# Week 1 Assignments

## A Critical Section

There are two threads, with the code listed below. The programmer did his best to achieve mutual exclusion, fairness, and to avoid deadlock. However, m*utual exclusion* is not guaranteed.

Answer the following questions:

1. how can both threads enter the Critical Section at the same time?
2. can *deadlock* occur? (why/why not?)
3. is this implementation *fair* (i.e. is it *starvation*-free)?

For all three questions: give a precise description how it happens. Make a table where you write on each line the executed statement (like A1 or B7), together with the actual value of the variables (flag[0], flag[1], lock[0], lock[1]) after execution of that statement.

bool flag[2] = { false, false };   
bool lock[2] = { false, false };

***thread A: thread B:***

while (true) while (true)

{ {

1. flag[0] = true; flag[1] = true;
2. lock[1] = false; lock[0] = false;
3. if (flag[1] == true) if (flag[0] == true)

{ {

1. lock[1] = true; lock[0] = true;
2. flag[0] = false; flag[1] = false;

} }

1. while (lock[1] || flag[1]) while (lock[0] || flag[0])

{ {

1. flag[0] = false; flag[1] = false;
2. flag[0] = true; flag[1] = true;

} }

1. CriticalSection(); CriticalSection();
2. flag[0] = false; flag[1] = false;
3. lock[0] = false; lock[1] = false;

} }

## Interleaving

Given the following statements:

x = 0  
def myThread():  
 global x  
 for i in range(100):  
 x += 1

myThread is started two times. They both execute the for-loop such that x will be incremented.

The operation x += 1 is not atomic; in assembler code it could be something like:

for one thread:

load R1, @x  
 inc R1  
 store R1, @x

for the other thread:

load S1, @x  
 inc S1  
 store S1, @x

(R1 and S1 are registers of the CPU)

Because those instructions are not secured with semaphores, strange situations can happen with the contents of x. If everything runs sequentially in a proper way, then we expect that x has afterwards a value of 200. A larger value than 200 is not expected.

The assignment:

* It appears that there is a scenario that x is 2 at the end of the process. Design this scenario (watch out: this requires a creative brain!!!!).
* If you cannot find such a scenario, what's the lowest value that you have discovered? (200?, 101?, 100?, 1?, …?)
* Describe how the threads are interleaving their statements to reach that value of x.

## Synchronization

Create and run 4 threads A, B, C and D.

They print the numbers 1 until 8 on one terminal. Thread A prints the number 1 and 5, thread B prints 2 and 6, thread C prints 3 and 7, thread D prints 4 and 8.

Requirements:

* the semaphores may be created before the threads are started
* the numbers are printed in the "right order"
* you may only use semaphores for synchronization (so no busy-wait loops, no shared memory)
* it should not make any difference in which order the threads are started

## Deadlock

Create three threads and three semaphores. Write synchronization code with the risk of deadlock, but where they also may run for hours without problems.

Implement in the simulator and demo the deadlock and the smooth operation.

The simulator can be found at: <https://git.fhict.nl/I878848/SyncSimulator> .