going to gain hands-on experience in the following areas:

- Extending the *LAMP* virtual Linux OS to have a GUI interface, so it can handle graphical output.
- Designing, developing and extending a modular simulator application.
- Developing statistical data for observing the simulated system performance.
- Creating and synchronizing multi-threaded programs in Linux.
- Using and protecting shared-data.
- Using signals to communicate events to your program.
- Graphically watching the effects of deadlocks and starvation.
- Creating some of the data structures that actual OSs use to efficiently manage their data.
- Working as a team.

Prerequisites:

Before you can work on the project as a team, you should work individually to install some kind of a GUI in your virtual Linux OS. You should do the following steps before you start the team work:

Step #0:

If you did not download the *LAMP* virtual machine before, do it now, it is available at the course web site. Those of you who are using Mac OSX or another Linux with GUI should do the same as well. The *LAMP* virtual machine runs *Debian Linux* which has only a console based CLI. Any programs using graphics will not work on it as it is. Since our project requires graphical display, then you need to provide one.

Important: For steps 1 – 4 only, login as root user.

Step #1:

Each student shall install a GUI Desktop to his/her *LAMP* virtual machine. Your choice of GUI must be considered carefully and seriously so it should not overwhelm your computer resources or make it run slow. Be careful; take precautions not to ruin your virtual machine.

Step #2:

Once you are done installing a GUI, reboot the *LAMP* virtual machine, and then use the `apt` utility to update the sources and install the required SDL libraries. These will provide a portable graphical library that can be used to write sophisticated graphical applications. Install the packages in the shown order by typing the following at the command prompt:

```
# apt update
# apt-get install libsdl2-2.0
# apt-get install libsdl2-dev
```

Step #3:

Download the project's package `EE463W23PRJ.zip` from the course website, unzip it and upload all the files to the *LAMP* virtual machine. Type the following at the command prompt to install the final piece of software, a simple graphics library that depends on SDL:

```
# dpkg -i sdl_bgi_2.5.0-1_amd64.deb
```

Step #4:

Now your system is ready to run programs that require graphical display. To do so, use a console terminal in the GUI Desktop and type:

```
# ./<program_name>
```

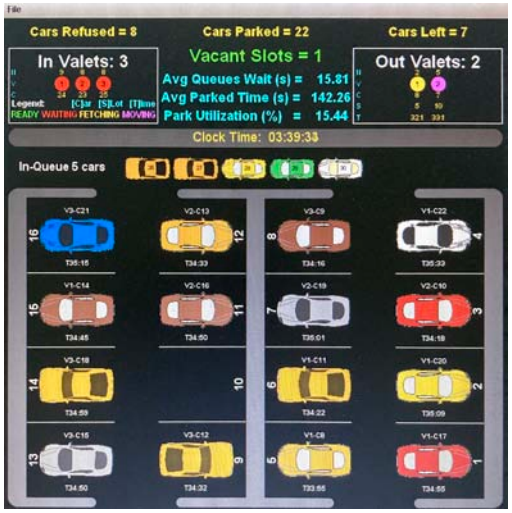
In the downloaded project's package, you will also find two more files:

- `carpark.tgz`: Provides a package for a humble *graphical-output* interface for a car-park simulator.
- `ReadMe.txt`: Introduces the contents of the package and a full explanation of how to install and use it on your virtual Linux system.

Read the **`ReadMe`** file and install the package, as directed, in a work directory on your virtual LAMP computer. Verify that you have all the described files, read the documentation provided in the three header files, and then you are ready to start working on the project as a team.

What to Do:

Work in groups of at most 3 students. Each individual project group shall design and write a C application under Linux to simulate the car-park as described above. In doing so, use **`thread-pools`** for in-valets and out-valets, where, **`in-valet`** threads simulate valets behavior when handling arriving cars; and **`out-valet`** threads simulate valets behavior when handling leaving cars. A **`monitor`** thread shall periodically read and display the status of the car-park as seen in the following figure:



The `main` thread shall read command-line arguments, initialize the car-park simulator, setup a `SIGTERM` signal handling function and then enters an endless loop where it shall randomly create incoming cars with *Poisson* distribution set according to the expected value entered at the command line. These cars shall be put in an arrival queue; if the queue is full, then an arriving car is refused service and sent away. When the `main` thread receives a `SIGTERM` signal, it shall cleanup (i.e. cancel the other threads, free dynamic memory and delete all the locks …etc.), shutdown the graphical system and exit.

