Lab 4: Synchronizing Threads

Master M1 MOSIG – Université Grenoble Alpes (Ensimag & UFR IM2AG)

2025

In this lab, we are going to study of few synchronization problems.

1 Instructions

This lab is not graded. The problems are to be solved on paper.

2 The Peruvian and Bolivian trains (R.C. Holt et al.)

Start thinking like a concurrent programmer![1]

High in the Andes Mountains, there are two circular railroad lines. As shown in Figure 1, one line is in Peru, the other in Bolivia. They share a section of track, where the lines cross a mountain pass that lies on the international border (from A to B).

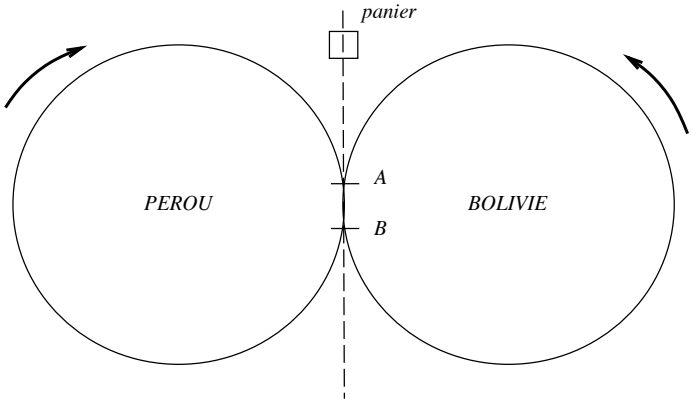

Figure 1: Somewhere in South-America

Unfortunately, the Peruvian and Bolivian trains occasionally collide when simultaneously entering the critical section of track (the mountain pass). The trouble is, alas, that the drivers of the two trains are blind and deaf, so they can neither see nor hear each other.

[1] This exercise is adapted from the famous exercise by R.C. Holt et al.

The two drivers agreed on the following method of preventing collision. They set up a large bowl at the entrance to the pass. Before entering the pass, a driver must stop his train, walk over to the bowl, and reach into it to see if it contains a pebble. If the bowl is empty, the driver finds a pebble and drops it in the bowl, indicating that his train is entering the pass; once his train has cleared the pass, he must walk back to the bowl and remove his pebble, indicating that the pass is no longer being used. Finally he walks back to the train and continues down the line. If a driver arriving at the pass finds a pebble in the bowl, he leaves the pebble there; he repeatedly takes a siesta and re-checks the bowl until he finds it empty. Then he drops a pebble in the bowl and drives his train into the pass.

Question 2.1: *A smart aleck colleague (graduate from the Mosig Master in Grenoble) claimed that subversive train schedules made up by Peruvian officials could block the Bolivian train forever. Explain.*

Question 2.2: *As you may have already understood, this exercise is a analogy where train drivers represent the threads, the bowl represents a variable in shared memory, and the shared section of track represents a critical section. Express the algorithm we just studied using pseudo C code.*

Question 2.3: *The problem described in Question 2.1 was never observed in practice. Explain.*

Question 2.4: *What is the synchronization problem that is illustrated by this story?*

Question 2.5: *Which properties are not guarantied by the proposed algorithm based on what was discussed in Question 2.1?*

Question 2.6: *The proposed algorithm also does not ensure safety. Demonstrate this point with an execution scenario.*

To try solving the problems of the previous algorithm, we propose the algorithm described in Figure 2. The algorithm is proposed for the case of 2 threads, T_0 and T_1. The intuition is to use the boolean $busy$ differently for T_0 and T_1.

Question 2.7: *What can you say about the properties of this new solution?*

We propose yet another algorithm, presented in Figure 3. In this new version, instead of a single variable $busy$, we use an array of two booleans $wants$ that allows each thread to announce when it wants to enter the CS. A thread can enter only if the other thread does not want to enter. If both want to enter, they give up for some amount of time and then try again.

