Chalmers | GÖTEBORGS UNIVERSITET
K. V. S. Prasad, Computer Science and Engineering

Concurrent Programming TDA383/DIT390

Monday 24 Aug 2015 pm in V

K. V. S. Prasad, tel. 0736 30 28 22
will visit approximately one hour after the start and one hour before the end

- **Permitted materials (Hjälpmedel): Dictionary (Ordlista/ordbok)**
- Maximum you can score on the exam: 68p. This paper has six questions, on pages 2 through 4, each carrying 12p, except the last, which carries 8p. An Appendix, on pages 5 and 6, summarises the pseudo-code, logic and Linda notation used in this question paper.

  To pass the course, you need to pass each lab, and score at least 28 p on the exam. Further:

  Exam grades: (CTH): grade 3: 28-40 p, grade 4: 41-54 p, grade 5: 55-68 p.
  (GU): grade G: 28-54 p, grade VG: 55-68 p.

  Course grades: CTH (exam + labs): grade 3: 40-59 p, grade 4: 60-79 p, grade 5: 80–100 p.
  GU (exam + labs): grade G: 40-79 p, grade VG: 80–100 p.
- Results: within 21 days.
- **Notes: PLEASE READ THESE**
  - Time planning: Allow 3 minutes per point; you will then have half an hour to look over your work at the end. **Do not get stuck for more time than you can afford on any question or part**. You will be compensated for any errors in the question paper, but not for extra time you spent on one question that you should have used for another.
  - Start each question on a new page.
  - The pseudo-code notation from the Appendix should suffice for your programs, but you can use Java, Erlang or Promela **provided the constructs you use agree with what the question is about.** The exact syntax of the programming notations you use is not so important as long as the graders can understand the intended meaning. If you are unsure, explain your notation.
  - The **correctness of some answers** is clear from **inspection**. **Other answers** must be **justified**, to help us judge them. **If you think a question is incorrect**, ambiguous, inconsistent, or incomplete, **say so** in your answer. **Make the smallest changes** you need to the question, and **state them**. If you need **assumptions** beyond those given, **state** them. If your solution only works under certain **conditions, state** the conditions.
  - Be **precise**. Programs are mathematical objects, and **discussions** about them may be **formal or informal**, but are **best mathematically argued**. Handwaving arguments will get only partial credit. Unnecessarily complicated solutions will lose some points.
  - DON'T PANIC!

Question 1. Below is an algorithm with two processes, p and q.

boolean flag:=false integer n:=0	
p	q
p1: while flag = false p2: n:=1-n	q1: while n=0 skip q2: flag := true

After executing p1, process p's next step is p2 if flag = false; it terminates if flag=true.

After executing q1, process q's next step is q1 if $n = 0$, and q2 if $n \neq 0$.

(Part a). Construct a scenario for which the program terminates. What is the value of n at the end? *(4p)*

(Part b). Construct another scenario for which also the program terminates, but with a different value of n at the end. *(4p)*

(Part c). Construct a fair scenario for which the program does not terminate. Show that your scenario is indeed fair. *(3+1p)*

To write down a scenario, list the labels of the statements in the order of execution.

Question 2. A mine has three lifts, each able to two carry miners. Every morning, there are very many miners to be carried down to work. (Pretend there are an unlimited number of miners). No lift will leave until it has two passengers. Lifts return empty from the work level to the surface.

You are to simulate this by a system that prints "on n" every time a miner enters lift number n and "off n" every time a miner leaves lift number n.

Represent the miners and lifts as processes. The lifts go in a loop: pick up two miners—go down—drop off two miners—go up. Each miner does just: get on—get off.