The interactions among all the threads require proper synchronization; they also may require access to shared data, which must be properly protected. It is your job to design the shared data, its means of protection against corruption and the necessary thread synchronization such that the following rules are observed:

- Implement mutual exclusion where appropriate.
- Use `Pthread locks`, `condition-variables` and `semaphores` for synchronization & mutual exclusion.
- Do not remove cars from an empty car park or an empty slot.
- Do not add cars to a full car park or an occupied slot.
- Avoid busy-waiting, starvation and deadlocks.
- Each thread must pause for a random period (up to 1s) before and after parking or removing a car.
- Each thread must pause for a random period (up to 0.2s) in its critical section.
- Cars shall stay parked a random period of time (from 1s to 180s).
- No car shall be un-parked before its intended leaving time.
- The `monitor` thread shall periodically print & graphically display the current status of the car park.
- Your simulator shall continue working until it is interrupted from the keyboard by `control-c`.

The simulator shall get the values of the simulation parameters from command line, and shall have default values for missing arguments. All parameters shall have *enforced* lower and upper limits, they are as follows:

1.	The car-park capacity:	An integer multiple of 4 in the range [12 – 40];	Default = 16.
2.	The number of in-valets:	An integer in the range [1 – 6];	Default = 3.
3.	The number of out-valets:	An integer in the range [1 – 6];	Default = 2.
4.	The arrivals queue capacity:	An integer in the range [3 – 8];	Default = 8.
5.	The arrivals expected number:	A real number in the range [0.01 – 1.50];	Default = 0.05.

Program Specification:

Your **C** program shall be invoked exactly as follows, note that items inside [] are optional:

```
$ ./carpark [psize inval outval qsize expnum]
```

The command line arguments to your car-park simulator are to be interpreted as follows:

- `psize`: The car-park capacity.
- `inval`: The number of in-valets.
- `outval`: The number of out-valets.
- `qsize`: The capacity of the arrivals queue.
- `expnum`: The expected number of arrivals. This is used in the *Poisson* random number generator, according to which new arriving cars are created in the main loop.

For example, if you run your program as:

```
$ ./carpark 50 10 10 10 1
```

or

```
$ ./carpark 40 6 6 8 1.0
```

Then your simulator shall have car-park capacity of 40, with 6 in-valets, 6 out-valets, queue size 8, and 1.0 expected number. If instead you run the program as:

```
$ ./carpark 16 3 2 8 0.05
```
or
```
$ ./carpark
```

Then your simulator shall have all default arguments: car-park capacity of 16, with 3 in-valets, 2 out-valets, queue size 8, and 0.05 expected number. If you run the program as:

```
$ ./carpark 10 5
```

Then your simulator shall have car-park capacity of 12, with 5 in-valets, 2 out-valets, queue size 8, and 0.05 expected number.

Simulation Statistics:

While running, the simulator shall gather and display some data about the performance of the car-park operations, these include the following:

- `oc`: Current number of occupied slots in the parking space.
- `nc`: Running total number of cars created during the simulation
- `pk`: Running total number of cars allowed to park
- `rf`: Running total number of cars not allowed to park
- `nm`: The number of cars currently acquired by in-valets
- `sqw`: Running sum of car-waiting times in the arrival queue
- `spt`: Running sum of car-parking durations
- `ut`: Current car-park space utilization

The output from the `monitor` thread must clearly show that your simulator is working as specified above. Study the output of your program to check it is operating properly, i.e. cars are continuously arriving, parking and leaving, consistently; all valets are actively working; no cars and no valets should be in neither deadlock, nor starvation states and the system shall not have busy-waiting problems. Upon receiving a ctrl-c signal, the simulator shall print the values of all the input parameters and the final values of the simulation statistics before it exits. See the following sample output for an example:

```
Mon Mar 14 18:47:34 AST 2022:    Received shutdown signal ..
Mon Mar 14 18:47:34 AST 2022:    Car park is shutting down ..
Mon Mar 14 18:47:34 AST 2022:    The valets are leaving ...
Mon Mar 14 18:47:50 AST 2022:    Done. 5 valets left.
Mon Mar 14 18:47:50 AST 2022:    Monitor exiting ...

Simulator started at:       Mon Mar 14 18:42:24 AST 2022
 Park Space Capacity was:   16 cars.
 Allowed queue length was:  5 cars.
 Number of in valets was:   3.
 Number of out valets was:  2.
 Expected arrivals was:     1.0.
Simulator stopped at:       Mon Mar 14 18:47:35 AST 2022

 CP Simulation was executed for:   311 seconds
 Total number of cars processed:   266 cars
   Number of cars that parked:      22 cars
   Number of cars turned away:     238 cars
   Number of cars in transit:        1 cars
   Number of cars still queued:      5 cars
   Number of cars still parked:     16 cars

 Average queue waiting time:        40.178 seconds
 Average parking time:             170.375 seconds
 Percentage of park utilization:    20.467 %

Mon Mar 14 18:47:50 AST 2022:    CarPark exits.
```

