Chalmers | Göteborgs Universitet

Principles of Concurrent Programming TDA384/DIT391

Saturday, 27 October 2018

Exam supervisor: Sandro Stucki (sandros@chalmers.se, 076 420 86 39)

(Exam set by K. V. S. Prasad based on the course given Sep-Oct 2018)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

Points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

Points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will not receive any points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; six questions, numbered Q1 through Q6. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1. The program below is meant to guess a non-zero integer that the user types in (at line 9). It does concurrent sequential searches, upwards and downwards from 0, and is correct if both processes terminate after one of them has found the goal.

```
1  class Guess implements Runnable {
2    int you, me;
3    public Guess (int x) {me=x; you=1-me;}
4      /*  Used in lines 25, 26 to make threads p and q.
5      In p, me=0 and you=1. In q, me=1 and you=0
6      p and q share variables t, found and goal. t is 0 or 1.*/
7    static int t=0;
8    static boolean found = false;
9    static int goal = Integer.parseInt(System.console().readLine());
10
11   public void run () {
12     int i = 0;
13     while (!found) {
14       while (t == you) {};
15       t = you;
16       if (me==0) {i++;} else {i--;};
17       if (i==goal) {found=true;};
18     };
19     if (i==goal) {System.out.println ("found " + i + "\n");}
20     else {System.out.println ("done " + i + "\n");};
21     t = you;
22   }
23
24   public static void main (String[] args) {
25     try { Thread p = new Thread (new Guess(0));
26           Thread q = new Thread (new Guess(1));
27           p.start (); q.start ();
28           p.join (); q.join ();
29     } catch (InterruptedException e) {System.out.println ("except");}
30   }
31 }
```

(Part a) Is the program correct? Say why you think it is, or find a counterexample scenario. In the latter case, say what the problem is—one or both processes loop, or something else. *(4p)*

(Part b) Comment out line 21 (`t=you`), and repeat Part a. Say why you think this new program is correct, or find a counterexample scenario. In the latter case, say what the problem is—one or both processes loop, or something else. *(4p)*

```
1 class CS-user implements Runnable {
2   int you, me;
3   public CS-user (int x) {me=x; you=1-me;}
4       /* Used in lines 27, 28 to make threads p and q.
5          In p, me=0 and you=1. In q, me=1 and you=0.
6          p and q share variables t and z. t is 0 or 1.
7          z[i] means i is not trying to enter its CS. */
8   static int t=0;
9   static boolean[] z = {true, true};
10  public void run () {
11      while (true) {
12            // Non-critical section (NCS) before L2, after L5.
13            // NCS code omitted. Processes can die here, but not in CS.
14        L2: z[me] = false;
15            t = you;
16            while (true) {
17              L3: if (z[you] || t==me) {
18                    // Critical section (CS) after L3, before L5.
19                    // CS Code omitted.
20                L5: z[me]=true;
21                    break;
22                  };                          //end if
23              }                               //end while
24      }                                       //end while
25  }                                           //end run
26  /* Main program, skipping details:
27           Thread p = new Thread (new CS-user(0));
28           Thread q = new Thread (new CS-user(1));
29          p.start (); q.start ();                    */
30 } //end CS-user
```

In the transition table below, pi = "p is at Li", T = true, and so on.

s = (pi, qi, z[0], z[1], t)		sp=next state if p moves	sq=next state if q moves
s1	(p2, q2, T, T, 0)	(p3, q2, F, T, 1) = s5	(p2, q3, T, F, 0) = s3
s2			
s3	(p2, q3, T, F, 0)	(p3, q3, F, F, 1) = s7	(p2, q5, T, F, 0) = s4
s4			
s5	(p3, q2, F, T, 1)	(p5, q2, F, T, 1) = s9	(p3, q3, F, F, 0) = s6
s6			
s7	(p3, q3, F, F, 1)	no move	(p3, q5, F, F, 1) = s8
s8	(p3, q5, F, F, 1)	no move	(p3, q2, F, T, 1) = s5
s9	(p5, q2, F, T, 1)	(p2, q2, T, T, 1) = s2	(p5, q3, F, F, 0) = s10
s10			

Figure 1: CS-user program and state-transition table for Q2 and Q3.

Page left deliberately blank, so you can separate page 3 for use with Q2 and Q3.

Q2. Fig. 1 on page 3 shows a program meant to solve the critical section (CS) problem, and an abbreviated state-transition table for the program, with four blank rows.

The outer loop of the program (lines 11 to 24) encloses the NCS and a pre-protocol (lines 14 and 15), followed by the inner loop (lines 16 through 23). Let `OK = z[you] || t==me`. The inner loop repeats until OK; then, the process executes its CS and post-protocol (L5) and breaks out to the main loop.

The state transition table considers only the points L2, L3 and L5 in the program. Let `pi` stand for `p2, p3, or p5`, and the boolean `pi` for "process `p` is at Li". Similarly for `qi`. Let zp=z[0] and zq=z[1]. The values of z are abbreviated T and F.