Use a protected object C to coordinate communication between the miners and the lifts. See the Appendix if you are not sure what a protected object is. Begin with the declarations below. Assume the function index has already been programmed.

```
protected object C
    integer array[1..3] free := [2,2,2];
        // array showing free places in each lift (numbered 1 through 3)
    integer sumfree := 6;
        // int variable showing total of free places (in any lift)
    integer fun index;
        // index is a function returning 0 if sumfree=0,
        // and otherwise the number of a lift with a free place
```

(Part a). Write the pseudo-code for the lift and miner processes, and the rest of the protected object C. (You get credit for making your code as simple as possible). *(8p)*

(Part b). Now suppose you were writing in a language that only gave you binary semaphores (no monitors or protected objects), but you want to use the answer to Part a as your high level design. How would you simulate that design using binary semaphores? Sketch the code in only enough detail to explain the idea. *(4p)*

Question 3. Here is yet another algorithm to solve the critical section problem, built from atomic "if" statements (p2, q2 and p5, q5). The test of the condition following 'if', and the corresponding "then" or "else" action, are both carried out in one step, which the other process cannot interrupt. The / operator is integer division, so $2/2 = 3/2 = 1$.

integer S := 0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: if even(S) then S:=2 else S:=3	q2: if even(S/2) then S:=5 else S:=7
p3: await (S$\neq$ 1 $\wedge$ S$\neq$3)	q3: await (S$\neq$ 6 $\wedge$ S$\neq$7)
p4: critical section	q4: critical section
p5: if odd(S/2) then S:=S-2 else skip	q5: if odd(S) then S:=S-1 else skip

Commands p1, p4, q1 and q4 (the critical and non-critical sections) do not access the variable S, and are therefore omitted in the abbreviated state transition table (a tabular version of a state diagram) below for the program. Many entries in the table are left blank (—).

Each state is represented by a triple (pk, ql, Sn), where pk and ql say respectively what p and q will next execute, and Sn is the value of S. The states are listed in the order in which they appear as the table is built up starting from (p2, q2, 0), and are named s1 through s10. (There are 10 states in all). The left hand column lists the states. 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 are free to execute the next step, and either may do so. But in some states, such as s5 below, one or other of the processes may be blocked. The middle and last columns show the next state and give its name for easy reference.

	State = (pk, ql, Sn)	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)	—	—
s3	(p2, q3, 5)	—	—
s4	—	—	—
s5	(p3, q3, 7)	(p5, q3, 7)=s8	no move
s6	—	—	—
s7	—	—	—
s8	—	—	—
s9	—	—	—
s10	(p2, q2, 4)	(p3, q2, 2)=s2	(p2, q3, 5)=s3

(Part a) Fill in the dashes to complete the state transition table. *(6p)*

(Part b) Prove from your state transition table that the program ensures mutual exclusion. *(3p)*

(Part c) Prove from your state transition table that the program does not deadlock (there are await statements, so it is possible for a process to block). *(3p)*

Question 4. Refer again to the program in Question 3. 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.

Below, we write pi as a logical proposition to mean "process p is at pi".

(Part a). Show that $(p3 \wedge q3) \rightarrow (S = 3 \vee S = 7)$ is invariant. *Hint:* Reason about what must have happened for the program to get to $(p3 \wedge q3)$. *(4p)*

(Part b) Assume that $(p3 \wedge q5) \rightarrow (S = 3)$. Prove that if $p3 \wedge q5$, then p cannot move until after q executes $q5$. (That is, mutual exclusion holds). *(4p)*

(Part c) Assume that $p3 \wedge q1 \rightarrow (S = 2)$ is invariant, and that q is stuck in a loop in q1. (Remember that while p4 and q4 are assumed to terminate, p1 and q1 may loop). Assuming fairness, prove that $p3 \wedge q1 \rightarrow \Box\Diamond p5$. *(4p)*

Question 5. A directed graph G consists of a set N of *nodes* and a set E of pairs of nodes representing *one-way arcs* connecting the first node of the pair to the second. So if $(i, j) \in E$, then $i, j \in N$ (i.e., both i and j are nodes), and there is an arc starting at i and ending at j. A *path* from node k to node l, if such exists, is either the arc (k,l) or a sequence of arcs (k,n_1), (n_1,n_2), (n_2,n_3), ..., (n_r,l), (where $r \geq 1$).

We make several simplifying assumptions. Given nodes i and j, we assume that there is at most one arc $(i, j) \in E$; there are no parallel arcs sharing the same starting node i and ending node j. We also assume there are no *self-loops*, i.e., arcs of there form (i,i). Finally, we assume there are no cyclic paths following which we can start at a node i and arrive again at i.

