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

Concurrent Programming TDA381/DIT390

Tuesday 23 October 2012, 14:00 to 18:00.

K. V. S. Prasad, tel. 0730 79 43 61

- 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, for total points on exam + labs:

  Chalmers: grade 3: 40 - 59 points, grade 4: 60 - 79 points, grade 5: 80 - 104 points.
  Chalmers ETCS: E = 40–47, D = 48–59, C = 60–75, B = 76–87, A = 88-104
  GU: Godkänd 45-79 points, Väl godkänd 80-104 points

- 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.
  - 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.
  - If a question does not give you all the details you need, make reasonable assumptions, but state them clearly. If your solution 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:

integer n := 0 boolean flag := false	
p	q
	q1: while n=0
p1: while flag = false	– Do nothing
p2: n := 1-n	q2: flag := true

(Part a). What are the possible values of *n* when the program terminates? Give an example scenario for each value. *(6p)*

(Part b). Does the program always terminate? *Hint:* Try to find a scenario that loops even if the scheduler is fair. *(6p)*

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: wantp := true	q2: wantq := true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp := false	q5: wantq := false

Construct the state diagram for an abbreviated version of this program, or as much of it as you can. How would you use it to show that mutual exclusion holds for the program? *(12p)*

Question 3. **(Part a)**.Assume that the operating system allows you to group a sequence of actions as atomic (i.e., either they all happen or none of them do), and also provides the type "process". Draw a state diagram of a process as seen by the operating system, showing transitions between running, sleeping, etc. . *(2p)*

(Part b). Implement a general semaphore on top of such an operating system. What data does it hold? Define the operations on it. *(4p)*

(Part c). Suppose you have one producer and one consumer, communicating via a finite buffer. The producer has to wait if the buffer is full, and the consumer has to wait if the buffer is empty. Write programs for the producer and the consumer, using general semaphores. Define the datatype for the buffer and initialise whatever semaphores you use. On the buffer, assume only the pre-defined operations `append` and `take`. *(6p)*

Question 4. Five philosophers sit in a circle. They have five forks, one each between every pair of philosophers. A philosopher can eat only when (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. Here is a solution with monitors.

```
monitor ForkMonitor
  integer array [0..4] fork := [2, ..., 2]
  condition array [0..4] OKtoEat
  operation takeForks(integer i)
        if fork[i] $\neq$ 2
              waitC(OKtoEat[i])
        fork[i+1] := fork[i+1] -1
        fork[i-1] := fork[i-1] -1
  operation releaseForks(integer i)
        fork[i+1] := fork[i+1] +1
        fork[i-1] := fork[i-1] +1
        if fork[i+1] = 2
              signalC(OKtoEat[i+1])
        if fork[i-1] = 2
              signalC(OKtoEat[i-1])
```

```
philosopher i
```

```
  loop forever
p1:   think
p2:   takeForks(i)
p3:   eat
p4:   releaseForks(i)
```

To keep the code simple, we have not written `fork[i+1 mod 5]` and so on, assuming instead that the operations + and - on the index of `fork` wrap around.

(Part a). Let `E` be the number of philosophers eating, and `F` = $\Sigma_{i=0}^{4}$`fork[i]`. Then `F = 10 - 2*E`. Assume that the formula `not empty(OKtoEat[i]) -> (fork[i] < 2)` is invariant, and prove by contradiction that it never happens that every philosopher is enqueued waiting to eat. *(6p)*

(Part b). Let `eating[i]` stand for "philosopher i is at p3". Prove that the formula `eating[i] -> (fork[i]=2)` is invariant. *(6p)*

Question 5. Write a program to solve the dining philosophers problem using separate processes for each fork and each philosopher. See question 4 for a description of the problem. Assume communication between the processes is by message passing over channels. Your solution must have each channel connected to exactly one sender and one receiver. *(12p)*

Question 6. Write a Linda program to print out the value of the biggest integer in a set of N distinct positive integers. You must allow an arbitrary number W of worker processes, each running the same code, and a printer process. Assume that the space has the following notes at the start:

value notes: N notes of the form `('v', n)`, where `n` is a positive integer.
count note: a tuple `('count', N)`, recording the number of value notes.
max note: a tuple `('max', -1)`
worker note: a tuple `('W', W)`, recording the number of active workers.

You may add other notes if you wish. Write the code that each worker process will run, and the printer process that prints out this value at the end. Each worker should use only limited local memory. The program as a whole should allow as much parallelism as possible. *(12p)*

Question 7. Here is a solution to the critical section problem using an exchange command, which atomically swaps the values of two variables.

boolean C := true	
p	q
boolean Lp := false	boolean Lq := false
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p-: repeat	q-: repeat
p2: exchange(C, Lp)	q2: exchange(C, Lq)
p3: until Lp	q3: until Lq
p4: critical section	q4: critical section
p5: exchange(C, Lp)	q5: exchange(C, Lq)

As usual, assume that `p4` and `q4` terminate, but do not assume that `p1` or `q1` do so. Assume that each of `p1, p4, q1`, and `q4` take some time, but assume nothing about their relative runtimes.

Assume further that `p` and `q` run on separate (but identical) processors. So the segments `p-` to `p3` and `q-` to `q3` are busy-wait loops.

(Part a). Even though the processes are always active, there is a sense in which the program can deadlock. Define this kind of deadlock, and prove that the program above is free from it. *Hint:* Look for an invariant involving the three variables `C, Lp` and `Lq`. *(6p)*

(Part b). Show that `box (p2 -> diamond p5)`. That is, whenever process p is at p2, it will progress eventually to p5. *Hint:* Assume it doesn't, then show that q must progress, and therefore show a contradiction. Remember the busy-wait assumption. *(6p)*

-

——-END of QUESTION PAPER——-