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

Concurrent Programming TDA383/DIT390

Saturday 22 Oct 2016 pm in HA/HB/HC

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 7, each carrying 12p, except Question 1, which carries 8p. An Appendix, on pages 9 and 10, 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; that is, about 25 mins. for Question 1, and about 35 mins. for each of the other questions. 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**.
  - 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 you give your constructs the same semantics as the question requires.** 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.

Question 1. Consider the following "Stop the loop" program.

integer n := 0 boolean flag := false	
p	q
p1: while flag = false	q1: while flag = false
p2: n := 1 - n	q2: if n=0
	q3: flag := true

(Part a). Construct a scenario for which the program terminates. *(2p)*

(Part b). What are the possible values of *n* when the program terminates? Construct a scenario for each. *(2p)*

(Part c). Construct a non-terminating scenario for the program. *(2p)*

(Part d). Is your non-terminating scenario fair? *(2p)*

Question 2. 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
loop forever	loop forever
p1: d := produce	q1: d:= PC.get
p2: PC.put(d)	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. *(5p)*

(Part b). Now rewrite your code using general semaphores instead of the monitor. The single statements `p2` and `q1` above will now be replaced by pre-protocol—op—post-protocol sequences, where op is either `append` or `head`, and the protocols use the semaphores. *(3p)*

(Part c). If your monitor were to be implemented using semaphores, how many semaphores would you need? What would each do? *(1p)*

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

Question 3. Here is the Manna-Pnueli algorithm to solve the critical section problem. It uses no synchronisation or concurrency primitives other than the `await` commands in `p3` and `q3`, and the atomic `if` commands in `p2` and `q2`. In the `if` commands, the test on the condition and the subsequent corresponding assignment execute as one uninterruptible command. Interpret `await C`, where `C` is a boolean, to mean "block until C".

integer wantp=0, wantq=0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: if wantq = -1	q2: if wantp = -1
then wantp := -1	then wantq := 1
else wantp := 1	else wantq := -1
p3: await (wantp $\neq$ wantq)	q3: await (wantp $\neq$ -wantq)
p4: critical section	q4: critical section
p5: wantp := 0	q5: wantq := 0

Below, with some entries missing (shown by dashes), is the state transition table (abbreviated "table") for an abbreviated version of the above program, skipping as usual the non-critical sections p1 and q1, and the critical sections p4 and q4, leaving only the pre-protocols p2 and p3, and q2 and q3, and the post-protocols p5 and q5.

We abbreviate `wantp` as `wp` and `wantq` as `wq`.

The left hand column lists the states (where *p* and *q* are, and the values of *wp* and *wq*). 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, but in states like s_{11}, one or both of *p* and *q* may be blocked. The program has a lot of states (17), but is actually quite simple.

	State = (pi, qi, wp, wq)	next state if p moves	next state if q moves
s_1	(p2, q2, 0, 0)	s_7 = (p3, q2, 1, 0)	s_2 =(p2, q3, 0, -1)
s_2	(p2, q3, 0, -1)	–	s_4 =(p2, q5, 0, -1))
s_3	(p2, q3, 0, 1)	s_{11} =(p3, q3, 1, 1)	s_5 =(p2, q5, 0, 1)
s_4	(p2, q5, 0, -1)	–	s_1 =(p2, q2, 0, 0)
s_5	(p2, q5, 0, 1)	s_{13} =(p3, q5, 1, 1)	s_1 =(p2, q2, 0, 0)
s_6	(p3, q2, -1, 0)	s_{14} =(p5, q2, -1, 0)	s_9 =(p3, q3, -1, 1)
s_7	(p3, q2, 1, 0)	s_{15} =(p5, q2, 1, 0)	–
s_8	–	–	s_{12} =(p3, q5, -1, -1)
s_9	(p3, q3, -1, 1)	s_{16} =(p5, q3, -1, 1)	blocked
s_{10}	–	–	blocked
s_{11}	(p3, q3, 1, 1)	blocked	s_{13} =(p3, q5, 1, 1)
s_{12}	(p3, q5, -1, -1)	blocked	s_6 =(p3, q2, -1, 0)
s_{13}	(p3, q5, 1, 1)	blocked	s_7 =(p3, q2, 1, 0)
s_{14}	(p5, q2, -1, 0)	s_1 =(p2, q2, 0, 0)	–
s_{15}	(p5, q2, 1, 0)	s_1 =(p2, q2, 0, 0)	s_{17} =(p5, q3, 1, -1)
s_{16}	–	s_3 =(p2, q3, 0, 1)	–
s_{17}	(p5, q3, 1, -1)	s_2 =(p2, q3, 0, -1)	blocked

