Chalmers | Göteborgs Universitet

Concurrent Programming TDA383/DIT390

Wednesday, 9 January 2019

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

(Exam set by K. V. S. Prasad)

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. Below is the pseudo-code of a program with two threads, p and q.

static volatile int n = 0;	
p	q
p1: while (n < 2) {	q1: n++;
p2: println(n);}	q2: n++;

The labels p1, p2, q1 and q2 are given only for ease of reference. After executing p1, thread p terminates if $n \geq 2$. If $n < 2$, then p's next steps are p2 and p1 again. Thread q executes (once) q1 and then q2 .

If you prefer actual Java code to pseudo-code, please see Appendix A.

(Part a) Must the value 2 appear in the output? Say why it must appear, or give a scenario where it doesn't appear. *(2p)*

(Part b) What is the maximum number of times the value 2 can appear in the output? Why? *(2p)*

(Part c) Assuming a fair scheduler, what is the maximum number of times the value 1 can appear in the output? Give a scenario. *(2p)*

(Part d) If the scheduler can be unfair, what is the maximum number of times the value 1 can appear in the output? Give a scenario. *(2p)*

Figure for use with Q2 and Q3.

The program below is meant to solve the critical section (CS) problem. In the pseudo-code below, of a program with two threads, p and q, the instruction `swap(turn, token)` exchanges the values of `turn` and `token` atomically, i.e., in one step, with no instruction by the other thread occurring part way through `swap`.

static volatile int token = 1;			
	p		q
	int turn = 0;		int turn = 0;
	while (true) {		while (true) {
	//non-critical section		//non-critical section
	while (true) {		while (true) {
p2:	swap(turn, token);	q2:	swap(turn, token);
p3:	if (turn==1) {break;}	q3:	if (turn==1) {break;}
	};		};
p5:	//critical section	q5:	//critical section
	swap(turn, token);		swap(turn, token);
	}		}

The labels are only for ease of reference. Remember that `pi` means that *the next command* `p` will execute is `pi`. Remember also that the critical sections must terminate, but that the non-critical sections need not.

If you prefer actual Java code to pseudo-code, please see Appendix A.

The transition table below abbreviates "`p` (resp. `q`) is at `pi`(resp. `qi`)" by just "`pi`" (resp. "`qi`") for `i=2,3,5`. We abbreviate "the value of `turn` in thread `p` (resp. `q`)" by `tp` (resp. `tq`), and then write `P` (resp. `Q`) to mean `tp=1` (resp. `tq=1`). We write `T` to mean `token=1`. We write `_` to mean either `token=0` or `tp=0` or `tq=0`.

Some entries are left for you to fill in. The table has 11 rows.

s = (pi, qi, token, tp, tq)		sp=next state if p moves	sq=next state if q moves
s1	(p2, q2, T, _, _)	fill this in = s6	(p2, q3, _, _, Q) = s3
s2	(p2, q3, T, _, _)	fill this in = s7	(p2, q2, T, _, _) = s1
s3	(p2, q3, _, _, Q)	(p3, q3, _, _, Q) = s8	(p2, q5, _, _, Q) = s4
s4	(p2, q5, _, _, Q)	(p3, q5, _, _, Q) = s9	(p2, q2, T, _, _) = s1
s5	(p3, q2, T, _, _)	(p2, q2, T, _, _) = s1	(p3, q3, _, _, Q) = s8
s6	fill this in	fill this in = s10	fill this in
s7	fill this in	fill this in = s11	fill this in
s8	(p3, q3, _, _, Q)	(p2, q3, _, _, Q) = s3	(p3, q5, _, _, Q) = s9
s9	(p3, q5, _, _, Q)	(p2, q5, _, _, Q) = s4	(p3, q2, T, _, _) = s5
s10	fill this in	fill this in	fill this in
s11	fill this in	fill this in	fill this in

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

This page left blank so tht you can detach p. 3 when doing Q2 and Q3, which refer to the program and table on p. 3.

Q2. Refer to the program in Fig. 1 on p. 3. The local variables `turn` and the global variable `token` take only values 0 and 1.

