




Assignment 2: Threading

2INC0 - Operating systems

From

Full Name	Student ID
Daniel Tyukov	1819283
Nitin Singhal	1725963
Ben Lentschig	1824805

An aerial photograph of the TU/e campus in Eindhoven, Netherlands, taken at sunset. The image shows a large, modern glass-fronted building in the foreground, with other campus buildings and greenery visible in the background. The sky is a deep red and orange, reflecting on the glass surfaces of the buildings.

Eindhoven, January 14, 2025

1 | Basic solution

1.1 | Problem Introduction

Using threads a smart intersection is modeled as seen in the figure below:

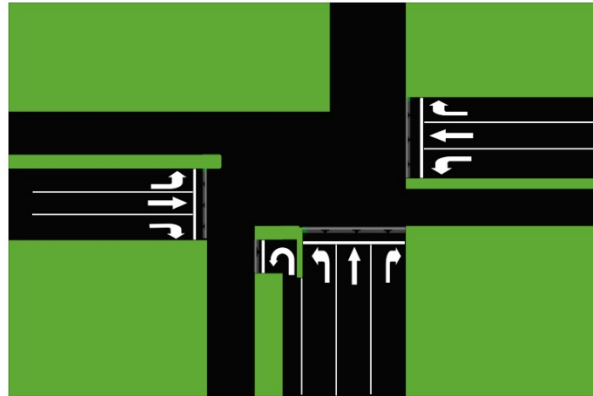


Figure 1.1: Car intersection

Each highway has several lanes each heading in a different direction (straight, right, left or uturn). Each lane has its own traffic light and needs to be managed such that if a car arrives on a lane, it wants to turn green and let this car pass. However, due to the amount of lanes and overlap of the intersection a method to manage the lights needs to be implemented.

1.2 | Solution scheme

The basic solution involves a main constraint that only one mutex is used for all 16 threads, meaning only one lane gets to turn green and their respective cars are crossing the intersection. To do so we can read the inputs of the function *supply_arrivals()* with one thread, which shares this memory with other threads acting as lanes which then can run *manage_light()*. As each lane thread is waiting for a semaphore, if one thread is allowed to change the light by running *manage_light()*, it will do so by locking a mutex, which is only unlocked after it has turned its light green and the car has passed the intersection. After completion all of the used memory which was previously allocated to the arrival of a car will be freed as memory is shared between threads. We pass the information to the manage light function via a structure with the side and direction of the car, as well as, a pointer to the mutex used. After all this is run, the main function will then wait for all threads to finish and terminate the code.

```

1 static void* manage_light(void* arg){
2     allocation_of_input_arguments();
3     while (get_time_passed() < END_TIME){
4         sem_wait(); // Wait for a car arrival
5         mutex_lock(); // Lock the intersection mutex
6         Get_arrival_info(); // Get the arrival information
7         printf("traffic light %d %d turns green at time %d for car %d\n", side,
8             direction, get_time_passed(), arrival.id);
9         sleep(CROSS_TIME);
10        printf("traffic light %d %d turns red at time %d\n", side, direction,
11            get_time_passed());
12        mutex_unlock(); // Unlock the intersection mutex
13    }
14    free(args);
15    return NULL;
16 }
17 typedef struct { // input of manage_light
18     Side side;
19     Direction direction;
20     pthread_mutex_t* intersection_mutex;
21 } LightArgs;

```

2 | Advanced solution

The implemented advanced solution uses 8 mutexes, 4 for blocking the exit lanes and 4 for "spatial blocking". The following images are used to model the paths of cars through the intersection. The 4 spatial mutexes are each associated with a cell on the 2x2 white square. When a path crosses a cell, it must lock the associated mutex. The exact positioning of the squares and the paths on the image is an assumption, but this is not a problem as long as the output is correct.

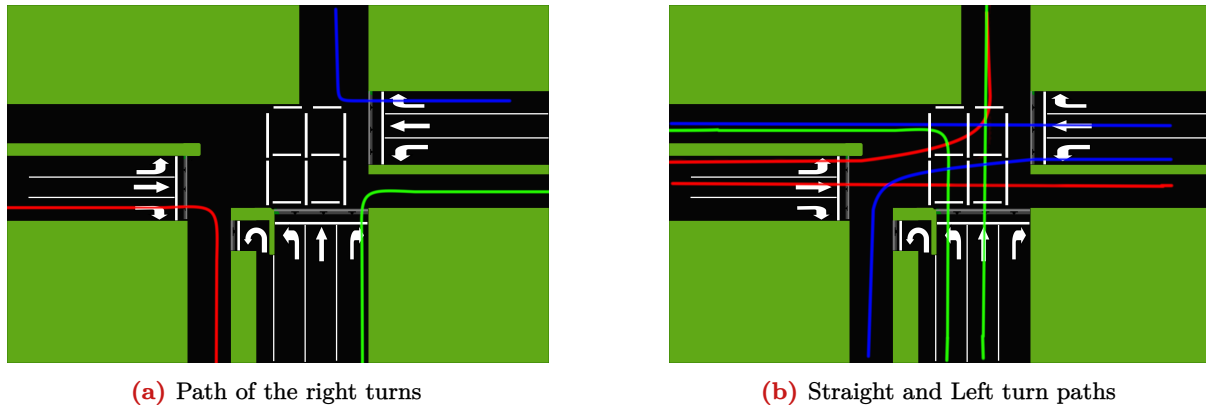


Figure 2.1: Paths for right, straight, and left turns.