Hints:

- **Start early**.
- Read about the pthread API and the way threads should be setup and used.
- Start designing your simulator by identifying what components you need to write code for.
- Fully specify any new components. These may be new *data structures* and/or new *functions*. Specifying those means to write in detail, what they are, what is their role in the system, how they interact with other system components, and how can they be used. This does not include the detailed implementation code. These specifications together with any *illustrations*, *drawings*, and/or *tables* should comprise your system *design document*.
- Be careful not to introduce memory leaks! i.e. when you create dynamic variables, they need to be freed when they are not required anymore. [Every malloc() call must have a matching free() call.]
- You can observe what happens during the test run by using the following Linux command:

```
$ top -d 0.1 -H -p<carpark-process-id>
```

  This will list, every 0.1 seconds, all the threads associated with the specified process id.
- Use the experience you learned from the lab experiments.
- **Start Early**. Again!? Yes, we repeated it for emphasis. It is so important.

What to turn in:

Your project must be submitted through Blackboard in three parts:

- An early progress report is due on Thursday 26/01/2023, at 11:59pm.
- A second progress report is due on Sunday 05/02/2023, at 11:59pm.
- A final full report is due on Wednesday 15/02/2023, at 11:59pm.

Your first submittal must include the following items:

1. Your project **time schedule** and the distribution of project tasks to the group members.
2. Your **design document** for the car-park implementation. This is an important project document. It must include your overall simulator algorithm, all non-trivial algorithms and formulas, all required data structures, and a detailed description of each function, that includes: its purpose, inputs, outputs, preconditions, author and the date of last modification.
3. A brief, two-page **memo**, summarizing the current state of the project, including: finished parts, parts you are still working on, parts not started yet and any difficulties or problems still not resolved (other than the usual excuses like, we had exams, computer or family problems …etc.).

Your final report must include the following items:

1. The final updated version of your **design document**, including any critical changes you have made since the last report and the reason(s) for the change. Note: Highlight the changes only.
2. A **README** file that describes:
   a. What are the difficulties, if any, that negatively affected your project outcome.
   b. What requirements that you designed and realized.
   c. What requirements that you designed but failed to realize, and why.
   d. The project's build command and any special requirements we need to be aware of.
   e. Your comments about the observed output and performance of the system.
3. One .zip file containing a cover-page and the compressed files of your project as follows:
   a. *Source programs*: only **.c** and **.h** files, and your **Makefile**. No executable or .o files please. You must do a "`make clean`" before creating the .zip file. I will copy them to my LAMP virtual machine and build the executables using your source files and your **Makefile**, and run them there. So, please test your files before you submit them.
   b. All the documents from items 1 and 2: The final **design document** and your **README** file.
   c. A meaningful sample of your program's output in .pdf form.
   d. A PowerPoint presentation for your project.

Each individual student is required to give a presentation about the work he/she did for this project. Presentations are to be scheduled later.

Grading the project:

You need to do a careful design and implementation of the project. Remember that you can't pass this course without at least making a serious attempt at this project assignment.

The grade for this project will be divided as follows:

- **10%** for meaningful and acceptable early progress report showing a real, time relative progress.
- **10%** for meaningful and acceptable second progress report showing real progress.
- **25%** for a well written, detailed and clear final design document and README file.
- **30%** for the complete and working implementation, i.e. the car-park simulator.
- **10%** for a clear, well documented and commented program.
- **15%** for a well organized and clear presentation.

Important:

1. You were given a relatively adequate time period to complete your project. Should any group **submit a broken program** or a **non-working implementation**, and **no sensible design**, this will only show that the whole group is incompetent, careless or did not work hard enough to successfully complete the project. Therefore, the entire project would be **worthless** and **will not be evaluated**. Such a group (if any) will receive zero (0) for the project. No excuses.

2. All requirements must be submitted on the due dates. No late submittals will be accepted at all.

3. Do not worry too much. Start working and Enjoy. **Good Luck**.