Sync Week01

Alexandre Flion - Arthur Adam

2023-04-01

Contents

1 Critical Section

1.1 Mutual exclusion

In this situation, both threads can enter the critical section at the same time. Let's expose our thoughts:

Since it is possible for a thread to execute mutliple statements in row the other is waiting. We can imagine that one of the thread use that kind of mechanism.

We need threadB to arrive at the critical section and access it first by executing 5 statements (B1-B2-B3-B6-B9).

After that, threadA must begin and must use almost the same path, executing 5 statements in a row (A1-A2-A3-A4-A5).

When threadA has executed statement A5, threadB executes statements B10 and B11.

ThreadA will just need to execute statement A6 and to "wait" for threadB to arrive.

ThreadB will then execute the same path of statement: B1-B2-B3-B6.

Line	flag[0]	flag[1]	lock[0]	lock[1]	Comment
B1		T			
B2			F		
B3					Condition unsatisfied
B6					Condition unsatisfied
B9					Critical Section
A1	T				
A2				F	
A3					Condition satisfied
A4				T	
A5	F				
B10		F			
B11				F	
A6					Condition unsatisfied
B1		T			
B2			F		
B3					Condition unsatisfied
B6					Condition unsatisfied
A9					Critical Section
B9					Critical Section

Thus, threadA and threadB will arrive at the same time at the critical section.

1.2 Deadlock

The base state is all variables at false. Here is the table describing how to get to a deadlock in the given code.

Line	flag[0]	flag[1]	lock[0]	lock[1]	Comment
A1	T				
B1		T			
A2					
B2					again
A3					Condition verified
B3					Condition verified
A4				T	
B4			T		
A5	F				
B5		F			
A6					Condition verified, lock[1] = T
B6					Condition verified, lock[0] = T

At this point the program has reached a deadlock since neither locks can be set to $false$ and both loop require one of the locks to be set to $false$.

1.3 Fair implementation

A starvation is, for a given thread, a return to the initial state of his data at a time X, after this thread went through the critical section.

Let's try to reproduce this behaviour by moving one thread only:

Line	flag[0]	flag[1]	lock[0]	lock[1]	Comment
B1		T			
B2			F		
B3					Condition unsatisfied
B6					Condition unsatisfied
B9					Critical Section
B10		F			
B11				F	

We can see that every variable (flag[0], flag[1], lock[0], lock[1]) of the program has returned to their initial state "False".

Thus, we can say that this implementation is not starvation-free.

2 Interleaving

Here is the list of steps to recreate in order to get $x = 2$:

- First thread loads x in R1
- Second thread does 99 loops, gets to $x = 99$
- First thread increments R1 and stores it into x, now $x = 1$
- Second thread loads x in S1, $S1 = 1$

- First thread finishes its loops, $x = 101$
- Second thread increments S1 and stores it into x, now $x = 2$

Needless to say, this scenario is highly improbable and would most likely never happen.

3 Synchronization

Refer to file week1_synchronization.py

4 Deadlock

Refer to file week1_deadlock.py

I could not get the simulator to work with my code, but it shouldn't matter as the deadlock is completely random and cannot be replicated using the simulator.

If you want to test the deadlock, just change the line with the randint to a value much lower, and it will be more probable for it to happen.