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

Concurrent Programming TDA383/DIT390

Saturday 21 Mar 2015 pm in M

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: 68 p. This paper has six questions, on pages 2 through 6, each carrying between 6 and 15 p. An Appendix, on pages 7 and 8, summarises the pseudo-code, lexicographic ordering, 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**. Students are compensated for errors in the question paper, but not for extra time spent on one question that should have been 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. Assume that for the function f, there is some integer value i for which $f(i) = 0$. Below are two algorithms that each use two processes in parallel to search for i.

Algorithm 1.1	
boolean found	
p	q
integer i := 0	integer j := 1
p1: found := false	q1: found := false
p2: while not found	q2: while not found
p3: i := i+1	q3: j := j-1
p4: found := f(i)=0	q4: found := f(j)=0

Algorithm 1.2	
boolean found:=false	
p	q
integer i := 0	integer j := 1
p1: while not found	q1: while not found
p2: i := i+1	q2: j := j-1
p3: found := f(i)=0	q3: found := f(j)=0

For each algorithm, show either that both processes terminate after one of them has found an i for which $f(i) = 0$, or find a counterexample. *(3+3p)*

Question 2. In the delightfully old-fashioned university of Freedonia, students still read physical books but can't afford to buy them, so the library stocks multiple copies of course books. Suppose a course has s students and the library holds c copies of the course book b, with $0 < c < s$. Students borrow b, but neither know nor care which of the c copies they get. A student who tries to borrow b when all copies are loaned out will sleep till one of the copies is returned. When that happens, the library wakes up one of the sleepers and loans them the copy. Suppose each student is modelled by the following code:

```
Student
loop forever
p1:   non-critical section (might terminate here)
p2:    pre-protocol (borrow a copy of the course book - might involve sleeping)
p3:    read the borrowed copy
p4:    post-protocol (return copy)
```

(Part a). (1) Using a single general semaphore, write simple pre- and post-protocols to borrow and return a book. (2) Define the operations you use on the semaphore. *(3+1p)*

(Part b). Suppose the library computerizes its work, and uses one monitor per book to take care of all the c copies. (1) Write the monitor code, and the code needed to use it. (2) Define the operations you use on condition variables. (3) Suppose a student p returns a book, and there is a sleeper q waiting for it. In your code, does p leave the monitor before q wakes up (Mesa rule), or does q wake up and borrow the book while p waits (immediate resumption rule)? *(3+1+1p)*

(Part c). Now suppose the programming language has no monitor construct, only semaphores. Re-implement a semaphore solution, making in each student process the exact sequence of operations on semaphores that would happen if you were to use monitors (which in turn were implemented using semaphores). So each student process has all the code to manage queues, mutual exclusion to shared variables, etc. Begin with these declarations:

```
Student
                               /* variables shared by the student processes */
int count = c;                 /* number of available copies */
int waiting = 0;               /* number of students asleep, waiting for a copy */
binary semaphore mutex;        /* for mutual exclusion over count and waiting */
binary semaphore bookfree;     /* for the queue of sleeping students */
```

Now write the rest of the student code. *(6p)*

Question 3. Here is yet another algorithm to solve the critical section problem, built from "await" commands (p3, q3), that await either of two conditions, and atomic case commands p2, q2, p5 and q5. In the case commands, the test on S, and the subsequent assignment to it, all take place without interruption. The global variable S has 5 possible values, S_1 through S_5. These are given mnemonic names: $S_1 = Z$, $S_2 = P$, $S_3 = Q$, $S_4 = PQ$, and $S_5 = QP$.

```
                  type switch = {Z, P, Q, PQ, QP}
                          switch S := Z
p                                    q
   loop forever                         loop forever
p1:   non-critical section           q1:   non-critical section
p2:   case S of                      q2:   case S of
         Z → S:=P;                            Z → S:=Q;
         Q → S:=QP;                           P → S:=PQ;
         else → skip                          else → skip
p3:   await (S=P or S=PQ)            q3:   await (S=Q or S=QP)
p4:   critical section               q4:   critical section
p5:   case S of                      q5:   case S of
         P → S:=Z;                            Q → S:=Z;
         PQ → S:=Q;                           QP → S:=P;
         else → skip                          else → skip
```

Below is part of the state transition table (abbreviated "table") for an abbreviated program, skipping p1, p4, q1 and q4. The left hand column lists the states (where p and q are, and the value of S), lexicographically (the Appendix says more on this): (pi, qj, S_k) before $(pi', qj', S_{k'})$ iff $i < i'$, or $i = i'$ and $j < j'$, or $i = i'$ and $j = j'$ and $k < k'$. The middle (resp. right) column gives the next state if p (resp. q) next makes a move. In states like s_2, either p or q can make the next move. In states like s_5, one or both of p and q may be blocked. There are 9 states in all.

