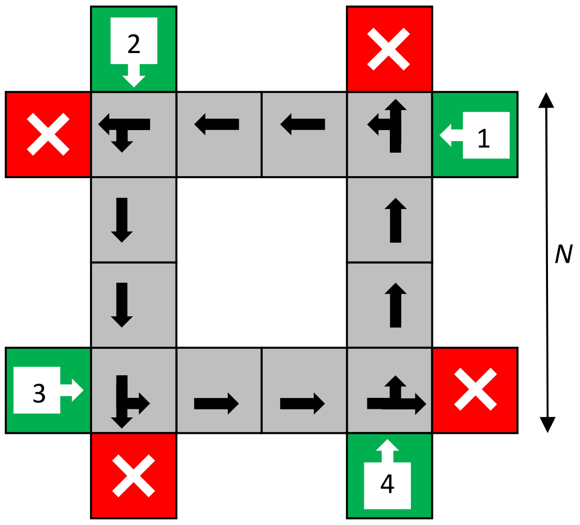
HW3

In this assignment you will practice multi-thread programming combined with mutual exclusion.
Assume that you've been asked by the MOT (Ministry of Transportation) to build a simulator of a traffic circle that will help them to predict the congestion in a junction, according to some parameters. The parameters are detailed (in capitalized letters) below.
The simulation structure is as follows:



The traffic circle is built of square units where each one of them may contain one car at most at any given time. The size of the square's side is N.
In the circle's corners, there are four "car generators" (green units). Each car generator generates a new car once in a randomized time, which is picked uniformly at random between MIN_INTER_ARRIVAL_IN_NS and MAX_INTER_ARRIVAL_IN_NS [ns] (nano-seconds).
A car that has been generated enters the circle only once the square to which it wants to enter AND the square before it are free.
A car tries to progress to the next square every INTER_MOVES_IN_NS [ns]. Then, if the next square is free, the car moves there; else, it stays in its current slot, until the next square is free.
Also in the circle's corners are 4 sinks. Each car moves around the circles. When a car progresses from a square before a sink, it disappears from the simulation with probability FIN_PROB; and continues moving around the circle with probability 1 - FIN_PROB. A car may vanish in a sink only after it has finished moving along at least one side of the square.
The simulation takes SIM_TIME seconds.
10 times during the simulation, the program should take a snapshot of the circle - namely, check which squares have cars on them; and then print the results like in the following example (for N=5):

```
**
 @@@*
*@@@*
 @@@
**  *
```

Where * denotes a car. The (N-1)x(N-1) squares in the middle of the traffic circles are denoted by @. A free square is denoted by a blank.
For relaxing the demands, the snapshot does not have to be consistent. Namely, cars may continue to progress (or try to progress) when another thread takes the snapshot.
Detailed requirements are in the next page.

Requirements

- Your system must use a mutex per each square.
- Cars should be able to progress simultaneously in different squares in the circle, as long as they don't use the same square.
- When a call to lock the mutex fails, you should check the returned value, for identifying whether it happened because the mutex is already locked, or because an error occurred. In the latter case, you should finish the simulation with an appropriate error message.
- Similarly, you should check the returned value when creating a new thread (pthread_create()).
- You should initialize all the mutexes at the beginning of the program, and destroy all of them when the program finishes from any reason.
- Minimize the sections of code which require mutual exclusion.
- The program should use a single process, with multiple threads.
- The program should avoid deadlocks.
- The program should print nothing beside of the 10 prints of the circle along the simulation. However, you may print blank lines before and/or after each print of the circle, so as to make the prints clearer.
- Note, that using extreme values for the parameters may yield strange results. Eg, if you generate new cars too frequently, the operating system may fail to allocate resources for opening new threads for them. Below are recommended values for the parameters:

  ```
  #define N 5
  #define FIN_PROB 0.1
  #define MIN_INTER_ARRIVAL_IN_NS 8000000
  #define MAX_INTER_ARRIVAL_IN_NS 9000000
  #define INTER_MOVES_IN_NS            100000
  #define SIM_TIME 2
  ```