Chalmers | Göteborgs Universitet

Concurrent Programming TDA383/DIT390

Tuesday, 19 December 2017

Teacher/examiner: K. V. S. Prasad (prasad@chalmers.se, 0736 30 28 22)

Answer sheet

Question 1. **(Part a)**. p1,q1,q2,q3,p2,p3,q4,p1,q1. *(2p)*

(Part b). p1,p2,p3,(p1,p2,p3,q1,q2,q3)* *(3p)*

(Part c). p1,q1,q2,q3,q4,p2,p3* — Also correct solution for b) *(3p)*

Question 2. **(Parts a through c)**. *(12p)*

```
\begin{verbatim}
import java.util.concurrent.Semaphore;

class BuildingFirm {

    // Constant
    final int NUM_SPECIALISTS = 2;

    // Semaphore definitions
    Semaphore start = new Semaphore(0);
    Semaphore done = new Semaphore(0);

    class TeamManager extends Thread {
        public void run() {
            while(true) {
                for (int i = 0; i<NUM_SPECIALISTS; i++) {
                    start.release();
                }
                for (int i = 0; i<NUM_SPECIALISTS; i++) {
                    done.acquire();
                }
            }
        }
    }

    class Worker extends Thread {
        public void run() {
            while(true) {
                start.acquire();
                // Do some work
                done.release();
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i<NUM_SPECIALISTS; i++) {
            new Worker().start();
        }
        new TeamManager().start();
    }
}
```

(Part d). The simplest way to do this with binary semaphores is

to declare arrays of binary semaphores `start` and `done`, one array element per worker. When a house is completed, the manager signals all the `start`s, and then waits for each `done` semaphore in the array order (even if the workers don't finish in that order).

(3p)

Question 3. (Part a) *(5p)*

	State = (pi, qi, Svalue)	next state if p moves	next state if q moves
s1	(p2, q2, 0)	(p3, q2, 2)=s2	(p2, q3, 5)=s3
s2	(p3, q2, 2)	(p5, q2, 2)=s4	(p3, q3, 7)=s5
s3	(p2, q3, 5)	(p3, q3, 3)=s6	(p2, q5, 5)=s7
s4	(p5, q2, 2)	(p2, q2, 0)=s1	(p5, q3, 7)=s8
s5	(p3, q3, 7)	(p5, q3, 7)=s8	no move
s6	(p3, q3, 3)	no move	(p3, q5, 3)=s9
s7	(p2, q5, 5)	(p3, q5, 3)=s9	(p2, q2, 4)=s10
s8	(p5, q3, 7)	(p2, q3, 5)=s3	no move
s9	(p3, q5, 3)	no move	(p3, q2, 2)=s3
s10	(p2, q2, 4)	(p3, q2, 2)=s2	(p2, q3, 5)=s3

(Part b) There is no state with (p5, q5, sn). *(2p)*
(Part c) There is no state where neither p nor q has a move. *(2p)*

Question 4. **(Part a)**. To get to $(p3 \wedge q3)$, assume p got to p3 (so S = 2 or 3) and is waiting when q executes q2, giving S=7. If q got to q3 first, S=5 or 7, and then after p2, S=3. *(4p)*

(Part b) Immediate given the assumption. After q5, S=2 and p can get past p3. *(2p)*

(Part c) Again immediate, given the assumption. If p3 executes, which it must if the scheduler is fair, then p gets to p5. Points for knowing what box and diamond and fairness mean. *(3p)*

Question 5. (Part a). *(2p)*

```
init_graders(0) -> ok ;
init_graders(N) ->
  spawn(fun() -> grader() end),
  init_graders(N-1).
```

(Part b). *(8p)*

```
grader() ->
  examiner ! {idle, self()},
  receive
    finished -> ok ;
    {grade, Exam} ->
      examiner ! {ready, grade(Exam)},
      grader()
  end.
```

(Part c). The message passing is synchronous. The examiner is blocking while the grader is blocking. Only one worker is working at any given time. *(2p)*

(Part d). Improvement: the communication is now asynchronous and multiple workers can run simultaneously because the examiner doesn't block. [1 point]

Problem: the same ungraded exam could be given to multiple graders, resulting in duplication of work, since the examiner does not keep track of which exams are pending. (Although if you assume `get_ungraded_exam` does take this into account, then that should be ok). [2 points] *(3p)*

Question 6. (Part a). *(10p)*

```
1 class Table {
2   Lock lock = new ReentrantLock();
3   Condition wait = lock.newCondition();
4
5   int cookies;
6
7   Table (int c) { this.cookies = c;
8   }
9
10  int howManyLeft() {
11    try {
12      lock.lock();
13      return cookies;
14    } finally {
15      lock.unlock();
16    }
17  }
18
19  void get() throws InterruptedException {
20    try {
21      lock.lock();
22      while (cookies <= 0)
23        wait.await();
24      cookies -= 1;
25    } finally {
26      lock.unlock();
27    }
28  }
29
30  void refill(int i) {
31    try {
32      lock.lock();
33      cookies += i;
34    } finally {
35      wait.signalAll();
36      lock.unlock();
37    }
38  }
39 }
```

(Part b). Yes, because multiple threads will get `howManyLeft()==0` before the first has time to bake more. Suggestions: (1) make `howManyLeft` block after first time returning 0, (2) add a `shallIBake` method in `Table` which only assigns baking task to one caller. *(4p)*