Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Wednesday, 9 June 2021

Exam supervisor: N. Piterman (piterman@chalmers.se, 073 856 49 10)

(Exam set by N. Piterman and G. Schneider, based on the courses given in September-October 2020 and January-March 2021)

Material permitted during the exam (hjälpmedel):
As the exam is run remotely we cannot realy restrict your usage of material.

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:

- You should be monitored on the dedicated zoom channel while taking the exam!
- Submit the exam solution as a **PDF** file on Canvas. The solution should be typeset using your favourite software. **No** scanned handwritten notes or diagrams are allowed.
- 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; five questions, numbered Q1 through Q5. 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.
- A Word template and a Latex template are available on Canvas so you can use them to deliver your answer.

Q1.(14p). This question is concerned with the model of cake baking production using Erlang.

The production system consists of two types of processes. There is a baking process that keeps track of which cakes have been baked, and M baker processes that do the work of baking. Each cake is baked by one baker, and one baker can bake multiple cakes sequentially. Baking a cake takes a non-trivial, indeterminate amount of time. Every baker asks the baking process for an unbaked cake, bakes it, and then gives it back. This is repeated until all cakes are baked. The baker processes terminate when there is no work left to be done (note though that the baking process never terminates).

You can assume the following functions (you do not need to concern yourself with the internal structure of the cake data types.)

- `bake(Cake)`: Bake the given cake (the work done by the bakers). Blocks while the cake is being baked. Returns a baked version of `Cake`.
- `get_unbaked_cake(Cakes)`: Find and return a cake in the list `Cakes` which has not yet been baked. Returns `false` if all cakes have been baked.
- `set_baked_cake(Cakes, Cake)`: Mark that `Cake` in the list `Cakes` has been baked.

(Part a). Implement the `init_bakers` function, which spawns M baker processes (each running the `baker` function, which you will implement in the next question). Use the following signature:

```
init_bakers(M) -> ...
```

(2p)

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

```
baking1(Cakes) ->
  receive
    {idle, Pid} ->
      case get_unbaked_cake(Cakes) of
        false -> Pid ! finished ;
        Cake  ->
          Pid ! {bake, Cake},
          receive
            {ready, CakeBaked} ->
               baking1(set_baked_cake(Cakes, CakeBaked))
          end
      end
  end.
```

Implement the `baker` function, which communicates with this baking process and behaves as described above. You can decide the signature of this function, but it should match your implementation of `init_bakers` above. You can assume that the baking process is running and registered to the atom `baking`. *(6p)*

(Part c).The 'baking1' function defined above is not efficient. Explain why. *(2p)*

(Part d). The following is an attempt at improving the baking function:

```
baking2(Cakes) ->
  receive
    {idle, Pid} ->
      case get_unbaked_cake(Cakes) of
        false -> Pid ! finished ;
        Cake ->  Pid ! {bake, Cake}
      end,
      baking2(Cakes) ;
    {ready, CakeBaked} ->
      baking2(set_baked_cake(Cakes, CakeBaked))
  end.
```

Explain why the above solution is an improvement over the previous version. Are there any potential problems with this implementation? Explain. *(4p)*

Q2 (19p). We have seen the following parallel implementation of merge sort (lecture 09, combination of slides 21, 22, and 25):

```
1 public class PMergeSort extends RecursiveAction {
2   private Integer[] data;
3   private int low, high;
4
5   @override
6   protected void compute() {
7     if (high - low <= 1) {
8       sort(data,low,high); // sort sequentially small chunks of 1024
9       return;              // or less
10    }
11    int mid = low + (high - low)/2; // mid point
12    // left and right halves
13    PMergeSort left = new PMergeSort(data,low,mid);
14    PMergeSort right = new PMergeSort(data,mid,high);
15    left.fork();  // fork thread working on left
16    right.fork(); // fork thread working on the right
17    left.join();  // wait for sorted left half
18    right.join(); // wait for sorted right half
19    merge(mid);   // merge halves
20  }
21 }
```

The following appears somewhere in the `main`:

```
1 RecursiveAction sorter = new PMergeSort(numbers,0,numbers.length);
2 ForkJoinPool.commonPool().invoke(sorter);
```

Based on the dependency graph (or otherwise) for a run of `invoke(sorter)` when the array numbers has 8 elements, answer the following.

(Part a). How many threads participate in the computation? *(4p)*

(Part b). What is the maximum number of tasks that can be executed in parallel in this implementation on the same data (excluding parent tasks waiting for a child task to finish)? *(4p)*

You apply the second optimization in slide 25. That is, you change line 16 to `right.compute();` and comment out line 18.