With these assumptions, write a message-passing program P using channels to take a graph with at least 2 nodes and find if there is a path from node k to node l; here $k \neq l$. If yes, P should print out "ok", otherwise it may either loop or hang or print out "no path". Assume that messages to a channel c are distributed fairly among all processes waiting on c.

(Part a) Specify the processes P consists of, and explain how it works. *(6p)*
(Part b) Does P work for both synchronous and asynchronous channels? Explain. *(2p)*
(Part c) Modify P to print out the length of the first path it finds (the number of arcs in it). *(4p)*

Question 6. Begin again with graphs as in Question 5, with the simplifying assumptions. Assume a given graph is loaded into a Linda space as a set of pairs, one pair for each arc in the graph.

Here you are to write two Linda programs to take a graph with at least 2 nodes and find if there is a path from node k to node l; here $k \neq l$. If yes, your program should print out "ok" on at least some runs. (We allow that on some runs your program may hang or loop even if there is a path from node k to node l). If there is no path from node k to node l, your program may either loop or hang or print out "no path".

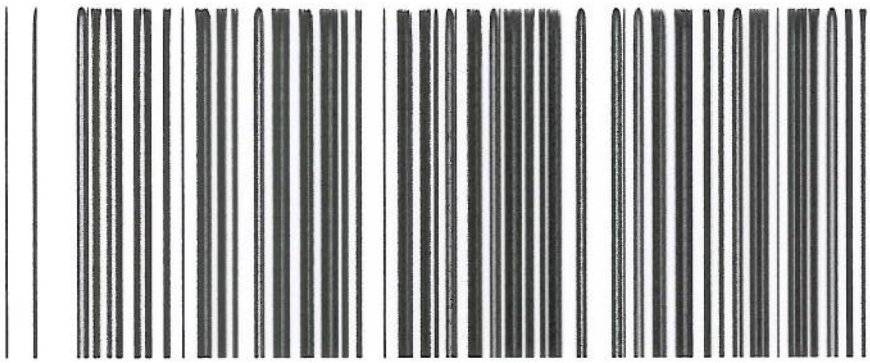
Fairness: assume that if a tuple t satisfies pattern p, then t will be returned sooner or later in answer to a *remove(p)* or *read(p)*; some other tuple cannot always be chosen in preference.

(Part a) Write a single process S with k and l in its internal memory to solve the problem in the above sense: if there is a path from k to l, then S will find the path on at least some runs. *(4p)*

(Part b) Now consider a Linda program consisting of arbitrarily many copies of S. Do the various instances of S ever wait for each other? Using the fairness assumption, show that this parallel Linda program will find a path from k to l if such exists. *(2+2p)*

——END of QUESTION PAPER——

A Appendix



A Appendix

A.1 SUMMARY OF BEN-ARI'S PSEUDO-CODE NOTATION

Global variables are declared centred at the top of the program.

Data declarations are of the form `integer i := 1` or `boolean b := true`, giving type, variable name, and initial value, if any. Assignment is written `:=` also in executable statements. Arrays are declared giving the element type, the index range, the name of the array and the initial values. E.g., `integer array [1..n] counts := [0, ..., 0]`.

Next, the statements of the processes, often in two columns headed by the names of the processes. If several processes `p(i)` have the same code, parameterised by `i`, they are given in one column.

So in Question 1, p and q are processes that the main program runs in parallel. The declarations of $flag$ and n are global. Declarations local to p or q (none here) would be in the respective columns.

Numbered statements are atomic. If a continuation line is needed, it is left un-numbered or numbered by an underscore `p_`. Thus `loop forever`, `repeat` and so on are not numbered. Assignments and expression evaluations are atomic. Indentation shows the substatements of compound statements.

The synchronisation statement `await b` is equivalent to `while not b do nothing`. This may be literally true in machine level code, but at higher level, think of `await` as a sleeping version of the busy loop.

For channels, `ch => x` means the value of the message received from the channel `ch` is assigned to the variable `x`. and `ch <= x` means that the value of the variable `x` is sent on the channel `ch`.