	State = (pi, qi, S)	next state if p moves	next state if q moves
s_1	(p2, q2, Z)	(p3, q2, P)	(p2, q3, Q)
s_2	(p2, q3, Q)	(p3, q3, QP)	(p2, q5, Q)
s_3			
s_4	(p3, q2, P)	(p5, q2, P)	(p3, q3, PQ)
s_5	(p3, q3, PQ)	(p5, q3, PQ)	no move
s_6			
s_7			
s_8	(p5, q2, P)	(p2, q2, Z)	(p5, q3, PQ)
s_9			

(Part a) Complete the table lexicographically (we have left 4 lines blank). *(4p)*

(Part b) Prove from your table that the program ensures mutual exclusion. *(2p)*

(Part c) Prove from your table that the program does not deadlock. *(2p)*

(Part d) 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 M of all states that must lead to a $p5$-state in zero, one or more moves. First, $M :=$ the set of all $p5$-states. E.g., $s_8 \in M$. Next, $M := M \cup \{s_5\}$, as s_5 must lead to a p_5-state. Then $M := M \cup \{s_4\}$, etc. If every $p2$-state leads to M infinitely often, then fair scheduling means that the move will be made at some point. *(4p)*

Question 4. Refer again to the program in Question 3, reproduced here for convenience. Remember that the "await" commands (p3, q3) await either of two conditions, and that the case commands p2, q2, p5 and q5 are atomic.

type switch = {Z, P, Q, PQ, QP} switch S := Z	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: case S of	q2: case S of
Z → S:=P;	Z → S:=Q;
Q → S:=QP;	P → S:=PQ;
else → skip	else → skip
p3: await (S=P or S=PQ)	q3: await (S=Q or S=QP)
p4: critical section	q4: critical section
p5: case S of	q5: case S of
P → S:=Z;	Q → S:=Z;
PQ → S:=Q;	QP → S:=P;
else → skip	else → skip

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.

In the sequel, we write *pi* as a logical proposition to mean "process *p* is at *pi*". Also, for $S = X$, we write just X, as the symbols *Z*, *P*, *Q*, *PQ* and *QP* are unambiguous in context.

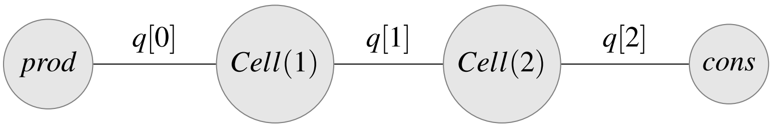
Let $M_p \equiv p4 \rightarrow (P \vee PQ)$, i.e., if *p* is at *p*4, then *S* will be *P* or *PQ*.

(Part a). Show that $\Box M_p$, i.e., that M_p is invariant (always holds).
Hint: How could M_p be false? Either because *p* arrives at *p*4 when *S* is neither *P* nor *PQ*, or by both *P* and *PQ* becoming false while *p* is at *p*4. Show that neither can happen. *(4p)*

(Part b). Assume $\Box M_p$ and its symmetric counterpart $\Box M_q$ for *q* are both always true. Prove that the program ensures mutual exclusion, i.e., $\Box\neg(p4 \wedge q4)$. *(3p)*

(Part c). Show that $\neg\Diamond\Box(p3 \wedge q3)$, i.e., the program cannot reach a deadlocked state.
Hint: Suppose $\Box(p3 \wedge q3)$, i.e., the system is deadlocked. Then what must *S* be? Can it hold this value after *p*2 or *q*2 (one of which must be the last command before $p3 \wedge q3$)? *(4p)*

Question 5. Implement a buffer of size MAX using a set of MAX processes called $Cell(i)$, where i ranges over $1..MAX$, and an array $0..MAX$ of synchronous channels q. Use channel $q[i]$ to connect $Cell(i)$ and $Cell(i+1)$. You will also need a process *prod*, which produces an infinite stream of identical items (represented as 1's), and a consumer process *cons* that receives the items and throws them away. Finally, use channel $q[0]$ to connect *prod* to $Cell(1)$, and channel $q[MAX]$ to connect $Cell(MAX)$ to *cons*. Here's a picture with $MAX = 2$.



And here are the declarations:

q: array [0..MAX] of synchronous channel		
proctype Cell(i: 1..MAX)		
	int x;	/* one local variable per process */
p1:	...	/* commands begin here */
p2:	...	
p3:	...	

Thus each process $Cell(i)$ can hold at most one item at a time, in variable x. Use only channels $q[i-1]$ and $q[i]$ in $Cell(i)$.

(Part a) Write code for *prod*, *cons* and $Cell(i)$. Label each line, and put each command on a separate line, so that every state of each process has a name. *(4p)*

(Part b) Let $MAX = 2$. Make a state transition table for the system in the picture. Each system state consists of the individual states of *prod*, *cons* (should be only one state each), and of $Cell(1)$ and $Cell(2)$. So if your program for $Cell(i)$ has 3 states, the system will have 9 states. Notice that the channels, being synchronous, have neither buffering capacity nor memory, and therefore have only one state. Each transition between system states consists of a send by one process on one of the channels, and a receive by another process on the same channel. *(4p)*

(Part c) Show **two scenarios each** that result in the buffer holding 0 or 1 items (so four scenarios in total). The buffer holds an item if it has been sent by the producer but not yet consumed by the consumer. *(4p)*

Question 6. You are given a set S of n distinct positive integers. Your task is to write a process type $B(\mathit{int}\ i)$ to help find the largest integer, *max*, in S. The process $B(i)$ may have local memory, but no access to shared memory or to channels. It may interact with other processes only via a Linda tuple space (or "board") using the Linda primitives (summarised in the Appendix).