From figure 2.1a, we can conclude that the right turns and u turns do not go through the middle of the intersection and only block their exit lanes. Straight turns from each side are modeled as blocking 2 of the 4 spatial mutexes. Left turns from each side block the other 2 spatial mutexes respectively. This is done so that cars from 1 side can go both straight and left at the same time.

Consider the south side. When going straight or left, 2 mutexes are blocked (both arranged vertically). The bottom locked mutex (bottom left or right) blocks straight turns from the west and left turns from the east. The top (right or left) mutex blocks left turns from the west and straights from the east. Blocking *both* straight and left turns from adjacent sides is why 2 spatial mutexes are blocked every time.

The following table summarizes the locking of the spatial mutexes for straight and left turns from the south and east. The blocking of the spatial mutexes is symmetric. For instance, left turns from the west side block the same mutexes as straight turns from the east side. This produces the effect that cars from the east and west side can both simultaneously go straight or go left, but if one goes left and the other straight, this will be an intersection (see figure 2.1). Square_0 and Square_1 store for each thread the mutexes they lock.

Turn side	Turn Direction	Square Mutex 0	Square Mutex 1
East	Straight	Top-left	Top-right
East	Left	Bottom-left	Bottom-right
South	Straight	Top-right	Bottom-right
South	Left	Top-left	Bottom-left

Table 2.1: Mutex assignments for different car paths.

DEADLOCK ANALYSIS: The mutexes are prevented from deadlocking because they are always called in a specific order. It can be seen from the `manage_light` function that mutexes are always blocked in the order `exit_lanes` \rightarrow `spatial_0` \rightarrow `spatial_1`.

```

1 pthread_mutex_lock(m_exit_lanes);
2   if((direction==0)|| (direction==1)){
3     pthread_mutex_lock(m_squares_0);
4     pthread_mutex_lock(m_squares_1);}

```

From the code that assigns the global mutexes to these local variables, it can be seen that mutexes with the smaller index are always stored in `spatial_0`.

```

1 //The selector variable's numerical value is unimportant. It only needs to
2 distinguish between cases.
3 switch(selector){
4     case 21:
5         m_squares_0 = args->m_squares[1];
6         m_squares_1 = args->m_squares[3];
7         break;
8     case 20:
9         m_squares_0 = args->m_squares[0];
10        m_squares_1 = args->m_squares[2];
11        break;
12    ...
13    case 30:
14        m_squares_0 = args->m_squares[0];
15        m_squares_1 = args->m_squares[1];
16        break;
17    }
    //Some cases are omitted for brevity

```

In addition to these mutexes, the only blocking calls made in the code are `pthread.join`. These are made either to the given supply arrival thread which terminates on its own, or after sending cancellation orders to traffic light threads. So there is no deadlock here as well.

EXPERIMENT: To test the speed of the advanced code, it was run against a packed intersection. Every second, a car arrives. 10 Cars arrive total, one for every lane. The output is

```

1   traffic light 1 0 turns green at time 0 for car 0
2   traffic light 1 1 turns green at time 1 for car 1
3   traffic light 1 2 turns green at time 2 for car 2
4   traffic light 1 0 turns red at time 5
5   traffic light 2 2 turns green at time 5 for car 5
6   traffic light 1 1 turns red at time 6
7   traffic light 2 0 turns green at time 6 for car 3
8   traffic light 2 3 turns green at time 6 for car 6
9   traffic light 1 2 turns red at time 7
10  traffic light 2 1 turns green at time 7 for car 4
11  traffic light 2 2 turns red at time 10
12  traffic light 2 0 turns red at time 11
13  traffic light 2 3 turns red at time 11
14  traffic light 3 2 turns green at time 11 for car 9
15  traffic light 2 1 turns red at time 12
16  traffic light 3 0 turns green at time 12 for car 7
17  traffic light 3 1 turns green at time 12 for car 8
18  traffic light 3 2 turns red at time 16
19  traffic light 3 0 turns red at time 17
20  traffic light 3 1 turns red at time 17

```

A total of 17 seconds are used. The last car took 9 seconds to arrive. The earliest any algorithm could have finished was $9 + \text{crossing time} = 14$ seconds. The basic code takes 50 seconds, 3 times as long.

```

1   traffic light 1 0 turns green at time 0 for car 0
2   traffic light 1 0 turns red at time 5
3   traffic light 1 1 turns green at time 5 for car 1
4   traffic light 1 1 turns red at time 10
5   traffic light 1 2 turns green at time 10 for car 2
6   traffic light 1 2 turns red at time 15
7   traffic light 2 0 turns green at time 15 for car 3
8   traffic light 2 0 turns red at time 20
9   traffic light 2 1 turns green at time 20 for car 4
10  traffic light 2 1 turns red at time 25
11  traffic light 2 2 turns green at time 25 for car 5
12  traffic light 2 2 turns red at time 30
13  traffic light 2 3 turns green at time 30 for car 6
14  traffic light 2 3 turns red at time 35
15  traffic light 3 0 turns green at time 35 for car 7
16  traffic light 3 0 turns red at time 40
17  traffic light 3 1 turns green at time 40 for car 8
18  traffic light 3 1 turns red at time 45
19  traffic light 3 2 turns green at time 45 for car 9
20  traffic light 3 2 turns red at time 50

```