Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 9: Master Solution
Spring Semester 2020

Dining Philosophers

a) There is a possibility to deadlock: In the first step, each philosopher acquires the fork on their left side at the same time. Now that every philosopher has one fork in their left hand, they will try to pick up the fork on their right side, but this fork is already held by their neighbor. In this state, everyone waits for their neighbor on the right to release the fork, but since nobody has acquired both forks to eat, this will never happen and the philosophers will starve.

The problem is that there is a cyclic dependency in the case described above: P1 waits for P5, P5 waits for P4, P4 waits for P3, P3 waits for P2, and P2 waits for P1.

b) The correct solution is to break the cyclic dependency. This means that one philosopher, e.g. philosopher P5 has to pick up the forks in a different order. This could be achieved by introducing a order between the forks, and all the philosophers have to pick up the fork with the lower order first. This resolves the deadlock, because there are no cyclic dependencies any more.

Note that introducing a timeout will not solve the problem: Even if the philosophers would release the fork after some time of waiting, it could still happen that they would do this at the same time. This would be a livelock, where all philosophers would constantly acquire and release the fork on their left, but nobody would get to eat.

c) As there are only five forks and every philosopher needs two forks to eat, the maximum amount of philosophers eating concurrently is two, however there is a possibility that only one philosopher is eating. Consider P2, P3, P4 each acquiring the fork on their left und P1 acquires both forks. Nobody else, except P1 can now eat.

All solutions to fix that problem require the philosophers to communicate in some way, i.e., introducing a single lock for a pair of forks, or forming a queue of waiting philosophers and a central arbiter (”waiter”) which selects which philosopher can eat next.

Better than Dijkstra

This is the proposed lock:

```
C0: b(i) := false;
C1: if k != i then begin
C2: if !b(j) then go to C2;
    else k := i; go to C1; end;
    else CS;
    b(i) := true
```

Let us first rewrite this in some more Java-like pseudocode, first we add some indention.

```
C0: b(i) := false;
C1: if k != i then
    begin
C2:     if !b(j) then go to C2;
        else k := i; go to C1;
    end;
    else CS;
    b(i) := true
```

Now we translate go to statements into while loops and renumber the statements.

```
S1:    b(i) := false;
S2:    while (k != i) {
S3:        while (!b(j)) {};
S4:        k := i;
       }
S5:    // CS
S6:    b(i) := true
```

Note that not all variables are initialized properly. Let us assume k=0 and b=[true,true].

We have two ways to solve this problem. The lazy and smart one is to assume both threads are in the critical section, then proof that this can or cannot happen by "tracing back" from there.

Another option is to draw the full state-space diagram. This is a lot of work, the upper bound on the number of states we need to consider is exponential in the number of bits of storage and threads, in this case we could have up to $6^2 * 2^3$ states to examine. Note that there are tools to facilitate this approach[1].

Let's do the first approach. We assume both treads P0 and P1 are in the critical section (mutual exclusion is violated). Lets also assume P0 enters first (they both execute the same code, so this assumption is without loss of generality). Lets name our threads P_0 and P_1. For P_0 $i = 0$ and $j = 1$, for P_1 $i = 1$ and $j = 0$.

So we can write the actions of P_0 that lead to P_0 to be in the CS like this:

$$W_{P_0}(b[0] = false) \rightarrow R_{P_0}(k = 0) \rightarrow CS_{P_0}$$

Now following our assumption that P_1 enters the CS second, we assume it goes once through the while loop before entering the CS:

$$W_{P_1}(b[1] = false) \rightarrow R_{P_1}(k = 0) \rightarrow R_{P_1}(b[0] = 1) \rightarrow W_{P_1}(k = 1) \rightarrow R_{P_1}(k = 1) \rightarrow CR_{P_1}$$

[1] see for example the spin model checker, available at http://spinroot.com/

Now we can check if we can find a valid interleaving for both of those traces, i.e., an arrangement such that all “happens before“ arrows are pointing from left to rigt and all reads return the value last written to the respective variable:

$W_{P_1}(b[1] = false) \rightarrow R_{P_1}(k = 0) \rightarrow R_{P_1}(b[0] = true) \rightarrow W_{P_0}(b[0] = false) \rightarrow$
$R_{P_0}(k = 0) \rightarrow CS_{P_0} \rightarrow W_{P_1}(k = 1) \rightarrow R_{P_1}(k = 1) \rightarrow CR_{P_1}$

This counterexample shows that the lock does not provide mutual exclusion.

Transitive Closure

If a relation R contains (a, b) and (b, c) then the transitive closure is the smallest relation that contains (a, c). If we apply this to our example, the transitive closure gives us the relation "reachable" (directly or indirectly).

From / To	Aachen	Bern	Chemnitz	Dresden	Erfurt	Frankfurt	St. Gallen	Hamburg
Aachen	X	X	X	X			X	
Bern	X	X	X	X			X	
Chemnitz	X	X	X	X			X	
Dresden	X	X	X	X			X	
Erfurt					X			
Frankfurt						X		
St. Gallen	X	X	X	X			X	
Hamburg	X	X	X	X			X	X

Program Order

Let us first define a relation “is executed before“:

S1, S2
S2, S3
S3, S4
S3, S5

Program order is now the transitive closure of those operations.

S1, S2
S2, S3
S1, S3
S3, S4
S1, S4
S2, S4
S3, S5
S1, S5
S2, S5

Synchronization Actions

Reading and writing of volatile variables is a synchronization action, same as locking or unlocking, and starting or stopping a thread, or any action that checks if a thread has terminated. Nothing else.