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

Concurrent Programming TDA383/DIT390

Wednesday 21 Dec 2016, 2 pm—6 pm, in Samhällsbyggnad

Exam set by K. V. S. Prasad.
Supervised by Michal Palka (031-772 1079), who will visit one hour after the start.

- **Permitted materials (Hjälpmedel): Dictionary (Ordlista/ordbok)**
- Maximum you can score on the exam: 68p. This paper has six questions, Q1 through Q6, on pages 2 through 7. An Appendix, on page 8, gives additional notes on the logic, process and Linda notation used in this question paper.

  To pass the course, you need to pass each lab, and score at least 24p on the exam. Further:

  Exam grades: (CTH): grade 3: 24-40 p, grade 4: 41-53 p, grade 5: 54-68 p.
  (GU): grade G: 24-53 p, grade VG: 54-68 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.

- **Notes: PLEASE READ THESE**
  - **Do not get stuck for more time than you can afford on any question or part**.
  - Start each question on a new page.
  - The pseudo-code notation from the questions should suffice for your programs, but you can use Java, Erlang or Promela **provided you give your constructs the same semantics as the question requires.** The exact syntax you use is unimportant 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.
  - **Pseudo-code notation** for Q1 through Q4. The ***if*** construct below is also used in Q5 and Q6. For more general information on process constructs and logic, see the Appendix.
    1. Global variables are declared at the top of the program. Data declarations are of the form ***integer i*** or ***integer i := 1*** or ***boolean b := true***, giving the initial value, if any.

2. The statements of the processes are in columns headed by the names of the process. Indentation indicates sub-statements of compound statements.
3. All commands are numbered, but not control flow directions such as ***if*** and ***while***. Numbered statements are atomic, as are assignments and expression evaluations.
4. The statement ***await b*** is equivalent to ***block until b***.
5. A scenario is a list of the labels of the statements in the order of execution.
6. The ***if*** construct we use has multiple branches of the form :: *guard* $\rightarrow$ *commands*. A guard is a boolean or an input command; the former is *open* when it is true, and the latter when input is available. To execute an ***if***, an open guard is chosen non-deterministically and the commands following it executed. The last guard can be ***else*** or ***timeout***; this branch is chosen if and only if none of the previous guards is open.

Q1. Consider the following program.

integer n := 0	
p	q
p1: while n < 2 do p2: write(n)	q1: n := n+1 q2: n := n+1

(Part a). Construct a scenario for which the program gives the output 02 and halts. *(2p)*

(Part b). Construct a scenario for which the program produces no output and halts. *(2p)*

(Part c). How many times can 1 appear in the output if the scenario is fair? *(3p)*

Q2. This question asks you to write programs to solve the producer-consumer problem. In each part below, you must ensure that the producer does not try to add items to a full buffer, and that the consumer does not try to take items from an empty one.

Here is the program structure if PC is a monitor that implements the buffer. We have not shown the monitor itself, simply indicated that the buffer is a queue. Assume there is an operation append (item, buffer) that adds an item to the end of the queue, and a function head(buffer) that returns an item removed from the front of the queue.

finite queue of integer buffer := empty queue any synchronisation structures you need	
producer	consumer
while true do p1: d := produce p2: PC.put(d)	while true do q1: d:= PC.get q2: consume(d)

(Part a). Implement the monitor PC in the program above, with operations put(d) to add an integer d to the buffer, and get() to return an integer removed from the buffer. *(6p)*

(Part b). Re-do Part(a), using a protected object instead of a monitor. *(3p)*

Q3. 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 ***if*** commands p2, q2, p5 and q5. In the ***if*** commands, the test on S, and the subsequent assignment to it, all take place without interruption. The global variable S has 5 possible values: Z, P, Q, PQ, or QP.

```
type switch = {Z, P, Q, PQ, QP}
switch S := Z

p                                       q
    while true do                           while true do
p1:   non-critical section              q1:   non-critical section
p2:   if                                q2:   if
        :: S=Z → S:=P;                          :: S=Z → S:=Q;
        :: S=Q → S:=QP;                         :: S=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:   if                                q5:   if
        :: S=P  → S:=Z;                         :: S=Q  → S:=Z;
        :: S=PQ → S:=Q;                         :: S=QP → S:=P;
        :: else → skip                          :: else → skip
```

Below is part of the state transition table for an abbreviated version of this program, skipping p1, p4, q1 and q4 (the critical and non-critical sections).