If $S = \{5, 8, 9\}$, a starter process (which you need not write) will run the processes B(5), B(8) and B(9). Your job is to ensure that when the processes $B(i)$ have each terminated, the board will show *max* (here, 9) as an element in an easily recognised tuple, specified below.

(Part a). Here, the board may have only a single tuple on it, of the form $\langle$"*bid*"$, i\rangle$, where "*bid*" is a constant string and i is an integer. Assume that the starter process will place $\langle$"*bid*"$, -1\rangle$ on the board. Write $B(i)$ to update the board as needed, and when all the $B(i)$'s have terminated, this lone tuple should be $\langle$"*bid*"$, max\rangle$. *(4p)*

How many updates does your program make? *(1p)*

(Part b). In (Part a), there are typically several updates of the board — somewhat like bids at an auction. In (Part b), we want the board to record the bids along the way.

Process $B(i)$ here should try to record a bid if i is larger than all the integers so far bid, and should terminate without bidding if it sees that an integer larger than i has already been bid.

Bids are to be recorded as triples of the form $\langle$"*bid*"$, r, i\rangle$, where "*bid*" is a constant string, $i \in S$, and r is the *rank* of the bid. The first bid recorded has rank 1, the second rank 2, and so on.

Thus, each $i \in S$ should appear in at most one triple. For any two distinct triples $\langle$"*bid*"$, r_1, i_1\rangle$ and $\langle$"*bid*"$, r_2, i_2\rangle$, you must have $r_1 \neq r_2$ and $i_1 \neq i_2$ and $r_1 < r_2$ iff $i_1 < i_2$.

If $S = \{5, 8, 9\}$, your program should have four possible computations, or "runs", 1 through 4 in the table below, which shows the tuples left on the board when the run has terminated.

Run	First bid	Second bid	Third bid
1.	$\langle$"*bid*"$, 1, 5\rangle$	$\langle$"*bid*"$, 2, 8\rangle$	$\langle$"*bid*"$, 3, 9\rangle$
2.	$\langle$"*bid*"$, 1, 5\rangle$	$\langle$"*bid*"$, 2, 9\rangle$	
3.	$\langle$"*bid*"$, 1, 8\rangle$	$\langle$"*bid*"$, 2, 9\rangle$	
4.	$\langle$"*bid*"$, 1, 9\rangle$		

If $B(5)$ makes the first bid, $B(8)$ and $B(9)$ can both post the second bid (run 1 and run 2). In run 3, the very first bid means $B(5)$ can terminate. In any run, the bid $\langle$"*bid*"$, r, i\rangle$ with the highest rank r should have $i = max$.

Your process B may place other tuples on the board if you need. Say what you need the starter to do. But aim to minimise (a) the number of kinds of tuples, (b) the number of interactions with the board, and (c) the amount of information held by each process. *(6p)*

How many comparisons does your program make? *(1p)*

——END of QUESTION PAPER——

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 declaration of $found$ is global. The declarations of i and j are local.

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 indicates 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.

A.2 LEXICOGRAPHIC ORDERING

Lexicographic ordering, or dictionary ordering, helps to locate an item quickly in a list. For example, the four words *bug*, *cat*, *bit* , *car*, are listed in a dictionary in the order *bit*, *bug*, *car*, *cat*. We look in the list for these words using these rules:

(1) Order by the first letter. (2) If the first letters are equal, order by the second letter. (3) If both the first and second letters are equal, order by the third.

Examples: (1) *bug* before *car* because $b < c$, even though $u > a$, (2) *bit* before *bug* because $b = b$ and $i < u$, even though $t > g$, (3) *car* before *cat* because $c = c$ and $a = a$ and $r < t$.

Apply the same idea to ordering states in Question 3. Use $p2 < p3 < p5$, then $q2 < q3 < q5$, then $Z < P < Q < PQ < QP$ where the last is the mnemonic way of saying $S_1 < S_2 < S_3 < S_4 < S_5$.

Hence the rule in Question 3, giving an ordering that will help you (and us) quickly find a state in your table, something you will need to do often as you answer the questions.

(pi, qj, S_k) before $(pi', qj', S_{k'})$ iff (1) $i < i'$ or (2) $i = i'$ and $j < j'$, or (3) $i = i'$ and $j = j'$ and $k < k'$

Just as *bir* is not a word, so too $(p2, q2, PQ)$ is not a reachable state from the start state, and so does not appear in the list. There are many such missing combinations, all representing unreachable states.

A.3 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.4 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 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——