We consider only the points `p2, p3, p5, q2, q3` and `q5` in the program. We abbreviate "`p` (resp. `q`) is at `pi`(resp. `qi`)" by just "`pi`" (resp. "`qi`") for `i=2,3,5`.

Remember that we abbreviate"the value of `turn` in thread `p` (resp. `q`)" by `tp` (resp. `tq`), and then write `P` (resp. `Q`) to mean `tp=1` (resp. `tq=1`). We write `T` to mean `token=1`. We write `_` to mean either `token=0` or `tp=0` or `tq=0`.

The table represents each state by a 5-tuple `(pk, ql, token, tp, tq)`. The left column lists the states, sorted first on `pi`, then successively on `qi`, `token`, `tp`, and `tq`. The states are named s1 through s11. The next state if p (respectively q) next executes a step is given in the middle (respectively last) column.

(Part a) Fill in the blanks to complete the state transition table. Each entry should show a state, and in the middle and last columns, also give its name. If you think a thread has no move from a state, write "no move" in the corresponding entry. *(4p)*

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

(Part c) Can the program deadlock, i.e., does it have a state from which neither thread can move? Prove your answer from the state transition table. *(1p)*

(Part d) Is a triple `(pk, ql, X)` enough to represent each state instead of the 5-tuple above? If so, what values would X take? *(3p)*

(Part e) A sequence of states $s_1, s_2, ..., s_k$ is said to be a *path* if there is a move by either `p` or `q` that leads from each s_i to s_{i+1}, and a path is called a *cycle* if $s_k = s_1$.

A `p5`-state is one where `p` is at `p5`. A `p`-*Starvation* cycle is a cycle with no `p5`-states, but at least one `p`-move. So looping around a `p`-*Starvation* cycle means `p` runs infinitely often, yet never reaches a `p5`-state.

Care! `s1, s3, s4, s1, ...` is a cycle, and it avoids `p5`, but it does not count as `p`-*Starvation*, because no step in the cycle is a `p` move.

Find a `p-Starvation` cycle in the program. *(4p)*

(Part f) Let S_0 be the set of all p5-states. Find a set S_1 of states from which every move leads either to S_0 or to S_1. *(3p)*

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, tp, tq` and so on. We also use the state names from Q2.

Here in **Q3**, 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. With everyday language, be careful not to be fuzzy, or to mistake wishful thinking for proof.

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

Let I be $(\texttt{tp} + \texttt{tq} + \texttt{token} = 1)$ and M be $\neg(p5 \wedge q5)$.

(Part a). Show that I is invariant, i.e., that it holds when the program starts, and then after every step of execution. *(2p)*

(Part b). Show that $\texttt{p5} \rightarrow (\texttt{tp} = 1)$. Deduce that M is invariant. *(2p)*

(Part c). Show that $(\texttt{token} = 1) \rightarrow (\neg\texttt{p5} \wedge \neg\texttt{q5})$. *(2p)*

(Part d). Show that whenever $(\texttt{p2} \wedge \texttt{q2})$, then $\texttt{token} = 1$. *(2p)*

(Part e). Sketch a proof to show that $(\texttt{token} = 1) \rightarrow \Diamond(\texttt{token} = 0)$ and $(\texttt{token} = 0) \rightarrow \Diamond(\texttt{token} = 1)$. *(3p)*

Q4. Any number of east- and west-bound cars cross a single-lane bridge carrying only one car at a time. The cars behave as below.

static semaphore ES = 1; static semaphore WS = 0			
eb (eastbound car)		wb (westbound car)	
e1:	//before the bridge	w1:	//before the bridge
e2:	ES.acquire;	w2:	WS.acquire;
e3:	//crossing bridge	w3:	//crossing bridge
e4:	WS.release;	w4:	ES.release;
e5:	//after the bridge	w5:	//after the bridge

The labels are only for ease of reference. Remember that `ei` means that *the next command* `eb` will execute is `ei`. The actions `e3` and `w3` must terminate, but the actions `e1, w1, e5` and `w5` need not.

Appendix A gives an actual Java code version of the pseudo-code above, for those who prefer actual code. The actual code uses a single parameterised `run` function, so the semaphores are declared in an array, and variables are used to note the direction of travel.

