Crossroad simulator

Summary

This projet aims to simulate four crossroads disposed in square. The purpose is to manage the vehicle stream to get a "city" working well, without any traffic light. The vehicle are thus interconnected with the road infrastructure, and everything must be synchronized.

The project must be implemented in C without any weird library, and it should make use of threads, shared memory, pipes, mutex (semaphores), etc…

The goal is to learn how to use those system, low-level functions, as asked by the course LO41 in UTBM.

Compiling and launching

Just copy the git repository and run `make`:

```
git clone https://git.libskia.so/skia/lo41.git
cd lo41
make
```

Then launch the program with

```
./crossroads
```

Enjoy watching vehicles on the road! :)

Data structures

The project has been made a bit like an object-oriented program, with some kinds of constructors, destructors, and data-structures related functions (a bit like methods).

Everything is made using pointers to avoid duplicating memory.

- Map: This is the top structure, containing all the others.
- Road: Basically a matrix of vehicle, plus the start and end crossroad.
- Crossroad: Four input roads and four output road, plus a buffer of vehicles.
- Vehicle: Its current road, its speed, its current lane and offset on the road.
- Fleet: A list of vehicle of the same type. There is one fleet per controller thread.

Critical sections The most critical sections are the roads: they are the only datas that are used by all the threads. That's why the map contains an array of mutex to lock each road independently, in order not to fall into an deadlock situation.

Another critical section, which is less critical, is the map itself, but it is locked only during the initialization phase of the threads, and is then not needed anymore until SIGINT is received, when everything needs to be freed.

Figure 1 in annex is a Petri net illustrating how the mutex on a critical section would work with four thread. The four mutex states represent always the same mutex protecting the critical section.

Implementation

There are seven threads, in addition to the first main process:

- Four for the different crossroads
- Three for the three different types of vehicle (mainly the speed changes for now)

Once the program is run, it will keep working until receiving a SIGINT (ˆC) signal. This is of course made using sigaction.

Everything is synchronized around the map structure, instanciated only one time, and the list of roads, containing the vehicles.

The thread routines

All the threads are launch in the main process, and have a function that loops forever until the `run` variable is set to 0. That happens when the main function catches the SIGINT signal, which calls then the ending function of the program after freeing everything.

The loop is based on a clock using `usleep()`. The threads don't need to be synchronized, since mutex are used for critical sections access.

Crossroads The main function of a crossroad is to pop the stacked vehicles at the end of its input road, store them into its buffer to simulate the time taken to cross it, then put the stored vehicles on the right road, looking at the vehicle's goal.

Choosing the right path The goal of a vehicle is implemented as an integer following the next conditions:

- goal/10 is the id of the last crossroad to be crossed
- goal%10 is the id of the last road before the exit

There are two cases:

- The vehicle is on its last crossroad and just needs to be send to the exit.

In that case, the direction is given by the following formula: D=(goal%10)%4 We get something between 0 and 3, and just define when building the map, that UP=1, DOWN=2, LEFT=3, RIGHT=0.

- The vehicle is not on its last crossroad: we just send it to the less loaded neighbor.

The used map is the following:

```
               UP: 1

               5    1
               |    |
          3 -- 0 -- 1 -- 4
LEFT: 3        |    |      RIGHT: 0
          7 -- 2 -- 3 -- 0
               |    |
               2    6

               DOWN: 2
```

Vehicle controllers The purpose of those three threads is to manage a fleet of vehicles. Every loop, all the roads are scanned, and the contained vehicles on a given road are stepped forward. Then the next road is scanned, and so on until all the roads have been checked. The whole operation is repeated every clock time.

Main loop The loop in the main process just perform the displaying of the map.

Printing the map

This has been made easily by allocating a chunk of memory, so basically a big matrix. Then some generic functions have been implemented to write a road, or a crossroad, in the given chunk, at the given coordinates, and for the given direction and orientation.

That way, a crossroad just prints itself and its adjacent roads by computing their coordinates with respect to their length, width, and position.

Finally, printing the map just requires printing the four crossroads at the right place.

