Chalmers | Göteborgs Universitet

Principles of Concurrent Programming TDA384/DIT391

Tuesday, 19 December 2017

Teacher/examiner: K. V. S. Prasad (prasad@chalmers.se, 0736 30 28 22)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

Points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

Points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will not receive any points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Write the solutions to different exercises on different pages.
- Try to be precise. For the syntax of the programming notation you use in your answers, try to use the same syntax and style used in the questions. But you can also use pseudo-code, *as long as* the intended meaning is precise and clear. If need be, explain your notation.

Question 1. Consider the following program.

boolean flag := true; integer n := 0	
p	q
p1: while flag	q1: while flag
p2: n := n+1	q2: n := n-1
p3: flag := true	q3: if n < 0
	q4: flag := false

(Part a). Construct a scenario where the program terminates and $n >= 0$. *(2p)*

(Part b). Construct a fair scenario where the program does not terminate. *(3p)*

(Part c). Construct a scenario where the program does not terminate and q4 is executed infinitely often. *(3p)*

Question 2. A small building firm can only build one house at a time, and cannot start on a new one till the present one is completed. The firm has N specialist workers such as a mason, a carpenter, an electrician, a plumber, etc. They are told to start on a house by the team manager, who then waits till each worker reports that they are done on this house, before starting the team on the next house. The firm never stops building houses.

On the next page is a code skeleton of a program modelling the behaviour of this small firm.

```
class BuildingFirm {

    final int NumSpecialists = 2;

    // Semaphore definitions to be defined...

    class TeamManager extends Thread {
        public void run() {
            // To be defined...
        }
    }

    class Worker extends Thread {
        public void run() {
            // To be defined...
        }
    }

    // Starting the workers and team manager
    public static void main(String[] args) {
        for (int i = 0; i<NumSpecialists; i++) {
            new Worker().start();
        }
        new TeamManager().start();
    }
}
```

Your task is to replace the comments `// To be defined...` as follows:

(Part a). Write the definition of the semaphores you will use in your solution. For each semaphore, indicate its name and the number of permits with which it is initialised. These semaphores will be global variables that can be accessed by any thread. *(2p)*

(Part b). Write the implementation of the method `run()` of the class `Worker` according to the description above. Remember that the only shared variables among threads are `NumSpecialists` and the semaphores you defined in Part a. *(5p)*

(Part c). Write the implementation of the method `run()` of the class `TeamManager` according to the description above. Remember that the only shared variables among threads are `NumSpecialists` and the semaphores you defined in Part a. *(5p)*

(Part d). How would your solution change if you only use binary semaphores? *(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. The / operator is integer division, so $2/2 = 3/2 = 1$.

integer S := 0	
p	q
loop forever	loop forever
p1: // non-critical section	q1: // non-critical section
p2: if even(S) then S:=2 else S:=3	q2: if even(S/2) then S:=5 else S:=7
p3: await (S≠ 1 ∧ S≠3)	q3: await (S≠ 6 ∧ S≠7)
p4: // critical section	q4: // critical section
p5: if odd(S/2) then S:=S-2 else skip	q5: if odd(S) then S:=S-1 else skip

Commands p1, p4, q1 and q4 (the critical and non-critical sections) do not access the variable S, and are therefore omitted in the abbreviated state transition table (a tabular version of a state diagram) below for the program. Many entries in the table are left blank (—).

Each state is represented by a triple (pk, ql, Sn), where pk and ql say respectively what p and q will next execute, and Sn is the value of S. The states are listed in the order in which they appear as the table is built up starting from (p2, q2, 0), and are named s1 through s10. (There are 10 states in all). The left hand column lists the states. The next state if p (respectively q) next executes a step is given in the middle (respectively last) column. 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 s5 below, one or other of the processes may be blocked. The middle and last columns show the next state and give its name for easy reference.

	State = (pi, qi, Svalue)	next state if p moves	next state if q moves
s1	(p2, q2, 0)	(p3, q2, 2)=s2	(p2, q3, 5)=s3
s2	(p3, q2, 2)	—	—
s3	(p2, q3, 5)	—	—
s4	—	—	—
s5	(p3, q3, 7)	(p5, q3, 7)=s8	no move
s6	—	—	—
s7	—	—	—
s8	—	—	—
s9	—	—	—
s10	(p2, q2, 4)	(p3, q2, 2)=s2	(p2, q3, 5)=s3

(Part a) Fill in the dashes to complete the state transition table. Each entry should show a state, and in the middle and last columns, also give its name. *(5p)*

(Part b) Prove from your state transition table that the program ensures mutual exclusion. *(3p)*
(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, reproduced below for convenience.

integer S := 0	
p	q
loop forever	loop forever
p1: // non-critical section	q1: // non-critical section
p2: if even(S) then S:=2 else S:=3	q2: if even(S/2) then S:=5 else S:=7
p3: await (S≠ 1 ∧ S≠3)	q3: await (S≠ 6 ∧ S≠7)
p4: // critical section	q4: // critical section
p5: if odd(S/2) then S:=S-2 else skip	q5: if odd(S) then S:=S-1 else skip

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.

Ben-Ari's textbook reviews briefly the notation of propositional logic and explains linear temporal logic. Below, we write pi as a logical proposition to mean "process p is at pi".

