Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Solution 8: Master Solution
Spring Semester 2017

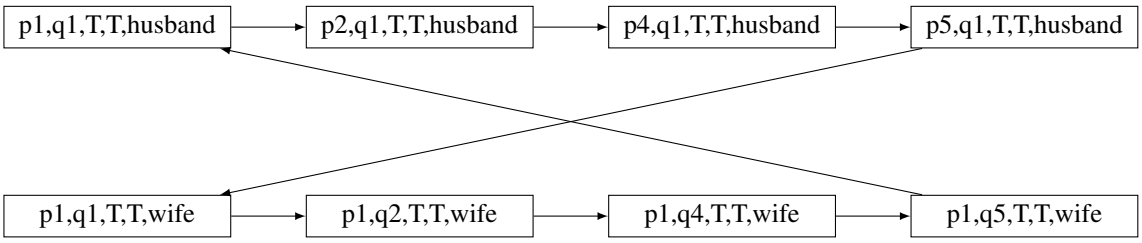
Analyzing locks

a) After analyzing the code we see that:

Table 1: Livelock state diagram

owner			
husband.hungry = true			
wife.hungry = true			
husband		wife	
p1:	while hungry	q1:	while hungry
p2:	owner != me	q2:	owner != me
p3:	sleep	q3:	sleep
p4:	spouse == hungry	q4:	spouse == hungry
p5:	owner = spouse	q5:	owner = spouse
p6:	CR	q6:	CR
p7:	hungry = false	q7:	hungry = false
p8:	owner = spouse	q8:	owner = spouse

Then, we do the state diagram for the livelock.



(state diagram only illustrates the minimal set of states involved in the livelook, other states exist)

And we can see that both threads will be blocking each other every time. This is indeed a livelock.

Even though the implementation does not allow more than one thread to access the shared resource "spoon", this implementation of the mutual exclusion protocol is incorrect for the following reason: we require from a mutual exclusion implementation that it guarantees progress for at least one thread that wants to enter the critical section provided that the critical section is free. This is clearly violated here.

b) One way to solve the livelock problem is to impose an ordering when acquiring the lock on the shared resource. In this way, we can assure how threads will access this. In our specific problem, we could make one of the spouses to be not so polite, and actually take the spoon after certain number of retries e.g. we could make the wife always to take it first.

Optimistic vs Pessimistic concurrency control

a) The CAS operation can be used to enable optimistic concurrency. Steps 1-2 in the above algorithm are done without acquiring a lock. Step 3 uses a CAS operation to store the updated next seed value only if no other thread has changed it. If the CAS succeeds, then step 4 is executed and the operation is complete. If the CAS fails because another thread updated the seed, then the operation goes back to step 1 and tries again.

b) When more threads are used, the AtomicRandom generation becomes more and more expensive. This is because there exists a very high contention for the seed which makes many of the threads retry the atomic operation possibly many times. Retrying the atomic operation is a very expensive instruction which leads to poor performance of such pseudorandom generation method. A way to improve this could be to add a back-off mechanism in case of the atomic operation failure in order to avoid unnecessary retries.

c) Optimistic concurrency control could be useful whenever there is low data contention, thus we assume that although there will be conflicts, they will be rare. We will look for indication if two threads actually tried to update the shared resource at the same time. If this is the case, then one of the threads' operation will be discarded and retried i.e. performing an extra atomic operation.