When asked for a scenario, just list the labels of the statements in the order of execution. With synchronous channels, sender and receiver act together, so show both statements as a pair being a single move in the scenario.

EXTENSION OF BEN-ARI'S PSEUDO-CODE NOTATION You can explicitly declare processes by a line of the kind "proctype p(integer i)" giving the name of the process and its parameters. Then write an explicit "init" process that starts the program. Explicit commands like "run p(5); run p(6)" are used to run processes, in this case to start process p with parameter 5, and then start another instance of p with parameter 6.

These extensions give new expressive power. The "run" command means the number of processes in a program can change during execution. If processes are passed channels as parameters, the network of channels between processes can change dynamically.

PROTECTED OBJECTS These are like simplified monitors, providing encapsulation and synchronisation, but without condition variables and explicit operations on them scattered all over the code. Instead, protected object operations have boolean guards (after the keyword "when"); calling an operation will block until its guard becomes true. Only one operation can be done at a time, and all guards are re-evaluated after each operation. Guards that are always true need not be mentioned.

```
protected object Sem
    integer s := k
    operation Wait when s > 0
        s := s − 1
    operation Signal
        s := s + 1
```

Above is a protected object Sem simulating a semaphore s initialised to k. Operations Sem.Wait and Sem.Signal now do the work of the wait and signal operations on s.

A.2 LOGIC

The symbols used here for the operators of propositional logic are: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", and $\rightarrow$ for "implies". These have the obvious meanings, but two differ from what might be your interpretation of the name. Note that $p \vee q$ ("p or q") is false iff (if and only if) both p and q are false. This is an "inclusive or", so $p \vee q$ is true if both p and q are true. Also, note that $p \rightarrow q$ ("p implies q") is false iff p is true and q is false. In particular, this means $p \rightarrow q$ is true if p is false.

The particular form of logic used here is Linear Temporal Logic (LTL). LTL is propositional logic with two added operators, $\Box$ and $\Diamond$.

A proposition such as q_2 (process q is at label q_2) is true of state s_1 in Question 3, because in that state process q is at q_2. This proposition q_2 is also true of state s_4 but false of state s_2.

A *path* is a possibly infinite sequence of states. It is a possible future of the system. So from the same table, the infinite loop of states s_1, s_4, s_8, s_1, etc. is a possible future at any of the states s_1, s_4, and s_8.

A path satisfies the formula $\Box q_2$, say, if q_2 is true of the first state of the path, and for all subsequent states in the path. Eg., the infinite loop s_1, s_4, s_8, s_1, ... satisfies $\Box q_2$.

A path satisfies the formula $\Diamond p_5$, say, if p_5 is true of some state in the path. Eg., the infinite loop s_1, s_4, s_8, s_1, ... satisfies $\Diamond p_5$.

Finally, we extend the satisfaction definition from paths to states. A formula ϕ holds for state s (or, s satisfies ϕ) if every path from s satisfies ϕ.

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

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

A.3 LINDA

In Linda programs, processes communicate via *tuples* posted on a *board*. The first element of a tuple is often a constant string, saying what kind of tuple it is. Processes interact with the board through three kinds of atomic actions.

post(t) Here t is a tuple $\langle x_1, x_2, .. \rangle$, where the x_i are constants or values of variables. *post*(t) posts t on the board, and unblocks an arbitrary process among those waiting for a tuple of this pattern.

remove$(x_1, x_2, ..)$ Here the parameters must be variables or constants. The command *remove*$(x_1, x_2, ..)$ removes a tuple $\langle x_1, x_2, .. \rangle$ that matches the pattern of the constant parameters, and assigns the tuple values to the variable parameters. If no matching tuple exists, the process is blocked. If there are several matching tuples, an arbitrary one is removed.

read$(x_1, x_2, ..)$ Like *remove*$(x_1, x_2, ..)$, but leaves the tuple on the board.

——END of APPENDIX——

Here is a sketch of solutions to the exam of 24 Aug 2015.