(Part a). Show that $(p3 \wedge q3) \rightarrow (S = 3 \vee S = 7)$ is invariant. *Hint:* Reason about what must have happened for the program to get to $(p3 \wedge q3)$. *(4p)*

(Part b) Assume that $p3 \wedge q1 \rightarrow (S = 2)$ is invariant, and that q is stuck in a loop in q1. (Remember that while p4 and q4 are assumed to terminate, p1 and q1 may loop). Assuming fairness, prove that $p3 \wedge q1 \rightarrow \Box\Diamond p5$. *(4p)*

Question 5. In this question we model the exam grading process using Erlang. There is an examiner process who keeps track of which exams have been graded, and N grader processes who do the work of grading. Each exam is graded by one grader, and one grader can grade multiple exams. Grading an exam takes a non-trivial, indeterminate amount of time. Every grader asks the examiner for an ungraded exam, grades it, and then gives it back. This is repeated until all exams are graded. The grader processes terminate when there is no work left to be done (but the examiner process never terminates).

You can assume the following functions. You do not need to concern yourself with the internal structure of the exam data types.

- `grade(Exam)`: Grade the given exam (the work done by the graders). Blocks while the exam is being graded. Returns a graded version of `Exam`.
- `get_ungraded_exam(Exams)`: Find and return an exam in the list `Exams` which has not yet been graded. Returns `false` if all exams have been graded.
- `set_graded_exam(Exams, Exam)`: Mark that `Exam` in the list `Exams` has been graded. Returns a updated version of `Exams`.

(Part a). Implement the `init_graders` function, which spawns N grader processes (each running the `grader` function, which you will implement in the next question). Use the following signature:
`init_graders(N) -> ...` *(2p)*

(Part b).The examiner process runs the following function:

```
examiner1(Exams) ->
  receive
    {idle, Pid} ->
      case get_ungraded_exam(Exams) of
        false -> Pid ! finished ;
        Exam  ->
          Pid ! {grade, Exam},
          receive
            {ready, ExamGraded} ->
              examiner1(set_graded_exam(Exams, ExamGraded))
          end
      end
  end.
```

Implement the `grader` function, which communicates with this examiner process and behaves as described above. It is up to you to decide

the signature of this function, but it should match your implementation of `init_graders` above. You can assume that the examiner process is running and registered to the atom `examiner`. *(8p)*

(Part c).The `examiner1` function defined above is not efficient. Explain why. *(2p)*

(Part d). Here's an attempt at improving the examiner function:

```
examiner2(Exams) ->
  receive
    {idle, Pid} ->
      case get_ungraded_exam(Exams) of
        false -> Pid ! finished ;
        Exam ->  Pid ! {grade, Exam}
      end,
      examiner2(Exams) ;
    {ready, ExamGraded} ->
      examiner2(set_graded_exam(Exams, ExamGraded))
  end.
```

Explain why it is an improvement over the previous version. Are there any potential problems with this implementation? *(3p)*

Question 6. In this exercise, you will evaluate different implementations of a counting operation on a list data structure in Java, analyzing whether they are *thread safe*, that is executable by multiple concurrent threads without running into race conditions.

Recall the various implementations of linked sets presented during the course. They are all variants implementing the same *interface*, consisting of operations to remove elements, add elements, and test whether an element is in the set. In this exercise, we extend the interface with a new operation `int size()`, which simply returns the number of nodes stored in the set.

A simple implementation of method `size()` that works in a sequential setting is:

```
1  public int size() {
2    int size = -1;
3    Node<T> curr;
4    curr = head;                // set curr to the head node
5    do {
6      curr = curr.next();       // move curr to next node in chain
7      size += 1;                // increment size by 1
8    } while (curr != tail);     // until curr reaches the tail node
9    return size;
10 }
```

(Part a). Why is the local variable `size` initialized to -1? Why not to 0? *(2p)*

(Part b). Explain why the above implementation of `size()` is not thread safe. To this end, describe a concrete scenario where race conditions may occur. *(2p)*

(Part c). `CoarseSet` uses a variable `lock` of type `Lock` to guard access to the whole data structure. Modify the implementation of `size()` shown above so that it uses `lock` to avoid race conditions. (Recall the implementation `CoarseSet` of the thread-safe set data structures seen during the course and also described in Chapter 9 of Herlihy & Shavit). *(2p)*

(Part d).

In order to make `size()` run in constant time, we now consider a more efficient implementation that adds an attribute `size` of type `int` to the set, which keeps track of the current number of elements in the list. Thus, method `size()` simply returns the value of attribute `size` when it is called.

1 Write an implementation of `size()` in `CoarseSet` that is thread safe using locks. *(1p)*

2 Which operations of `CoarseSet` must update the value of attribute `size`? *(1p)*

3 Consider the implementation of method `remove` in `CoarseSet`. Illustrate what race conditions may occur if `remove` updates the value of attribute `size` *after* releasing `lock`. *(2p)*

(Part e).

Consider yet another variant of set implementation where we want to update attribute `size` without using any locks.

1 Choose a suitable *type* for attribute `size` so that it can be updated thread-safely without using locks. *(2p)*

2 Based on your choice of type, write a piece of code that increments `size` by one in a thread-safe manner without locking. *(2p)*