(Part a). Cars `eb1, eb2` wish to cross the bridge eastwards and `wb1, wb2, wb3`, westwards. List the orders in which the cars may cross. *(4p)*.

(Part b). Suppose there are also devices called *shuttles*, `se` and `sw`, that act as below:

se (eastbound shuttle)		sw (westbound shuttle)	
	while true {		while true {
se1:	//quick and finite;	sw1:	//quick and finite;
se2:	ES.acquire;	sw2:	WS.acquire;
se3:	//quick and finite;	sw3:	//quick and finite;
se4:	WS.release;	sw4:	ES.release;
	}	w5:	}

How might the addition of shuttles improve the situation in **Part a**? But the shuttles might also bring in a new problem. What? *(4p)*.

(Part c). Suppose that the statements `se1` and `sw1` are replaced by `sleep`s for a while. How does this help in **Part b**? *(3p)*

(Part d). Suppose now that the bridge has an east-bound lane and a west-bound lane, so collisions are not a problem, and can carry a maximum of 3 cars at a time. How would you take advantage of this increased carrying capacity? If cars `eb1, eb2` wish to cross the bridge eastwards and `wb1, wb2, wb3`, westwards, show some execution sequences that get all the cars across. *(4p)*

Q5. Consider the Erlang program below.

```
1 -module(wha).
2 -export([cell/4, feed/2, cm/0, go/1]).
3
4 cell(Pred, N, M, Succ) ->
5   receive
6     dump when M /= 100 ->
7       io:format("~w~n", [M]),
8       Succ! dump;
9     X when (X > N) and (X =< M) ->
10      Temp=spawn (wha, cell, [self(), X, M, Succ]),
11      cell (Pred, N, X, Temp);
12    X ->
13      Succ! X,
14      cell (Pred, N, M, Succ)
15  end.
16
17 cm() -> receive _Any -> cm()
18         end.
19
20 feed(First, [])     -> First!dump;
21 feed(First, [H|T]) -> First! H,
22                        feed(First, T).
23
24 go (L) ->
25   Sink = spawn(wha, cm, []),
26   First = spawn(wha, cell, [Sink, 0, 100, Sink]),
27   spawn(wha, feed, [First, L]).
```

On line 4, `Pred` and `Succ` are Pids, while `N` and `M` are integers. In line 6, `dump` is an atom, and `/=` means "not equal to".

The value of the anonymous variable (`_`) in line 17 is ignored.

`First` (lines 20—22, 26, 27) and `Sink` (lines 25, 26) are Pids, and `L` (lines 24, 27) is a list of integers.

In your answers below, include sketches showing you understand how the network of processes grows and shrinks.

(Part a). What output does `go([])` give? *(2p)*.

(Part b). What output does `go([5])` give? *(2p)*.

(Part c). What output does `go([6, 5])` give? *(4p)*.

(Part d). What output does `go([-3,5,4,103,4,0])` give? *(4p)*.

Q6 Q6 is like Q4, except that here you must use monitors instead of semaphores. Any number of east- and west-bound cars cross a single-lane bridge carrying only one car at a time.

(Part a). Program access to the bridge by a monitor with one method each to cross east-bound (resp. west-bound). How are these used? *(3p)*

(Part b). Add a method or methods to the monitor to unblock cars that might be stranded in **(Part a)**. How are these used? *(2p)*.

(Part c). How can you avoid a situation where the method(s) of **(Part b)** run all the time and hinder use of the bridge? *(1p)*

(Part d). What are the advantages and disadvantages of using monitors here instead of semaphores? *(2p)*

Notation: Follow Ben-Ari, as below. A monitor in Java is a class all of whose methods are `synchronized`. Condition variables are dealt with by explicit `wait()`s, and every method ends with `notifyAll()`. For example, here are bits of the producer-consumer monitor in Java.

```
1  class PCMonitor {
2    final int N=5;            //capacity of buffer
3    int Oldest=0, Newest=0;
4    volatile int Count=0;
5    int Buffer [] = new int [N];
6
7    synchronized void Append(int V){
8      while (Count==N)
9        try {
10         wait();
11       } catch (InterruptedException e) {}
12         //put element into buffer, manage Newest, Count, etc.
13     notifyAll();
14   }
15
16   synchronized int Take(){
17     int temp;
18     while (Count==0)
19       try {
20         wait();
21       } catch (InterruptedException e) {}
22         //take element into temp, manage Oldest, Count, etc.
23     notifyAll();
24     return temp;
25   }
26 }
```