Q1. a) The scenarios is p1, p2, q1, q2, p1. (n=1 at the end).

b) Then p1, p2, q1, p1, q2, p2, p1 terminates with n=0 at the end.

c) The scenarion (p1, p2, p1, p2, q1)* looping. Then n becomes 1, then 0, and q1 only tests it when n=0. This is fair, because both p and q execute infinitely often.

Q2. a) Here is a sketch of a solution.
What happens at the bottom of the shaft is abbreviated.

```
protected object C
    integer array[1..3] free := [2,2,2];
       // array showing free places in each lift (numbered 1 through 3)
    integer sumfree := 6;
       // int variable showing total of free places (in any lift)
    integer fun index;
       // index is a function returning 0 if sumfree=0,

    entry miner_top when sumfree > 0
       sumfree--;
       i:= index;       //find free lift i;
       free(i)--;
       print("on" i)

    entry lift_arrive (i)
       free(i)=2;
       sumfree++ ++

    entry lift_depart (i) when free(i)=0
          // go down, then:
       print("off" i)
       print("off" i)

proc lift(i)
    loop
       lift_depart(i);
       lift_arrive(i)


proc miner;
    miner_top
```

Q2 b) There are several solutions possible. For example, sumfree can be mimicked by a signal for sumfree ++, and by a wait for sumfree --. The array free can be guarded by one semaphore, while three others record whether free(i)=0. Call index only when you have exclusive access to sumfree; release this only after miner_top.

Q3. The table is

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)=s2

s10=(p2, q2, 4) (p3, q2, 2)=s2 (p2, q3, 5)=s3

Q 3 b) There is no state (p5, q5, sn) in the table. Remember that p5 and q5 are in the critical section; the process only leaves that after executing p5 or q5.

Q3 c) There is no state where neither p nor q can move.

Q4 a) To get to p3 and q3, we have either

p2 and q3 followed by doing p2. q3 implies S=5 or S=7. Then p2 makes S=3.

or

p3 and q2 followed by doing q2. p3 implies S=2 or S=3. In either case, q2 makes S=7.

Q4 b) If p3 and q5, we may assume S=3. Then p3 has to wait until q5 runs and makes S=2. Very simple, and not worth 4p, but it merely

checks whether the student can make a short logical argument.

Q4 c) Suppose p3, q1. So S=2, given. If q1 loops, it cannot reset S. (We can perhaps state in the intro that p2, q2, p5 and q5 are the only commands that reset S, instead of assuming that the students are supposed to know that protocol variables are only to be accessed in pre- or post-protocols). So p3 will move on to p5. We need to assume fairness, otherwise q could hog all the CPU time.

Q5. a) Assign a channel to each node, and a process to each arc. The channels are of type unit (they transmit beeps), but they can be declared to be of type int, bool, ... We won't use the value.

For the arc (i, j) we have a process

```
process P(i,j)          //where i and j are channels of unit
unit dummy;

i => dummy;         // read from channel i
loop forever
    j <= dummy;     // write to channel j
```

We also need

```
process Start (k)    //where k is a channel of unit
loop forever
        k <= dummy;     // write to channel k
```

and

```
process End (l)    //where l is a channel of unit
l <= dummy;     // read from channel l
print ("ok")
```

So to check if there is a path from k to l, Start floods k with beeps. All arcs from k will relay this fllod on to their end nodes. Flood the whole graph until a beep reaches l. Then we know there is a path.

Q5 b) The arcs don't synchronise with each other, so the channels can be synchronous or asynchronous.

Q5 c) For this, the channels are of type int. Start floods 0 onto k. Each arc (i,j) receives dummy and sends dummy+1. The end node gets a number giving the length of the path that reached it.

Q6. a)

```
process S(k, l)

int x;

loop
    read(k, x);
    if x = l
    then
        print("ok");
        break
    else
        k := x
end loop
```

This will start at x (=k to start with) and look for an arc leading away. If it finds l, great, otherwise just keep going till there are no arcs left (hang).

If there is a path k to l, some run of S will find it (by fairness assumption).

Q6 b) This is simply part a except that instead of running S repeatedly, we run all the repetitions in parallel.

Q6 is almost not about Linda, but about general ideas of parallel search.