A state transition table is a tabular version of a state diagram. The left hand column lists the states (where p and q are, and the value of S). The middle column gives the next state if p next executes a step, and the last column gives the next state if q next executes a step. 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 5 below, one or other of the processes may be blocked. There are 9 states in all.

	State = (pi, qi, S)	next state if p moves	next state if q moves
1.	(p2, q2, Z)	(p3, q2, P)	(p2, q3, Q)
2.	(p2, q3, Q)	(p3, q3, QP)	(p2, q5, Q)
3.			
4.	(p3, q2, P)	(p5, q2, P)	(p3, q3, PQ)
5.	(p3, q3, PQ)	(p5, q3, PQ)	no move
6.			
7.			
8.	(p5, q2, P)	(p2, q2, Z)	(p5, q3, PQ)
9.			

(Part a) Complete the state transition table. (we have left 4 lines blank). *(4p)*
(Part b) Prove from your state transition table that the program ensures mutual exclusion. *(2p)*
(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). *(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 p5-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. *(5p)*

Q4. Refer again to the program in Q3, reproduced here for convenience. Remember that the "await" commands (p3, q3) await either of two conditions, and that the ***if*** commands p2, q2, p5 and q5 are atomic.

type switch = {Z, P, Q, PQ, QP} switch S := Z	
p	q
while true do	while true do
p1: non-critical section	q1: non-critical section
p2: if	q2: if
:: S=Z → S:=P;	:: S=Z → S:=Q;
:: S=Q → S:=QP;	:: S=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: if	q5: if
:: S=P → S:=Z;	:: S=Q → S:=Z;
:: S=PQ → S:=Q;	:: S=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 be fuzzy, or to substitute wishful thinking for proof.

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.

Remember that $p \vee q$ ("*p* or *q*") is false iff (if and only if) both *p* and *q* are false, and that $p \rightarrow q$ ("*p* implies *q*") is false iff *p* is true and *q* is false.

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 both impossible. *(5p)*

(Part b). Let $M_q \equiv q4 \rightarrow (Q \vee QP)$. This is the symmetric counterpart of M_p. Assume $\Box M_p$ and $\Box M_q$ are both always true. Now prove that the program ensures mutual exclusion, i.e., $\Box\neg(p4 \wedge q4)$. *(5p)*

(Part c). Show that $\neg\Diamond\Box(p3 \wedge q3)$, i.e., the program cannot reach a deadlocked state.
Hint: Suppose $\Box(p3 \wedge q3)$. 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 $p3 \wedge q3$)? *(5p)*

Q5. In the program below, ***qin! 34*** means “output 34 on channel ***qin***”, and ***qend? n*** means “input a value from channel ***qend*** into variable ***n***”. All the processes have channels passed to them as parameters, and proctypes *End* and *Cell* declare private channels called *qe* and *qc*, respectively. The result is a growing network. For notes on the ***proctype***, ***run***, ***atomic*** and ***init*** constructs, see the Appendix. The ***fi*** (lines 17, 30) shows the end of the matching ***if*** (lines 13, 23).

```
 1  #define CAP 1                  //Used to set the buffer size of the channels
 2  chan qinit = CAP of int;       //Channel qinit carries integers, buffer size CAP
 3
 4  proctype Ints(chan qin) {      //Feeds a stream of integers into the system. Example stream shown.
 5     qin ! 34; qin ! 76; qin ! 23; qin ! 52; qin ! 3;
 6     qin ! 7;  qin ! 3;  qin ! 34; qin ! 35; qin ! 0;
 7  }
 8  proctype End(chan qend) {
 9     int n;
10     chan qe = CAP of int;       //Local channel declared
11
12     qend ? n;
13     if
14     :: n=0 −> skip              //The process skips to fi and terminates
15     :: else −> run Cell (n, qend, qe);  //Note the parameters!
16                run End (qe)     //Runs a new instance of itself
17     fi
18  }
19  proctype Cell(int c; chan qin, qout) {
20     int n;
21     chan q = CAP of int;
22     qin ? n;
23     if
24     :: n=0    −> printf(”%d\n”, c);  //Print c, pass on the 0, and terminate
25                  qout ! 0
26     :: n > c −> qout ! n;            //Pass on the n, and iterate
27                  run Cell (c, qin, qout)
28     :: else  −> run Cell (n, qin, q);
29                  run Cell (c, q, qout)
30     fi
31  }
32  init{
33     run Ints (qinit);
34     run End  (qinit)
35  }
```