Question 2.8: *This new method works well. But we observe that sometimes, both threads get blocked outside the CS for some large amount of time. Explain what happens.*

```
int busy=0;
int my_id; /* 0 or 1 */

void enter_CS(void)
{
    if (my_id == 0){
       while(busy == 1){ sleep(rand()%10); }
    }
    else{
       while(busy == 0){ sleep(rand()%10); }
    }
}

void leave_CS(void)
{
    if (my_id == 0){
       busy = 1;
    }
    else{
       busy = 0;
    }
}
```

Figure 2: A second algorithm

```
int wants[2]={0,0};
int my_id; /* 0 or 1 */

void enter_CS(void)
{
   wants[my_id] = 1;
   while(wants[1 - my_id] == 1){
      wants[my_id] = 0;
      sleep(rand()%10);
      wants[my_id] = 1;
   }
}

void leave_CS(void)
{
   wants[my_id] = 0;
}
```

Figure 3: A third algorithm

3 Implementing a barrier with semaphores

A barrier is a famous synchronization primitive. According to Wikipedia, a barrier for a group of threads in the source code means any thread must stop at this point and cannot proceed until all other threads reach this barrier.

Let us consider two threads A and B, and four instructions IA_1, IA_2, IB_1 and IB_2, such that we have the following code:

THREAD A	THREAD B
IA_1	IB_1
A_Sync	**B_Sync**
IA_2	IB_2

The synchronization should implement a barrier: IA_2 is executed after IB_1 has been executed; IB_2 is executed after IA_1 has been executed.

Question 3.9: *Design **A_Sync** and **B_Sync** using semaphores.*

4 Semaphores vs Condition Variables

We saw during the class that the original version of the semaphore algorithm was defined with two operations P and V (that correspond to `sem_wait` and `sem_post` respectively).

Question 4.10: *To show that mutexes plus condition variables are as powerful as semaphores, implement P and V operations using condition variables.*

Note: In addition to P and V, you should obviously introduce a `sem_init()` function.

5 Semaphores vs Condition Variables (2nd round)

We are implementing an application that takes actions based on mouse clicks. For this purpose, we have a dedicated thread that monitors mouse clicks and informs the other threads when a click occurs. To synchronize these threads, we would like to encapsulate the synchronization code into two functions called `new_click_event()` and `get_new_click()`. These two functions have the following semantics:

- `void new_click_event(void)`: Informs that a new click has been detected.
- `void get_new_click(void)`: Returns only when a click event is *available*. By *available*, we mean that each click can be associated with a single call to `get_new_click()`. We also mean that no event is lost, that is, if there is a new click event while no thread is blocked in the `get_new_click()` function, then the next thread calling `get_new_click()` will not block.

To better illustrate the semantics of these two functions, we provide two examples of execution in Figure 4 and 5.

0:	Thread T1 calls `get_new_click()`	T1 is blocked
1:	Thread T0 calls `new_click_event()`	T1 is unblocked
2:	Thread T0 calls `new_click_event()`	
3:	Thread T2 calls `get_new_click()`	T2 is not blocked

Figure 4: Execution scenario

0:	Thread T0 calls `new_click_event()`	
1:	Thread T0 calls `new_click_event()`	
2:	Thread T1 calls `get_new_click()`	T1 is not blocked
3:	Thread T2 calls `get_new_click()`	T2 is not blocked
4:	Thread T3 calls `get_new_click()`	T3 is blocked

Figure 5: Another execution scenario

Question 5.11: *Provide the code that implements the two functions using semaphores.*

Question 5.12: *Provide the code that implements the two functions using any synchronization mechanisms seen during the course, except semaphores.*

Note 1: In both cases, you have to avoid any kind of busy waiting.

Note 2: You are allowed to introduce any global variables you want. If need be, you can also introduce an `init()` function that would be called once at the very beginning of the program.