A Java code for the pseudo-code in the paper

Q1.

```
1   public class Hm {
2     static volatile int n = 0;
3
4     public static void main(String[] args) {
5       Hm hm = new Hm();
6       P p = hm.new P();
7       Q q = hm.new Q();
8       Thread tp = new Thread(p);
9       Thread tq = new Thread(q);
10      tp.start();
11      tq.start();
12    }
13
14    class P implements Runnable {
15      public void run() {
16        while (n < 2) {
17          System.out.println(n);
18        }
19      }
20    }
21
22    class Q implements Runnable {
23      public void run() {
24        n++;
25        n++;
26      }
27    }
28  }
```

Since the threads are very small, you would, to test this code in practice, include sleeps for random amounts of time in P and Q, hoping to get more interesting interleavings.

Q2 and Q3.

```
1   import java.util.concurrent.locks.ReentrantLock;
2   class CS_swap implements Runnable {
3     int turn, temp;        //turn = 1 means I hold token.
4     static int token=1;    //At init, the token is free.
5     static ReentrantLock lock = new ReentrantLock();
6
7     public CS_swap () {}
8     public void run () {
9       while (true) {           //outer loop, forever
10        while (true){
11          L2: lock.lock();
12          temp = turn; turn = token; token = temp;
13          lock.unlock();
14          L3: if (turn==1) {break;};
15        };
16        L5: lock.lock();
17        temp = turn; turn = token; token = temp;
18        lock.unlock();
19      }                        //end loop on line 9
20    }                          //end run on line 8
21
22    public static void main (String[] args) {
23      Thread p = new Thread (new CS_swap());
24      Thread q = new Thread (new CS_swap());
25                          //Make threads p and q.
26      try {p.start (); q.start ();
27          p.join (); q.join ();   //Wait for the threads to finish.
28      } catch (InterruptedException e) {System.out.println ("!");}
29    }
30  }//end CS_swap
```

This minimalist program produces no output. To test it, include trace prints, and sleeps for random amounts of time in p and q, to get more interesting interleavings. Also, a counter `inCS` to keep track of how many threads are in their critical section. To allow trace prints, `turn` might be made global, the thread identity 0 and 1 stored locally, and L3 might be under lock to allow a trace print of `inCS` without interference.

Q4.

```
1 import java.util.concurrent.Semaphore;
2
3 class Slane implements Runnable {
4   int same, opp;
5   static final int E=0, W=1;
6   static Semaphore [] EW = {new Semaphore(1), new Semaphore(0)};
7
8   public Slane (int x) {same = x; opp = 1 - same;}
9     /* Used in lines 24 and 25 to make threads
10    E (East-bound) and W (West-bound).
11    For E, same=0 and opp=1. Vice-versa for W. */
12
13  public void run() {
14    try {
15      EW[same].acquire();
16      System.out.println(Thread.currentThread().getName() + "goes");
17      EW[opp].release();
18    } catch (InterruptedException e) {
19      System.out.println(e.getMessage());
20    }
21  }
22
23 public static void main(String[] args) {
24  Slane CE = new Slane(E);
25  Slane CW = new Slane(W);
26  Thread e1 = new Thread(CE); e1.setName("e1");
27  Thread e2 = new Thread(CE); e2.setName("e2");
28  Thread w1 = new Thread(CW); w1.setName("w1");
29  Thread w2 = new Thread(CW); w2.setName("w2");
30  Thread w3 = new Thread(CW); w3.setName("w3");
31  w1.start(); w2.start(); w3.start(); e1.start(); e2.start();
32  }
33 }
```

Since the threads are very small, you would, to test this code in practice, include sleeps for random amounts of time in `run`, to try to get more interesting interleavings.

B 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:

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