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

# Concurrent Programming TDA381/DIT390

Saturday 22 October 2011, 8.30 – 12.30, M.

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

- Total points on the exam: 100

  Grading scale (Betygsgränser):

  Chalmers: 3 = 40–59 points, 4 = 60–79 points , 5 = 80-100 points
  Chalmers ETCS: E = 40–47, D = 48–59, C = 60–75, B = 76–87, A = 88-100
  GU: Godkänd 45-79 points, Väl godkänd 80-100 points

- Results: within 21 days.

- **Permitted materials (Hjälpmedel):**

  - Dictionary (Ordlista/ordbok)

- **Notes:**

  - Read through the paper first and plan your time. There are 6 questions, the first carrying 20 points, and the rest carrying 16 each. If you give each point 2 mins., you will have plenty of buffer time. So one plan might be to give the first question 40 mins., and the rest 30 mins. each, leaving the remaining 50 mins. for revision, corrections and tidying up.
  - Answers in English only, please. Our graders do not read Swedish.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each question on a new page.
  - 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 just put in an explanation of your notation.
  - 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. Points will be deducted for solutions which are unnecessarily complicated.
  - DON'T PANIC!

**Question 1.** **(Part a)**. For the program below, "mutual exclusion" means `p(1)` and `p(2)` should not simultaneously enter their respective critical sections. Assume that a process that enters its CS eventually exits it, but no such assumption holds for NCS.

```
proc p(n: integer) =
          loop forever
                   -- non-critical section (NCS)
                   pre-protocol
                   -- critical section (CS)
                   post-protocol
          end loop;
end proc;

main = par p(1);  p(2) end par
end main.     -- runs processes p(1) and p(2) in parallel
```

Using only ordinary variables (i.e. no monitors, semaphores, channels, or other concurrency constructs), write protocols that ensure mutual exclusion but risk deadlock or individual starvation respectively. How are these problems overcome by introducing atomic actions? *(12p)*

**(Part b).** What is the model of "process" you need to do all this (I.e., what states does a process go through and when does it transit between them)? *(2p)*

**(Part c).** Why do we need the notion of interleaving? Why do we think it's a bad idea to make assumptions about how long a process will take to do something? *(6p)*

**Question 2.** Here is a solution to the critical section problem using an exchange command, which atomically swaps the values of two variables.

```
C: integer = 1;

proc p(n: integer) =
          L: integer = 0;
          loop forever
                  --proc n is in its Non-Critical Section
                  repeat                                          --s1
                        exchange(L,C)                             --s2
                  loop until L=1;
                  --proc n is in its Critical Section
                  exchange(L,C);                                  --s3
          end loop;
end proc;

main = par p(1); p(2) end par
end main.
```

**(Part a).** Prove that mutual exclusion holds, using state diagrams. The labels s1, s2 and s3 suggest which statements you might want to include in an abbreviation of the program. *(12p)*

**(Part b).** What formula in Linear Temporal Logic would say the program is "free from individual starvation"? *(4p)*

**Question 3.** A semaphore S has a value $S.V \geq 0$ and a set $S.L$ of blocked processes. A process doing `wait(S)` is put in $S.L$ if $S.V = 0$ and decrements $S.V$ otherwise. A process doing `signal(S)` increments $S.V$ if $S.L$ is empty, and unblocks one process from $S.L$ otherwise. Usually, $S.L$ does not need to be mentioned, and $S.V$ is written just $S$.

**(Part a).** Consider the following program:

```
semaphore S = 1, T=0;
proc p =
    wait(S); print(``p'');  signal(T)
end proc; -- p

proc q =
    wait(T); print(``q''); signal(S)
end proc; -- q

main =
    par p; q end par
end main.
```

What are the possible outputs of the program? *(3p)*

What are the possible outputs if we erase `wait(S)` from process `p`? *(3p)*

What are the possible outputs if we erase `wait(T)` from process `q`? *(3p)*

**(Part b).** In what ways are monitors better than semaphores as concurrency primitives? *(3p)*

What is a condition variable? Define the operations on it. Why do they differ from the operations on a semaphore? *(4p)*

**Question 4.** Consider the following monitor solution to the dining philosophers problem. The array "fork" counts the number of free forks available to each philosopher; a philosopher can eat only when two forks are available. Parts of the code, marked ****, are missing.

```
monitor ForkMonitor =

    integer array(0..4) fork = [2 .. 2];
    condition array(0..4) of OK_to_Eat;

    operation take_Fork(integer i) =
          ****
          fork((i+1) mod 5) = fork((i+1) mod 5) - 1;
          fork((i-1) mod 5) = fork((i-1) mod 5) - 1
    end take_Fork;

    operation release_Fork(integer i) =
          fork((i+1) mod 5) = fork((i+1) mod 5) + 1;
          fork((i-1) mod 5) = fork((i-1) mod 5) + 1;
          ****
    end release_Fork;

end monitor -- ForkMonitor;

proc phil(i) =
     loop
         think;
         takeForks(i);
         eat;
         releaseForks(i)
     forever
end proc -- phil(i)
```

