# 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;

} }

**Answer to question A:**

In a scenario where the ENTIRE struct value gets taken from a register (that is, both booleans are taken from the registers at the same time, even when only 1 gets updated, and both get rewritten after 1 or 2 gets updated), a scenario exists where thread A enters line 1, holds the value for flag and then gets halted. Thread B then executes up until line 9 after which it gets halted by the schedular. Then, thread A kicks back into action and overwrites the flag register. This allows thread A to enter the critical section at the same time as thread B.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Thread** | **Line** | **Flag 0** | **Flag 1** | **Lock 0** | **Lock 1** |  | **Flag 0 OR Lock 0** | **Flag 1 OR Lock 1** |
| Both | Starting | 0 | 0 | 0 | 0 |  | FALSE | FALSE |
| A | 1 | 0 | 0 | 0 | 0 |  | FALSE | FALSE |
| B | 1 | 0 | 1 | 0 | 0 |  | FALSE | TRUE |
| B | 2 | 0 | 1 | 0 | 0 |  | FALSE | TRUE |
| B | 3 | 0 | 1 | 0 | 0 |  | FALSE | TRUE |
| B | 6 | 0 | 1 | 0 | 0 |  | FALSE | TRUE |
| B | 9 | 0 | 1 | 0 | 0 |  | FALSE | TRUE |
| A | 1 | 1 | 0 | 0 | 0 |  | TRUE | FALSE |
| A | 2 | 1 | 0 | 0 | 0 |  | TRUE | FALSE |
| A | 3 | 1 | 0 | 0 | 0 |  | TRUE | FALSE |
| A | 6 | 1 | 0 | 0 | 0 |  | TRUE | FALSE |
| A | 9 | 1 | 0 | 0 | 0 |  | TRUE | FALSE |

**Answer to question C:**

In a perfect world where the locking mechanism works as intended, I would say it is. Simply because after one thread exists the critical section the other thread will then exit the while loop after some time.

This again (as in the other 2 examples) is entirely dependent on whether the schedular will provide the threads with their required time on the CPU. In a SJF schedular, depending on whether the schedular determines that the 2 threads will take longer than all other jobs that get queued for execution, they will starve regardless. But that’s outside of the programmer’s control.

**Answer to question B:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Thread** | **Line** | **Flag 0** | **Flag 1** | **Lock 0** | **Lock 1** |  | **Flag 0 OR Lock 0** | **Flag 1 OR Lock 1** |
| Both | Starting | 0 | 0 | 0 | 0 |  | FALSE | FALSE |
| A | 1 | 1 | 0 | 0 | 0 |  | TRUE | FALSE |
| B | 1 | 1 | 1 | 0 | 0 |  | TRUE | TRUE |
| A | 2 | 1 | 1 | 0 | 0 |  | TRUE | TRUE |
| B | 2 | 1 | 1 | 0 | 0 |  | TRUE | TRUE |
| A | 3 | 1 | 1 | 0 | 0 |  | TRUE | TRUE |
| B | 3 | 1 | 1 | 0 | 0 |  | TRUE | TRUE |
| A | 4 | 1 | 1 | 0 | 1 |  | TRUE | TRUE |
| B | 4 | 1 | 1 | 1 | 1 |  | TRUE | TRUE |
| A | 5 | 0 | 1 | 0 | 1 |  | FALSE | TRUE |
| B | 5 | 0 | 0 | 1 | 1 |  | TRUE | TRUE |
| A | 6 | 0 | 0 | 1 | 1 |  | TRUE | TRUE |
| B | 6 | 0 | 0 | 1 | 1 |  | TRUE | TRUE |
| A | 7 | 0 | 0 | 1 | 1 |  | TRUE | TRUE |
| B | 7 | 0 | 0 | 1 | 1 |  | TRUE | TRUE |
| A | 8 | 1 | 0 | 1 | 1 |  | TRUE | TRUE |
| B | 8 | 1 | 1 | 1 | 1 |  | TRUE | TRUE |
| At this point, both threads re-enter the while loop. Since the locks are never updated, they will stay inside this loop forever, that is, if the schedular makes it so the code gets a "per-line fair execution time" approach. As such, one could state a deadlock has occurred. I say "could", cause it’s my understanding that a deadlock occurs when 2 processes enter a waiting state and then wait for the other to signal, they can continue. The threads in the scenario above are technically not in a waiting state, they are still running, just indefinitely. But that’s a technicality... | | | | | | | | |

## 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.

**Question: “It appears that there is a scenario that x is 2 at the end of the process. Design this scenario”**

After a lot of thought (again, since this is the resist, and previously I was also not able to picture a scenario where x would equal 2) I was, again, unable to find such a scenario. I did find one for all the following numbers via sketching out interleaving patterns: 200, 101, 100.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value for i** | **Thread** | **Line** | **X in reg.** | **X in mem.** |
| … | Both | 1 trough 5 | n/a | 0 |
| 0 | A | load | 0 | 0 |
| 0 | A | inc | 1 | 0 |
| 0 | B | load | 0 | 0 |
| 0 | B | inc | 1 | 0 |
| 0 | B | store | 1 | 1 |
| 1 | B | load | 1 | 1 |
| 1 | B | inc | 2 | 1 |
| 1 | B | store | 2 | 2 |
|  |  |  |  |  |
| 0 | A | store | 1 | 1 |
| 1 | A | load | 1 | 1 |
| 1 | A | inc | 2 | 1 |
|  |  |  |  |  |
| 2 | B | load | 1 | 1 |
| 2 | B | inc | 2 | 1 |
| 2 | B | store | 2 | 2 |
| Continue the pattern above to get 100, as the threads keep overwriting each other, meaning only 1 thread effectingly counts to 100 | | | | |
|  |
|  |
| Change up the first few itterations to make 1 thread count up to 1, and then use the same previously mentioned interleaving pattern to get 101 | | | | |  |
|  |
|  |

## 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> .