Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 8: Master Solution
Spring Semester 2020

Analyzing locks

a) After analyzing the code we can simplify it and rewrite it (in the notation from the lectures) in the following way:

Table 1: Livelock state diagram

Global variables:	
husband.isHungry = T	
wife.isHungry = T	
spoon.owner = husband	
husband	wife
Local variables:	Local variables:
spouse = wife	**spouse = husband**
p1: while (isHungry)	q1: while (isHungry)
p2: if (spoon.owner != this)	q2: if (spoon.owner != this)
p3: sleep	q3: sleep
p4: else if (spouse.isHungry)	q4: else if (spouse.isHungry)
p5: spoon.owner = spouse	q5: spoon.owner = spouse
p6: else CR	q6: else CR
p7: isHungry = false	q7: isHungry = false
p8: spoon.owner = spouse	q8: spoon.owner = spouse

Diner object **owner** in the object **spoon** and volatile variables **isHungry** from objects **husband** and **wife** are global variables. Parameter **spouse** is a local variable per process.

Program state has the following structure: $((PC_h, PC_w), (LV_h, LV_w), GV)$ where PC_i, and LV_i, are program counter and a tuple of current values of local variables of process i. GV is a tuple of current values of the global variables. Since in this example the local variables do not change, we omit them from the states for conciseness. In fact **spouse** variable is only used to make code of the two processes identical. Here follows the state diagram for the simplified algorithm above:

$$\begin{tikzcd}
\boxed{((p1,q1),(T,T,husband))} \arrow[r] & \boxed{((p2,q1),(T,T,husband))} \arrow[r] & \boxed{((p4,q1),(T,T,husband))} \arrow[r] & \boxed{((p5,q1),(T,T,husband))} \arrow[dlll] \\
\boxed{((p1,q1),(T,T,wife))} \arrow[r] & \boxed{((p1,q2),(T,T,wife))} \arrow[r] & \boxed{((p1,q4),(T,T,wife))} \arrow[r] & \boxed{((p1,q5),(T,T,wife))} \arrow[ulll]
\end{tikzcd}$$

(the state diagram only illustrates the minimal set of states involved in the livelock, 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 give priority to the wife.

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 state 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 state, 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 state 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.