(Part a) Complete the table (we have left 10 entries blank). *(5p)*

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

(Part c) Prove from your table that the program does not deadlock. *(1p)*
(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 S of all states that must lead to a $p5$-state in zero, one or more moves. First, $S :=$ the set of all $p5$-states. E.g., $s_{15} \in S$. Next, $S := S \cup \{s_9\}$, as s_9 must lead to a p_5-state. Then $S := S \cup \{s_9\}$, etc. This way, you will find many states that must lead to S. List these states first. *(3p)*

For the remaining few 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)*

Question 4. Refer again to the program in Question 3, reproduced here for convenience.

integer wantp=0, wantq=0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: if wantq = -1	q2: if wantp = -1
then wantp := -1	then wantq := 1
else wantp := 1	else wantq := -1
p3: await (wantp $\neq$ wantq)	q3: await (wantp $\neq$ -wantq)
p4: critical section	q4: critical section
p5: wantp := 0	q5: wantq := 0

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 $p2$ as a logical proposition to mean "process `p` is at `p1` or `p2`", $p3$ to mean "process `p` is at `p3`", and $p5$ to mean "process `p` is at `p4` or `p5`". We also use the state names from Question 3.

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 $N \equiv (p2 \text{ iff } (\texttt{wantp} = 0)) \wedge (q2 \text{ iff } (\texttt{wantq} = 0))$.

(Part a). Show that N is invariant, i.e., that $\Box N$ holds for the start state s_1. *(2p)*
(Part b). Show $\Box M$ by induction. First show that M holds at the start state s_1.

Then show that every transition preserves M. The only transitions that start with M and end with $\neg M$ are those starting from $p_3 \wedge q_5$ or $p_5 \wedge q_3$. By symmetry, we need deal only with $p_5 \wedge q_3$. Use Part a to show that in this state, each combination of `wantp` and `wantq` allows precisely one of p and q to proceed. You will need to consider the two kinds of states that can transition to a $p_5 \wedge q_3$ 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)*

Question 5. Below is a program in pseudo-code with channels. NOTE: Sift declares a private channel called q. Filter has two channels, both passed to it as parameters. Sift's lone channel is also passed to it as a parameter. The result is a growing network.

(Part a). Suppose we use the program with CAP=0, MAX = 100 and START = 2. Draw pictures of the network as the computation takes its first few steps. What does the program print? *(Hint: it does not print primes).* *(6p)*

(Part b). What happens if you set START=3? START=5? *(2p)*

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

(Part d). Does changing CAP have an effect? *(1p)*

(Part e). Change the program to print primes instead. *(2p)*

```
chan capacity(CAP) of integer qi;

proctype Ints(){
    for n := START to MAX do qi ! n
}

proctype Filter(int p; chan qin; chan qout) {
     int n;
     do forever
           n <= qin;             //input a number from qin, store in n.
           if n mod p = 0
           then qout <= n        //p divides n; output n on qout
           else skip
     od
}

proctype Sift(chan qin) {
    int p;
    chan capacity(CAP) of integer q;
           p <= qin;             //input a number from qin, store in p.
           print p;
           run Filter(p, qin, q);
           run Sift (q)
}

init{run Sift(qi); run Ints()}
```

Question 6. On p. 7 is a program in pseudo-code that uses the Linda tuplespace operations remove, read and post. (See Appendix A.1, and Appendix A.3). The program finds the prime numbers up to MAX. The square root of MAX is supplied as SQRT. The program uses several kinds of integer tokens in the space, with the labels '`cand`' and so on. At the end, the prime numbers are those remaining in the space as `('cand', 2)`, `('cand', 17)` and so on. Your task is to explain how the program works.

We use two programming conveniences.

1 `remove` and `read` actions use `eval(n)` to mean the value of variable `n`. Suppose n=13. Then the pattern `read('start', n)` will match the tuple `('start', 14)`, resetting `n` to 14, whereas `read('start', eval(n))` will only match the tuple `('start', 13)`.

2 `read` and `remove` actions are here allowed to attach a `timeout` clause, a sequence of statements that must end with `goto` *label*. If there are no tuples that match the given pattern, the process executes the code between `timeout` and `goto`, and then jumps to the statement *label*.

