CSE-231 Operating Systems Section B
Assignment 4
Topic: Concurrency and Synchronization Primitives

Please note:

All questions have to be done using the C programming language only.

The general guidelines mentioned in each question will ensure more readable code and an easier time during the demo for both you and your evaluator, so please follow them.

Helper functions can be added as necessary. Ensure that you are only using the synchronization primitive as mentioned in the question.

Question 1:

The dining philosophers problem (OSTEP Chapter 31, page 13) is a classic problem to demonstrate the concepts of deadlocks. The original setup contains five philosophers sitting on a round table, with a fork between each philosopher. Each philosopher can perform only one of two actions – eating and thinking. For eating, each philosopher requires 2 forks that are kept at their left and right sides. Allowing unrestricted access to each philosopher can lead to a deadlock.

Consider a modified version of the classic problem stated below:

Besides the 5 forks, there are now 2 bowls that are kept at the centre of the table. For eating, each of them now requires two forks and a bowl.

Model the above problem using threads as philosophers.

Specifically, each philosopher carries out 2 tasks, eating and thinking. Time required for both tasks can be simulated using the sleep() function (or any other function that causes a delay), and each philosopher must indicate what action it is carrying out by printing to the console.

You must also ensure that there are no deadlocks in the code.

You are only allowed to use Mutexes (locks) and Conditional Variables for this question

Guideline:

Please structure your code to contain the following functions:

```
void* philosopher(void* args)     // for running philosopher thread
void eating()                     // for entering eating state
void thinking()                   // for entering thinking state
```

Deliverables:

C code

Readme file explaining the following:

Why deadlocks can occur in the problem setup

How your proposed solution avoids deadlock.
Fairness of the solution i.e. for your implementation, which and how many of the 5 philosopher threads are able to eat, and a rough estimate of how often a philosopher is able to eat (if at all).

Question 2:

Imagine a situation where multiple passengers eagerly await their turn to take a ride in a car. This car has a limited capacity and can only set off when fully occupied, with a maximum of C passengers on board (where C is less than the total number of passengers). Passengers have the simple tasks of getting on and off the car, while the car itself must manage the loading, running, and unloading procedures. Passengers are allowed to board only when the car has completed the loading process, and the car can commence its journey once it has reached its maximum passenger capacity. Passengers can disembark from the car only after it has completed its unloading process.

Simulate the above by modeling the car and the passengers as threads. Take the total number of passengers and capacity as input from the user.

Simulate the above problem by modeling the passengers and car as threads. Specifically, the car thread has to do the following tasks:

- **Load specified number of passengers for the ride**
- **Wait for all passengers to get on the ride**
- **Run the duration of the ride**
- **Unload all the passengers until ride is empty**

Each passenger thread has to do the following:

- **Board the ride when it is available**
- **Get off the ride when the ride is over**

Time taken for each step can be simulated using the sleep function with appropriate duration, and every action carried out must be printed to console (in case of passengers, mention which passenger is carrying out the action using appropriate means). Synchronization can be carried out between car and passenger threads as necessary. Ensure that your code is deadlock-free. Use semaphores for synchronization.

<u>Guidelines:</u>

Please ensure that the following functions are present in the code:

```
void* car(void* args)            // car thread
void* passenger(void* args)      // passenger thread
void load()                      // loading car with passengers
void unload()                    // unloading passengers
void board()                     // passenger boards car
void offboard()                  // passenger gets off car
```

Deliverables:

- **C code**
- **Writeup explaining code logic and how you avoid concurrency bugs in code**

Question 3:

Consider the following problem:

There is a thin bridge between two sides of a river. Due to its size, only one car can travel from the left side of the river to the right and vice versa. The number of cars that can travel on the bridge simultaneously is 5. If 2 cars from opposite sides travel on the bridge simultaneously, they will not be able to cross, and travel will stop.

Modeling each car as a thread, write a program such that all cars from the left and the right side are able to cross without violating the above constraints (the number of cars on the left and right is to be taken as input from the user).

The following assumptions are also satisfied:

- **Once a car gets on the bridge, it will definitely cross it.**
- **Each car takes a fixed amount of time to cross the bridge. Crossing the bridge can be simulated as the thread calling the sleep() function.**

For every thread that crosses the bridge, you must indicate which thread it is and which side it is originating from by printing appropriate information to the console. Use semaphores for synchronization.

Guidelines/Hint:

It would be much easier to write different thread functions for left and right side cars. Therefore, please try and include the following functions:

```
void* left(void* args)          // cars on the left
void* right(void* args)         // cars on the right
void passing(int dir)           // car from some direction is traveling on the bridge
```

Deliverables:

- **C code**
- **Writeup explaining code logic and how you avoid concurrency bugs in code**