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

Concurrent Programming TDA382/DIT390

Thursday 24 October 2013, 14:00 to 18:00.

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

- Maximum you can score on the exam: 72 points. This paper has four pages, with seven questions, each carrying 12 points. Choose any six questions to answer. If you attempt all seven questions, we will ignore the question on which you score the least 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 - 104 pts.
  GU (on exam): Godkänd 24-53 pts, Väl godkänd 54-72 pts

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):**
  - Dictionary (Ordlista/ordbok)
- **Notes: PLEASE READ THESE**
  - Time planning: you have 40 minutes for each of the six questions you will answer. Do not get stuck for more time than you can afford on any question or part.
  - 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, JR, or Erlang 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.
  - Don't just write down answers; say why they are correct. This explanation is not needed in cases where it is clear how to check that the answer is correct.
  - If a question does not give you all the details you need, make reasonable assumptions, 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 declaration of *n* is 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 <= e` means that the value of the expression e 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. **(Part a)** Consider the following program:

integer n := 1	
p	q
p1: while n < 1	q1: while n ≥ 0
p2: n := n+1	q2: n := n-1

Construct scenarios in which

(1) process p above loops exactly three times, *(3p)*

(2) both processes p and q above loop infinitely often. *(3p)*

(Part b) Define the operations on a binary semaphore. *(3p)*

(Part c) What are condition variables in monitors? Why are they needed? *(3p)*

Question 2. Consider the following attempt at solving the critical section problem.

boolean wantp := false; wantq := false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp := true	q3: wantq := true
p4: critical section	q4: critical section
p5: wantp := false	q5: wantq := false

(Part a) Construct the state diagram for an abbreviated version of this program, or as much of it as needed to show that the program cannot ensure mutual exclusion. *(6p)*

(Part b) Prove informally (but precisely) that the program does not deadlock. *(6p)*

Question 3. **(Part a)** A small construction company builds only one house at a time, and must sell it before they can start another. So their employees (masons, plumbers, salesmen, etc.), must all finish their work on the present house before anyone can start on the next house. If any worker needs more time, their colleagues just have to wait till they are done. (Such coordination of activity in rounds is called barrier synchronisation).

Write a monitor to help the employees coordinate their activities. *(8p)*

(Part b). If you have n CPU's, barrier synchronisation can be used to speed up the hunt for the root r of a monotonically increasing function f. Suppose we know that $f(a_0) < 0$ and $f(a_{n+1}) > 0$. Then we know that the r lies between a_0 and a_{n+1}. For the first round of the hunt, split the interval $[a_0, a_{n+1}]$ into $n+1$ bits $[a_i, a_{i+1}]$ for $i = 0..n$, and evaluate f in parallel at the n points $a_i, i = 1..n$. Then find i such that $f(a_i) < 0$ and $f(a_{i+1}) > 0$, and the next round looks recursively in $[a_i, a_{i+1}]$.

First, assume that the time to compute $f(x)$ is independent of x. In what way is this application different from that of Part a? *(2p)*

Now assume that the time to compute $f(x)$ depends on x. Can you use this to speed up the hunt for the root? *(2p)*

Question 4. A railway network consists of a set of cities, numbered 1 through N, and a set of tracks. Each track $(n1, n2)$ connects exactly one city, $n1$, to exactly one city, $n2$.

Write a message-passing program using channels to take a railway network of $N \geq 2$ cities and find if it allows travel from city k to city l. If it can, your program should print out "ok", otherwise it may either loop or hang or print out "no path".

Hint (Use if you wish. You can also use your own method): Use channels c_i, where $i = 1..N$, to represent the cities, and let each track (i, j) be represented by a process $p(i, j)$.

(Part a) Write your program, including all the processes it needs. *(6p)*
(Part b) Modify your program to print out the length of the first path it finds (i.e., the number of tracks in it). For any other paths it finds, it should print out the length if it is smaller than any length found so far. *(4p)*
(Part c) Will your program work with both synchronous and asynchronous channels? *(2p)*

Question 5. Write a Linda program to print out the prime numbers between 2 and N. The primes need not be printed out in ascending order. Assume every process you create runs on its own CPU. You can use as many (identical) CPU's as you wish. Your goal is to find the primes in the shortest possible time. The program should terminate cleanly with each process terminating.

You may start with a Linda space containing the tuples you need (i.e., you don't have to write a process to generate these tuples, just say what tuples you are starting with).

Hints (Use if you wish. You can also use your own method):
1. Once we discover that c is a composite number, it is a waste of time to again check if any integer n divides c.
2. You may find it useful to process candidate primes in sequence. This can be done by starting with the pairs (2,3), (3,4), (4,5) ... (N-1, N), (N, 'top'). These form in effect a linked list. Filter processes can work their way up this list. When a process discovers that 4 is not a prime, the start of the list is changed to (2,3), (3,5) .. and so on. *(12p)*

Question 6. **(Part a)** A concurrent mergesort algorithm recursively divides an array into halves, sorts the two halves concurrently and merges the results together. Here is a program for one step of the recursion, using two binary semaphores, S1 and S2.

integer array A binary semaphore S1 := (0, ∅), S2 := (0, ∅)		
sort1	sort2	merge
p1: sort 1st half of A p2: signal(S1)	q1: sort 2nd half of A q2: signal(S2)	r1: wait(S1) r2: wait(S2) r3: merge halves of A

Modify the above program to use a single general semaphore instead of S1 and S2. *(3p)*

(Part b) Five philosophers sit in a circle, each executing an infinite "think - eat" loop. They have five forks, one each between every pair of philosophers. To eat, each philosopher must wait till (s)he has both the forks to their right and left. The problem is to program a fork sharing protocol that avoids deadlock and starvation, while allowing maximum parallelism. Only the availability of forks, no other rule, should detain a philosopher when they wish to eat.

(1) Solve the dining philosophers problem using separate processes for each philosopher and a protected object for the forks. *(5p)*

(2) What invariants hold for your program? *(2p)*

(3) What do you need to assume to avoid starvation? *(2p)*

Question 7. Here is Peterson's algorithm to solve the critical section problem without special instructions.

boolean wantp := false; wantq := false integer last := 1	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp := true	q2: wantq := true
p3: last := 1	q3: last := 2
p4: await wantq = false or last = 2	q4: await wantp = false or last=1
p5: critical section	q5: critical section
p6: wantp := false	q6: wantq := false

(Part a). Show that

$$(p4 \wedge q5) \rightarrow (wantq \wedge last = 1)$$

is invariant. Then by symmetry, it follows that so is

$$(p5 \wedge q4) \rightarrow (wantp \wedge last = 2).$$

Use these two formulae to show that mutual exclusion holds. *(6p)*

(Part b). Prove the following formulae:

$$p4 \wedge \Box\neg p5 \rightarrow \Box\Diamond(wantq \wedge (last \neq 2))$$

$$\Diamond\Box(\neg wantq) \vee \Diamond(last = 2)$$

(6p)

-

——-END of QUESTION PAPER——-