(Part a). How are the tuples containing `'start'` and `'done'` used? *(3p)*

(Part b). Why not have `('start')` and `('done')` tokens as singletons, instead of pairs like `(start, 2)` and `(done, 20)` and so on? *(3p)*

(Part c). How does the program terminate? *(3p)*

(Part d). Can you tidy up the termination so that no process is left hanging? *(3p)*

```
#define MAX 45, SQRT 7;
type ('cand', int), ('divisor', int), ('curr', int), ('start', int), ('done', int);

proctype Worker(int n) {
int curr;
     do forever
           remove ('start', eval(n));
           read('curr', curr);
           if n = curr then skip
           else if n mod curr = 0             //curr divides n
                then remove('cand', eval(n));
                     break;                   //out of do loop
                fi                            //else skip
           fi;
           post('done', n);
     od;
     post('done', n);
     do forever
           remove ('start', eval(n));
           post('done', n);
     od
}

proctype Master(){
      int curr, i;
      do forever
           remove('divisor', curr) timeout goto alldone;
           post('curr', curr);
           for i:= 2 to MAX do
                post('start', i);
           for i:= 2 to MAX do
                remove('done', i);
           remove ('curr', curr);
      od;
alldone:
}

init {     int i;
           for i:= 2 to MAX do
                post('cand', i);
           for i:= 2 to SQRT do
                post('divisor', i);
           for i:= 2 to MAX do
                run Worker(i);
           run Master()
}
```

——-END of QUESTION PAPER——

A Appendix: Pseudo-code, LTL and Linda notations

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

1. 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]`.

2. The statements of the processes are often in columns headed by the names of the process. If several processes `p(i)` have the same code, parameterised by `i`, they are given in one column. Indentation indicates sub-statements of compound statements.

3. All commands are numbered, but not control flow directions such as `loop forever` and `repeat`. If a continuation line is needed, it is left un-numbered or numbered by an underscore `p_`. Numbered statements are atomic. Assignments and expression evaluations are atomic.

4. The statement `await b` is equivalent to either `block until C` or to `while not b do nothing`, a *busy wait*. Which interpretation is meant will be pointed out in any question using `await`. Under the first interpretation, the system may *deadlock* (everyone is blocked); under the second, the system may *livelock* (everyone busy-waits). The only difference is in CPU-cycles. Both states show mutual impediment to progress, or circular waiting.

5. 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`.

6. A scenario is a list of 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

1. 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.

2. We extend Ben-Ari's notation for channels, allowing `channel capacity(n) of boolean forks[5]`. This declares an array of channels, `fork[0]` through `fork[4]`, each a channel of buffer capacity `n`, carrying boolean values. So `n=0` specifies a synchronous channel, and `n=5` specifies an asynchronous channel with buffering capacity 5. For theoretical discussion, we can also permit n to be `infinite`. The capacity declaration `capacity(0)` can be dropped, (i.e. in that case, assume `n=0` and therefore synchrony).

3. Input commands are allowed to attach a `timeout` clause, a sequence of statements that must end with `goto` *label*. If the channel is empty when an input command runs, the process executes the code between `timeout` and `goto`, and then jumps to the statement *label*.

A.2 LOGIC

1. The symbols used here for the operators of propositional logic are: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", and $\rightarrow$ for "implies", while p iff q (i.e., p if and only if q) is a convenient abbreviation for $(p \rightarrow q) \wedge (q \rightarrow p)$. 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 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.

2. A proposition such as q_2 (process q is at label q_2) is true of a 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.3 LINDA

In Linda programs, processes communicate via *tuples* posted in a *space*. The first element of a tuple is often a constant string, saying what kind of tuple it is. Processes interact with the space 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` in the space, 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 in the space.

We allow two extensions of the input constructs `remove` and `read`:

1. `remove` and `read` actions can use `eval(n)` to mean the value of variable `n`. Suppose n=13. Then the pattern `read('start', n)` will match the tuple `('start', 14)`, resetting the value of `n` to 14, whereas `read('start', eval(n))` will only match the tuple `('start', 13)`.

2. To `remove` and `read` actions can be attached a `timeout` clause, a sequence of statements that must end with `goto` *label*. If there are no tuples that match the given pattern, the process executes the code between `timeout` and `goto`, and then jumps to the statement *label*.

——-END of APPENDIX——