(Part a). Draw pictures of the network (like a timeline) as the computation takes its first few steps. *Hint:* For the local channel qc, name its instances qc_1, qc_2 ...; similarly for qe. What does the program print by the end? Note that a 0 signals the end of input to the system. *(6p)*

(Part b). Does setting CAP to 0 change the output of the program? Setting CAP to 2? *(2p)*

(Part c). Alter the program so that 0 means ”print out current state, and take more input”. *(4p)*

Q6. The program on p. 7 uses the Linda tuplespace operations ***remove*** and ***post***. (See the Appendix). It mimics a game that starts with *MaxW* white balls (W) and *MaxB* black balls (B), bouncing around in a tumble dryer. The rules are $W+B \rightarrow G$ (when a white ball bumps into a black one, they disappear, and a grey ball (G) appears) and $G+B \rightarrow$ (when a grey ball bumps into a black one, both disappear).

There are *MaxR* processes *R* that try to apply the two rules, and a termination detector *T* that, when neither rule can be applied any more, shuts off the *R*s, prints the values *w, b, g*, and halts. The variables *w, b, g* say how many white, black and grey balls are present.

Notation reminders: Note that some ***if*** constructs have boolean guards (lines 33, 34) and some have ***remove*** command guards (lines 11, 13, 18, 20). Remember that a boolean guard is *open* when it is **true**, and a ***remove*** guard when it can produce input. The program also uses both ***else*** (line 35) and ***timeout*** (lines 16, 21, 23); these are always open, but chosen if and only if none of the previous guards of the ***if*** are open. The timeout is immediate if there is no input.

Rather than rely on indentation alone, we use ***fi*** (lines 17, 22, 24, 36) and ***od*** (lines 25, 37) as bracketing constructs to show the end of the matching ***if*** and ***do***. For notes on the ***proctype***, ***run***, ***init*** and ***atomic*** constructs, see the Appendix.

(Part a). Let MaxW, MaxB, MaxR all be set to 3. What values could the program print? For each value, show one sequence of ***remove*** and ***post*** actions that produces it, saying also which instance of *R* executes each action (name them R_1, R_2, R_3). *(4p)*

(Part b). Show that *2*w + g - b = 2*MaxW - MaxB* is true throughout the execution of the program. *(2p)*

(Part c). Assume that *T()* runs whenever the variables *w, g, b* change value. Now show that all the processes terminate. *Hint:* There are two cases, depending on whether *val* $<$ *0.* *(4p)*

(Part d). The program looks for *W* or *G* first and then for *B*. Suppose we restore symmetry between *W, B, G* by introducing this additional code between lines 22 and 23:

```
:: remove('B') ->                  //Look for a B
    if
    :: remove('W') ->              //and a W
        atomic{
          post('G');
          w--; b--; g++};          //Did W+B->G.  Return to 'while'
    :: remove('G') ->              //or a G
        atomic{b--; g--};          //Did G+B-> .  Return to 'while'
    :: timeout -> post('B')        //Got no W or G; replace B; return to 'while'
    fi
```

