Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Thursday, 9 January 2020

Exam supervisor: Nir Pieterman (piterman@chalmers.se, 073 856 4910)

(Exam set by K. V. S. Prasad, based on the course given Sep-Oct 2019)

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.
- Answer each question on a new page. Glance through the whole paper first; six questions, numbered Q1 through Q6. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1 (10p). Below is the pseudo-code of a program with two threads, `p` and `q`. The variables `n` and `flag` are shared between `p` and `q`.

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 $n < 0$. *(1p)*

(Part b). Construct a scenario where the program terminates and $n < 0$, and p2 executes at least once. *(2p)*

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

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

(Part e). Construct a scenario where the program does not terminate and q4 is executed infinitely often. *(2p)*

Q2 (15p). 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 declarations 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 declarations 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. What should the scope of these semaphores be? *(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 perhaps 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 perhaps the semaphores you defined in Part a. *(5p)*

(Part d). How would your solution change if you only use binary semaphores? *(3p)*

Q3 (12p). The program below tries to solve the critical section problem. The global variable S can take values Z, P, Q, PQ, or QP. Commands (p3, q3) await either of two conditions. Commands p2, q2, p5 and q5 are atomic: testing S, and then assigning to it, run without interruption.

type switch = {Z, P, Q, PQ, QP} switch S := Z	
p	q
loop forever	loop forever
p1: //NCS (non-critical section)	q1: //NCS (non-critical section)
p2: case S of	q2: case S of
Z → S:=P;	Z → S:=Q;
Q → S:=QP;	P → S:=PQ;
else → skip	else → skip
p3: await (S=P or S=PQ)	q3: await (S=Q or S=QP)
p4: //CS (critical section)	q4: //CS (critical section)
p5: case S of	q5: case S of
P → S:=Z;	Q → S:=Z;
PQ → S:=Q;	QP → S:=P;
else → skip	else → skip

Below is part of the state transition table for an abbreviated version of this program, skipping p1, p4, q1 and q4 (the CS and NCS parts).

The left column shows the state (where is p, where is q, what is S). The middle column gives the next state if p now runs a step, and the last column gives the next state if q now runs a step. In some states both p or q are free to run a step. But in some states such as 5 below, one or both processes may be blocked. There are 9 states in all.

	State = (pi, qi, S)	next state if p moves	next state if q moves
1.	(p2, q2, Z)	(p3, q2, P)	(p2, q3, Q)
2.	(p2, q3, Q)		
3.			
4.	(p3, q2, P)		
5.	(p3, q3, PQ)	(p5, q3, PQ)	no move
6.			
7.			
8.	(p5, q2, P)	(p2, q2, Z)	(p5, q3, PQ)
9.			

(Part a) Fill in the blanks in the state transition table. *(4p)*
Now use your table in **Parts b, c and d**, to:
(Part b) show whether the program ensures mutual exclusion. *(2p)*
(Part c) show the presence or absence of deadlock. *(2p)*
(Part d) show that even if q dies in q1, the program ensures the liveness of p (i.e., it will progress). *(4p)*

Q4 (10p). The program in Q3 is reproduced below. Commands (p3, q3) await either of two conditions; commands p2, q2, p5 and q5 are atomic.

type switch = {Z, P, Q, PQ, QP} switch S := Z	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: case S of	q2: case S of
Z → S:=P;	Z → S:=Q;
Q → S:=QP;	P → S:=PQ;
else → skip	else → skip
p3: await (S=P or S=PQ)	q3: await (S=Q or S=QP)
p4: critical section	q4: critical section
p5: case S of	q5: case S of
P → S:=Z;	Q → S:=Z;
PQ → S:=Q;	QP → S:=P;
else → skip	else → skip

Below, you must argue from the program, not from the state transition table. You get credit for correct reasoning. You may use a mixture of formal logic and everyday language. Formulas and logical laws make your argument concise and precise. With everyday language, be careful to not be fuzzy, or to substitute wishful thinking for proof.

Below, pi is a logical proposition that means "process p is at pi". Also, for "S=X", we write just "X", as Z, P, Q, PQ and QP are unambiguous.

Note that $p \vee q$ ("p or q") is false iff (if and only if) both p and q are false, and that $p \rightarrow q$ ("p implies q") is false iff p is true and q is false.

Let $M_p \equiv p4 \rightarrow (P \vee PQ)$, i.e., if p is at $p4$, then S will be P or PQ.

(Part a). Show that M_p is invariant (always holds).
Hint: How could M_p be false? Either because p arrives at $p4$ when S is neither P nor PQ, or by both P and PQ becoming false while p is at $p4$. Show both impossible. *(4p)*

(Part b). Assume M_p and its symmetric counterpart M_q for q are both always true. Prove that the program ensures mutual exclusion, i.e., we never have $(p4 \wedge q4)$. *(3p)*

(Part c). Show that the program cannot reach a deadlocked state, i.e., one where we are stuck with $(p3 \wedge q3)$.
Hint: Suppose we are stuck with $(p3 \wedge q3)$. Then what must S be? Can it hold this value after $p2$ or $q2$ (one of which must be the last command before $p3 \wedge q3$)? *(3p)*

Q5.(13p). 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.

(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`. *(6p)*

(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)*

Q6.(10p). Recall that a data structure implementation is *thread safe* if its operations can be executed by multiple concurrent threads without running into race conditions. In this exercise, you will evaluate different implementations of an operation on a simple data structure in Java, analyzing whether they are thread safe.

The data structure simply stores two integers X and Y in a way that it is possible to increment both variables at once. Class Pair is a *sequential implementation* of the data structure:

```
class Pair {
    private int X;
    private int Y;
    public int getX() { return X; } // current value of X
    public int getY() { return Y; } // current value of Y
    public void incXY() { // increment X and Y
        X = getX() + 1;
        Y = getY() + 1;
    }
}
```

(Part a). Why is the above implementation of Pair not thread safe?

- Describe a concrete scenario where race conditions may occur.
- List all operations (that is, methods) that are not thread safe.

(5p)

(Part b). Write the implementation of a class `LockedPair`, which provides the same operations as `Pair` but is *thread-safe*. To this end `LockedPair` introduces a single variable `lock` to guard access to the data structure. (Your implementation of `LockedPair` may inherit from `Pair` or directly modify its implementation.) *(5p)*

Appendix

A Linear Temporal Logic (LTL) notation

1. An atomic proposition such as q_2 (process q is at label q_2) *holds for* a state s if and only if process q is at q_2 in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", $\rightarrow$ for "implies", $\Box$ for "always", and $\Diamond$ for "eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ ("ϕ or ψ") is false iff both ϕ and ψ are false. This is an "inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ ("ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\Box$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\Box\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$