Then the whole chunk is displayed in the console output and freed.

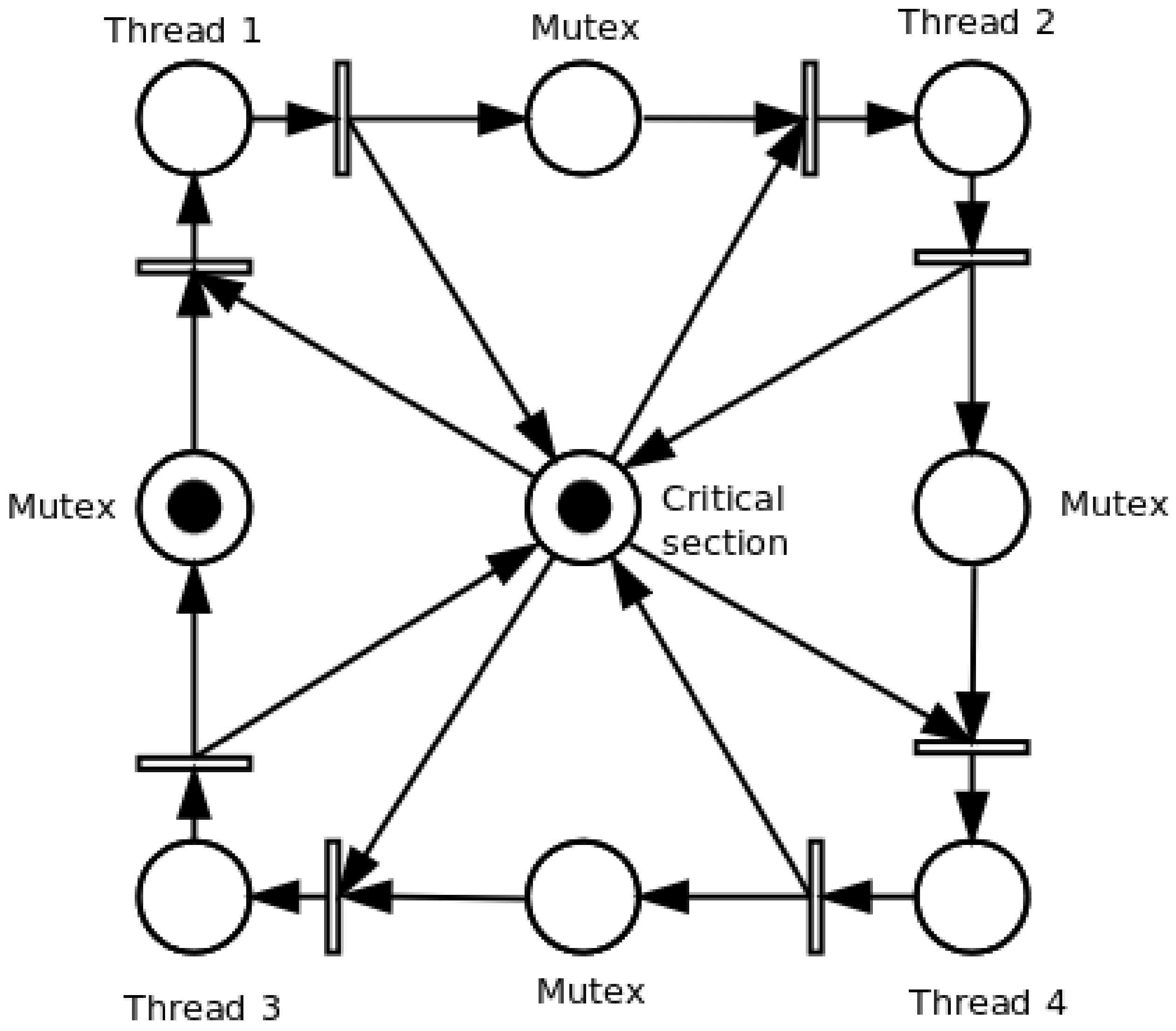

Figure 1: Petri net illustrating critical section