Find a combination of values for MaxW, MaxB, MaxR for which the program might not terminate. Show one sequence of ***remove*** and ***post*** actions that can loop, saying also which instance of *R* executes each action *(2p)*

```
 1 #define MaxW 12, MaxB 17, MaxR 12           //To experiment, change these
 2 type ('W'), ('B'), ('G');                   //A tuple represents a ball.
 3 int w := MaxW,  b := MaxB, g := 0;          //Variables to say how many white,
 4                                             //  black and grey balls are present.
 5 bool done:=false;                           //done:=true in lines 32 and 33, when game over.
 6
 7 proctype R() {
 8   while not done
 9   do
10     if
11     :: remove('W') ->                       //Look for a W
12         if
13         :: remove('B') ->                   //and a B
14              atomic{
15                 post('G');
16                 w--; b--; g++};             //Did W+B->G.  Return to 'while'
17         :: timeout -> post('W')             //Got no B; replace W, return to 'while'
18         fi
19     :: remove('G') ->                       //or look for a G
20         if
21         :: remove('B') -> atomic{b--; g--};        //Did G+B-> .  Return to 'while'
22         :: timeout -> post('G')             //Got no B; replace G, return to 'while'
23         fi
24     :: timeout -> skip                      //Got neither W nor G. Return to 'while'.
25     fi
26   od
27 }
28
29 proctype T(){
30   int val := 2*MaxW - MaxB;
31       if
32       :: (val < 0)   and (w=0) and (g=0) -> done:=true      //Guards are evaluated atomically
33       :: (val >= 0) and (b=0)             -> done:=true      //The if waits till a guard is true.
34       fi
35   printf("All done", w, b, g)
36 }
37
38 init {
39   for n:= 1 to MaxW do post('W');           //put in MaxW white balls
40   for n:= 1 to MaxB do post('B');           //and MaxB black ones
41   atomic{for n:= 1 to MaxR do run R();      //Start up MaxR instances of R
42              run T()
43   };
44 }
```

———-END of QUESTION PAPER———

A Appendix: LTL, Linda and process notations

A.1 LOGIC

1 We use iff to mean “if and only if”. We use $\neg$ for “not”, $\vee$ for “or”, $\wedge$ for “and”, and $\rightarrow$ for “implies”, while p iff q is a convenient abbreviation for $(p \rightarrow q) \wedge (q \rightarrow p)$. These have the obvious meanings, but note that $p \vee q$ (“p or q”) is false iff both p and q are false, and 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.

2 The proposition q_2 (process q is at label q_2) is true of state s iff process q is at q_2 in s.

3 We use Linear Temporal Logic (LTL), which is propositional logic with two added operators, $\Box$ and $\Diamond$. A formula ϕ of LTL holds for state s (or, *s satisfies* ϕ, written $s \models \phi$) if every path from s satisfies ϕ.

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 path π satisfies $\Box\phi$ (written $\pi \models \Box\phi$) if ϕ is true of the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ (written $\pi \models \Diamond\phi$) if ϕ is true of some state in π. Note that $\Box$ and $\Diamond$ are duals:

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

A.2 LINDA

In Linda programs, processes communicate via *notes* posted in a *space*. In this exam, the notes are constant strings. Processes interact with the space through two kinds of atomic actions.

post(t) Here *t* is a constant string. ***post(t)*** posts *t* in the space, and unblocks an arbitrary process among those waiting for a note with this string.

remove(x_1) Here the parameter is a constant string. The command ***remove***(x_1) removes a note that matches the parameter. If no matching tuple exists, the process is blocked. If there are several matching tuples, an arbitrary one is removed.

To remove actions can be attached a ***timeout*** clause, a sequence of statements. If there are no tuples that match the given pattern, the process *immediately* executes the code following timeout

A.3 Proctypes, the init process, the run command, and the atomic construct

You can explicitly declare processes by a line of the kind ***proctype p(integer i)*** giving the name of the process and its parameters. 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. An explicit ***init*** process starts the program.

These extensions give new expressive power. The ***run*** command means the number of processes in a program can change during execution. Processes can pass channels as parameters. This allows the network of channels between processes to change dynamically.

The commands enclosed within an ***atomic*** bracketing are all executed without interruption.

——-END of APPENDIX——