The table represents each state by a 5-tuple `(pk, ql, zp, zq, t)`. The left column lists the states, sorted first on `pi`, then successively on `qi`, `zp`, `zq`, and `t`. The states are named s1 through s10. The next state if p (respectively q) next executes a step is given in the middle (respectively last) column. In many states both p or q can execute the next step, and either may do so. But in some states, one or other of the processes may have no move to a new state.

(Part a) Fill in the blank rows to complete the state transition table. Each entry should show a state, and in the middle and last columns, also give its name. *(4p)*

(Part b) Prove from your state transition table that the program ensures mutual exclusion. *(1p)*

(Part c) Prove from your state transition table that the program does not livelock, i.e., enter a state which neither p nor q can leave. *(1p)*

(Part d) Does every state need all 5 elements of the 5-tuple to uniquely identify it, or do some of the elements follow from the others? Using such dependencies, what is the most concise state representation you can reach? *(4p)*

(Part e) Prove that given fair scheduling, every p2-state (one where p is at p2) will lead at some future point to a p5-state. Hint: Iteratively build a set S of all states that must lead to a p5-state in zero or more moves. First, $S :=$ the set of all p5-states, i.e., $S = \{s9, s10\}$. Next, $S := S \cup \{s6\}$, as s6 must lead into S. This way, find the states that must lead finitely to a p5-state. List these states first. *(3p)*

For the remaining states, show that every state on every path from a p2-state has a transition into S via a move by p. A fair scheduler has to choose one of these moves at some point. *(2p)*

Q3. Refer again to Fig. 1 on page 3, and see Q2 for notation used in reasonoing about the program. In particular, what we mean by p2, q3, zp, zq and so on. We also use the state names from Q2.

In this question, you must argue from the program, not from the state transition table (though you may seek inspiration from it!). You get full credit for correct reasoning, whether you use formal logic, everyday language, or a mixture. Formulas and logical laws make your argument concise and precise, and help you keep track of it. With everyday language, be careful not to be fuzzy, or to mistake wishful thinking for proof.

The Appendix reviews briefly the notation of propositional logic and linear temporal logic.

Let $M = \neg(p5 \wedge q5)$ and $L = p2 \rightarrow \Diamond p5$. Your task is to prove mutex, $\Box M$, and liveness, $\Box L$, assuming weak fairness: if a transition is continually enabled, it will take place at some time.

Let $N = (p2 \text{ iff } \texttt{zp}) \wedge (q2 \text{ iff } \texttt{zq})$.

(Part a). Show that N is invariant, i.e., that the start state $s1$ satisfies $\Box N$. *(2p)*

(Part b). Show $\Box M$ by induction. First show that the start state $s1$ satisfies M.

Then show that every transition preserves M. The only transitions that could start with M and end with $\neg M$ are those starting from $p3 \wedge q5$ or $p5 \wedge q3$. By symmetry, we need deal only with $p5 \wedge q3$. Use Part a to show that in this state, each combination of `zp`, `zq` and `t` allows precisely one of p and q to proceed. You will need to consider the two kinds of states that can transition to a $p5 \wedge q3$ state. *(5p)*

(Part c). Prove $\Box L$. *Hint:* Note that p can proceed from $p2$ to $p3$, by fairness. Then show that if p is blocked, q must unblock it by an execution of $q2$ or $q5$. If p is unblocked but not scheduled, then q will unblock p and block itself. *(5p)*

Q4. Consider the Erlang program below.

```
1 -module(what).
2 -compile(export_all).
3
4 ints(N, MAX, Pidi) when N > MAX ->
5   skip;
6 ints(N, MAX, Pidi) ->
7   Pidi!N,
8   ints(N+1, MAX, Pidi).
9
10 front() ->
11   receive
12     N ->
13       X = spawn(what, front, []),
14       io:format("~p~n", [N]),
15       filter(N, X)
16   end.
17
18 filter(P, X) ->
19   receive
20     N when N rem P == 0 ->
21       X!N,
22       filter(P, X);
23     _ ->
24       filter(P, X)
25   end.
26
27 what (P,MAX) ->
28   Pidi = spawn(what, front, []),
29   spawn(what, ints, [P, MAX, Pidi]).
```

(Part a). Suppose we compile the program and evaluate `what: what(2,100)`. Draw pictures of the network as the computation takes its first few steps. What does the program print? *(5p)*

(Part b). What is printed if you evaluate `what: what(3,100)`? `what: what(5,100)`? *(2p)*

(Part c). Does the program terminate? *(1p)*

(Part d). Does it matter whether the receiving processes read their inputs right away, or wait till several messages accumulate? *(2p)*

(Part e). Change the program so that `what: what(2,100)` prints all the primes from 2 to 100. *(2p)*