**(Part a).** Fill in the missing code to achieve a solution that guarantees mutual exclusion over the forks, and does not deadlock. *Hint:* The missing code deals with condition variables. *(8p)*

**(Part b).** What monitor semantics did you assume for your code above? Immediate resumption - the newly awoken process has priority over the signalling process- or some other semantics? Is the correctness of your answer affected if we change the monitor semantics? *(2p)*

**(Part c).** Let `eating[i]` mean `phil[i]` is eating. Then show that `eating[i]`$\rightarrow$`forks[i]=2` is invariant. *(6p)*

**Question 5.** **(Part a).** Program a buffer (of integers) as a server, which the producer and consumer processes invoke via synchronous message passing along named channels. I.e., the processes do not address each other, but the channels. Begin with a server process that holds an entire finite buffer as an array, and accepts "produce" and "consume" messages when it can. Assume you have available the necessary functions to manage the buffer data structure. Also write the user processes, i.e., the producer and consumer. *(5p)*

**(Part b).** How would you program such a server using asynchronous channels? *(3p)*

**(Part c).** Now suppose no process may hold more than one integer. Then a buffer holding several messages has to be a chain of processes. Implement an infinite buffer that stretches as needed, but never shrinks. So once it holds 5 integers, all future produced elements will pass through at least 5 processes before reaching the consumer, even if the consumer catches up with the producer. This strange never shrinking buffer is easier to program than one that shrinks.

*Hint:* A timeout can tell whether the buffer needs to expand. Use a local channel declared in the buffer process to create a new link, between the new cell and the continuing buffer. *(8p)*

**Question 6.** Here you will use Linda to sort a list of words such as `[dog, bat, cat]` to yield `[bat, cat, dog]`. The input list will be represented by triples of the form `(in, ``dog'', 1)`, `(in,``bat'', 2)` and `(in,``cat'', 3)`, and you should produce the triples `(out, ``dog'', 3)`, `(out,``bat'', 1)` and `(out,``cat'', 2)`. Here the "in" and "out" merely say whether the triple is an input or an output triple, and the integer in the triple gives the position of the word in the list being represented this way.

You are given the length $n$ of the input list separately, so you don't have to compute it, but your program should work for arbitrary $n \geq 0$.

You should assume that processes can only hold a small, fixed, number of triples. So no one process can hold the entire input or output list.

*Hints: Representing lists without indices.* `[bat, cat, dog]` can be represented by a set of pairs `(aaaa, bat)`, `(bat, cat)`, `(cat, dog)` and `(dog, zzzz)`, where we have assumed that `aaaa` comes before any word in the list, and `zzzz` after, so that they can be used as sentinel elements to mark the beginning and the end of the list.

We produce the output by chasing links: A pair is "activated" when its first element has been output. Its second element is then output, and in turn activates another pair. We commence by outputting the partner of the start sentinel and stop when we have output the partner of the end sentinel. For this to work, we assume that the words in the list are all distinct.

Inserting `bill` into the sorted list `[bat,cat,dog]` is easy in this representation: the set of pairs `(bat, cat)` and `(cat, dog)` becomes the set `(bat, bill)`, `(bill, cat)` and `(cat, dog)`.

**(Part a).** Write a Linda program that starts from a tuple space with the input triples and the length $n$, and produces the output triples representing the sorted input list. Embed the Linda calls (post, read and remove) in any programming language or pseudo-code, but all inter-process communication must be via the Linda calls. Maximise parallelization.

*Hint:* Remember that a process changes state after each action; in particular, it can become a parallel composition of two or more processes. *(8p)*

**(Part b).** How would you program this if the input were just `(``bat'')` , `(``cat'')`, and so on, instead of the triples `(in, ``bat'', 5)`, `(in, ``cat'', 3)`, and so on? *(3p).*

**(Part c).** In a simple but acceptable solution to (Part a) above, the input `cat` splits `(``bat'', ``dog'')`, but is also checked by every other pair in the chain. In a massively parallel system, this wouldn't matter, because the other pairs are being examined at the same time, and there isn't anything better to do anyway. But in a single CPU system, we would like to move on once `cat` has split `(``bat'', ``dog'')`, without then needlessly checking any other pairs. How can this waste be avoided? *Hint:* You might want to use a timeout. *(5p)*

———-THE END———-
(Page 6 is blank)