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

Concurrent Programming TDA383/DIT390

Saturday 25 October 2014, 14:00 to 18:00.

K. V. S. Prasad, tel. 0736 30 28 22

- Maximum you can score on the exam: 70 points. This paper has four pages, with seven questions, each carrying between 8 and 12 points.

  To pass the course, you need to pass each lab, and get at least 24 points on the exam. Further requirements for grades (Betygsgränser) are as follows:

  CTH (total on exam + labs): grade 3: 40 - 59 pts, grade 4: 60 - 79 pts, grade 5: 80 - 102 pts.
  GU (on exam): Godkänd 24-53 pts, Väl godkänd 54-70 pts

- Results: within 21 days.

- **Permitted materials (Hjälpmedel):**

  - Dictionary (Ordlista/ordbok)

- **Notes: PLEASE READ THESE**

  - Time planning: if you allow 3 minutes per point, you will 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**. On no account can you receive, for any question or part, more than the points assigned to it.
  - Start each question on a new page.
  - Answers in English only, please. Our graders do not read Swedish.
  - A SUMMARY follows of Ben-Ari's pseudo-code notation, used in this question paper.
  - Ben-Ari's pseudo-code should suffice for your programs, but you can use Java, Erlang or Promela if you think they are appropriate. 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, add an explanation of your notation.
  - **If the correctness of your answer is not clear from inspection, you must justify it.** If you feel you need to, make reasonable assumptions beyond those given, but state them clearly. If your solution only works under certain conditions, state the conditions.
  - Be as precise as you can. 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!

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.

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.

———-END of SUMMARY——

Question 1. Consider the following program:

boolean flag := false, turn := false	
p	q
p1: while not flag p2: turn := not turn	q1: while not flag q2: if not turn q3: flag := true

(Part a). Construct two scenarios for which the program terminates, one where *turn* is true at the end, and one where it is false. *(3+3p)*

(Part b). Find a weakly fair scenario for which the program does not terminate. *(3p)*

Question 2. **(Part a)**. Using synchronous channels, develop a program to sort n numbers, where $1 \leq n \leq 100$. Assume that the numbers to be sorted are all distinct, positive, non-zero integers. The n numbers are fed into a channel c_0, and a sentinel value 0 is fed into c_0 to signal the end of input.

Build a chain of n processes P_i, for $1 \leq i \leq n$, and channels c_i for $0 \leq i \leq n$. Each process P_i has channel c_{i-1} to its left and channel c_i to its right. Process P_i takes input from channel c_{i-1} and delivers output via channel c_i to process P_{i+1} to its right. Process P_1 takes from c_0 the input numbers to be sorted. If you need it, write a sink process to input numbers from c_n and throw them away.

When the program terminates, the input numbers should be stored one per process in local memory. Let d_i be the number held by process P_i. Then for i and j such that $i < j$, it should be that $d_i < d_j$. Write the code for the processes P_i to sort the input as described. *(8p)*

(Part b). Adapt your program to work with asynchronous channels. Assume the buffering capacity of each channel is 100. *(3p)*

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.

integer S := 0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: if even(S) then S:=4 else S:=5	q2: if S $<$ 4 then S:=3 else S:=7
p3: await (S$\neq$ 1 $\wedge$ S$\neq$5)	q3: await (S$\neq$ 6 $\wedge$ S$\neq$7)
p4: critical section	q4: critical section
p5: if S $\geq$4 then S:=S-4 else skip	q5: if odd(S) then S:=S-1 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 10 states in all.

	State = (pi, qi, Svalue)	next state if p moves	next state if q moves
1.	(p2, q2, 0)	(p3, q2, 4)	(p2, q3, 3)
2.	(p3, q2, 4)	(p5, q2, 4)	(p3, q3, 7)
3.	–	–	–
4.	–	–	–
5.	(p3, q3, 7)	(p5, q3, 7)	no move
6.	–	–	–
7.	–	–	–
8.	–	–	–
9.	–	–	–
10.	(p2, q2, 2)	(p3, q2, 4)	(p2, q3, 3)

(Part a) Complete the state transition table. *(6p)*
(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)*

Question 4. Refer again to the program in Question 3. This time, you must argue from the program, not from the state transition table (though you may seek inspiration from it!).

(Part a). Show that $(p3 \wedge q3) \rightarrow (S = 5 \vee S = 7)$ 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=5). Prove that if $p3 \wedge q5$, then p cannot move until after q executes $q5$. (That is, mutual exclusion holds). *(2p)*

(Part c) Assume that $p3 \wedge q1 \rightarrow (S = 4)$ is invariant, and that q always loops in q1. Then prove that $p3 \wedge q1 \rightarrow \Box\Diamond p5$. *(4p)*

Question 5. **(Part a)**. Solve the readers and writers problem with a protected object. There are two classes of processes that compete for access to the object: *readers* and *writers*. Readers have to exclude writers but not other readers. Writers have to exclude both readers and other writers. Write the code for the protected object and for the reader processes and the writer processes. *(5p)*

(Part b). Suppose a process P is waiting on a barrier to an operation on the protected object. How does the system know when P should be unblocked? *(3p)*

Question 6. Consider the program below, with binary semaphores as the sole communication method.

binary semaphore SA:=0, SB := 0, SC := 1		
process A	process B	process C

loop forever
wait(SB);
print("A");
signal(SC)
	loop forever
	wait(SC);
	print("B");
	signal(SA)
		loop forever
		wait(SA)
		print("C");
		signal(SB)

(Part a). What does the program print? *(3p)*

(Part b). Show that when any process is printing, all three semaphores are zero. Where are the other two processes at that time? *(3p)*

(Part c). Suppose we had declared the semaphores to be general semaphores instead, but with the same initial values as above. Would the program printout be different? *(2p)*

(Part d). The given program always initialises SC to 1 and the other semaphores to zero. Suppose we would like to initialise all the semaphores to zero, and then randomly signal one of them to get the above program started. Write processes to run in parallel with A, B and C, to achieve this without a random number generator. You may use as many processes and semaphores as you like, but no other communication method. *(4p)*

Question 7. **(Part a)**. Implement a bounded buffer of capacity n, where $n \geq 1$, in Linda. Assume the products to be stored in the buffer are all alike.

Write a producer process P and a consumer process C. The producer puts products v into the buffer, and has to wait if (and only if) the buffer is full—that is, when there are n products in the buffer. The consumer takes products from the buffer, and has to wait if (and only if) the buffer is empty—that is, when there are no products in it.

Your program will get most credit if the processes wait *only* for the conditions described above, and if the processes P and C maintain as little internal information as possible. But you can post as much other information as you need into the space. *(7p)*

(Part b). Generalise your program to allow multiple instances of process P and multiple instances of process Q. That is, to allow multiple producers (all running the same code P) and multiple consumers (all running the same code C). *(3p)*

-

——-END of QUESTION PAPER——-