Q5. A single-lane road changing direction when there is a gap in traffic.

```
1 class Car implements Runnable {
2   static final int E=0, W=1;
3   static Semaphore [] EW = {new Semaphore(0), new Semaphore(0)};
4   static Semaphore toll = new Semaphore(1);
5   static int [] q = {0, 0}, b = {0, 0};
6   int same, opp;
7   public Car (int x) {same = x; opp = 1 - same;}
8     /* Lines 39, 40 make threads Car(E) (resp. Car(W)), going east
9     (resp. west). For Car(E), opp=W. For Car(W), opp=E. The ints
10    q[E] (resp. b[E]) say how many cars are waiting to go (resp.
11    are crossing) eastwards. For west, q[W] and b[W]. Car(E) (resp.
12    Car(W)) might wait on semaphore EW[E] (resp. EW[W]). */
13
14  public void run() {
15    try {//to wait to enter,
16      toll.acquire();
17                // Trace print car name + "registers");
18      if (b[opp]>0){
19        q[same]++;
20        toll.release();
21        EW[same].acquire();
22      };
23      b[same]++;
24      if (q[same] > 0){
25        q[same]--;
26        EW[same].release();
27      } else toll.release();
28                  // Trace print car name + "crossing").
29      toll.acquire();
30      b[same]--;
31      if(b[same] == 0 && q[opp]>0) {
32        q[opp]--;
33        EW[opp].release();
34      } else toll.release();
35    } catch (InterruptedException e){System.out.println("except");}
36  }
37
38  /*  The main program, skipping details, is
39    Car CE = new Car(E); Car CW = new Car(W);
40    Thread e1 = new Thread(CE); Thread w1 = new Thread(CW);
41    etc., followed by e1.start(), w1.start(), etc. */
42 }
```

[**Q5. contd.**] A stretch of hill road is wide enough to fit only one vehicle, but cars can enter from both directions. Traffic flows in only one direction at a time; the direction, east (E) or west (W), can change when there is a gap in the traffic.

The cars are named `e1, e2, w1, w2`, etc., so we know which way they are going. For convenience in the program, E and W are given integer values 0 and 1. For each car, `same` is its direction, and `opp` is the opposite direction.

To control traffic, there are elecronically linked toll gates at both ends of the stretch, registering cars upon entry, and checking that every car has left safely.

(Part a). Put trace prints at line 17 (after `toll.acquire()`) and at line 28, giving car name and "registers" or "crossing", respectively. Suppose there are only 3 cars, two east-bound, and one west-bound. For the printout beginning with "e1 registers, w1 registers", and the one starting "w1 registers", what are the possible crossing sequences? Show a sequence of semaphore actions that produces each crossing sequence. *(4p)*

(Part b). Is a registering car sure to cross? *(2p)*

(Part c). Show that $0 \leq \texttt{toll} + \texttt{EW[E]} + \texttt{EW[W]} \leq 1$, where we use the names of the semaphores to stand for their values. *(4p)*

(Part d). Show that the crossing cars are therefore either all east-bound or all west-bound. *(3p)*

Q6. AtomicInteger is an int value that may be updated atomically. Recall that a non-blocking algorithm is *lock-free* if it guarantees system-wide progress, and *wait-free* if it also gaurantees per-thread progress.

Let `x` be a shared variable that might be set by multiple processes. The method compare-and-set, `x.CAS(expect, new)`, atomically sets `x` to `new`, and returns `true`, if `x == expect`. It returns `false` if `x != expect`. CAS is more expensive than ordinary reads and writes.

(Part a). The code snippet below is executed by thread `t`. Say, with reasons, what properties the snippet has.

```
1 AtomicInteger x = new AtomicInteger(0);
2              %thread t executes code below
3 int v;
4 do{
5   v = x.get();
6   v = v + 1;
7 } while (!x.CAS(v - 1, v));
```

Is the code starvation free? Lock-free? Wait-free? Can the process exit the code without updating `x`? *(2p)*

(Part b). The code snippet below is executed by thread `t`. Say, with reasons, what properties the snippet has.

```
1 AtomicInteger x = new AtomicInteger(0);
2 %thread t executes code below
3 for (int i = 0; i < 10000; i++) {
4   v = x.get();
5   v = v + 1;
6   if (x.CAS(v - 1, v))
7     break;
8 }
```

Is the code starvation free? Lock-free? Wait-free? Can the process exit the code without updating `x`? *(2p)*

(Part c). The above two snippets try to update a variable and retry if need be. Under high contention this could result in a lot of retries. For that situation, suggest a variation whose retries are cheaper. *(4p)*

(Part d). Under what conditions would lock-based algorithms offer better throughput than nonblocking ones? *(2p)*

A Linear Temporal Logic (LTL) notation

1. An atomic proposition such as $q2$ (process q is at label $q2$) *holds for* a state s if and only if process q is at $q2$ in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", $\rightarrow$ for "implies", $\Box$ for "always", and $\Diamond$ for "eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ ("ϕ or ψ") is false iff both ϕ and ψ are false. This is an "inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ ("ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\Box$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\Box\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$