(Part c). How many threads participate now in the computation? *(4p)*

(Part d). What is the maximum number of tasks that can be executed in parallel? *(3p)*

(Part e).

You now get an array with 9000 elements. Change the program according to the first advice in slide 25 so that the number of threads that participate in the computation does not change to all the previous answers. *(4p)*

Q3 (11p). This program solves solves the critical section problem for two-threads. Remember that we assume that a thread leaves the critical section after a finite time but may stay forever in the non-critical section.

The label p_i can mean the command that follows p_i, or the proposition that thread `p` is at p_i, and *the next command* `p` will execute is p_i.

int `turn`= 1; **int** `flaga`= 1; **int** `flagb`= 1;			
	`p`		`q`
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
p_2:	`flaga`= 0;	q_2:	`flagb`= 0;
p_3:	`turn`= 1;	q_3:	`turn`= 0;
p_4:	**while**(`flagb`!= `turn`) { };	q_4:	**while**(`flaga`== `turn`) { };
p_5	*//CS (critical section)*	q_5	*//CS (critical section)*
p_6:	`flaga`= 1;	q_6:	`flagb`= 1;
	}		}

For simplicity, we ignore the locations p_1 and p_4 and similarly q_1 and q_4. Process p moves directly from p_4 to p_6 and from p_6 to p_2 and similarly for q. We treat p_6 and q_6 as the critical section.

(Part a) Show that $(p_2 \iff (\texttt{flaga} == 1))$ is an invariant of the program. That is, it always holds. Show that it holds initially and that it is preserved under every transition of process p. *(2p)*

Use the invariant $(q_2 \iff (\texttt{flagb} == 1))$ without proof. Notice that these are equivalent to $((p_3 \vee p_4 \vee p_6) \iff (\texttt{flaga} == 0))$ and $((q_3 \vee q_4 \vee q_6) \iff (\texttt{flagb} == 0))$.

(Part b) Show that $(p_4 \implies ((\texttt{turn} == 1) \vee q_4))$ is an invariant of the program. Show that it holds initially and that it is preserved under every transition of **every process**. *(3p)*

The invariant $(q_4 \implies ((\texttt{turn} == 0) \vee p_4))$ holds as well.

(Part c) Show that $(p_6 \implies ((\texttt{turn} == 1) \vee q_4))$ is an invariant of the program. *(3p)*

Use the invariant $(q_6 \implies ((\texttt{turn} == 0) \vee p_4))$ without proof.

(Part d) Show that the program maintains mutual exclusion. *(3p)*.

Q4 (17p). The program from Q3 is repeated below for convenience.

int turn= 1; **int** flaga= 1; **int** flagb= 1;			
	p		q
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
p_2:	flaga= 0;	q_2:	flagb= 0;
p_3:	turn= 1;	q_3:	turn= 0;
p_4:	**while**(flagb!= turn) { };	q_4:	**while**(flaga== turn) { };
p_5	*//CS (critical section)*	q_5	*//CS (critical section)*
p_6:	flaga= 1;	q_6:	flagb= 1;
	}		}

You are going to construct the transition table of this program. A full state is of the form $(p_i, q_j, \texttt{flaga}, \texttt{flagb}, \texttt{turn})$, where i and j range over $\{2, 3, 4, 6\}$, and flaga, flagb, and turn range over 0 and 1. From **Q2** we know that flaga and flagb can be deduced from p_i and q_j. So a reduced state is of the form $(p_i, q_j, \texttt{turn})$. As transitions into p_4 and q_4 set turn, we can ignore the value of turn when both p and q are in locations 2 or 3. Only 16 states are reachable.

Notation: We denote the value of turn by x when we do not care about it. For example, (p_2, q_2, x) correponds to either $(p_2, q_2, 0)$ or $(p_2, q_2, 1)$.

Here is a partial state transition table for the program above. As mentioned, only 16 states are reachable from the initial state $(p_2, q_2, 1)$.

state		new state if p moves	new state if q moves
s1	$(2, 2, x)$	$(3, 2, x)$ = s3	
s2	$(2, 3, x)$		$(2, 4, 0)$ = s5
s3	$(3, 2, x)$	$(4, 2, 1)$ = s7	$(3, 3, x)$ = s4
s4	$(3, 3, x)$		
s5	$(2, 4, 0)$		$(2, 6, 0)$ = s6
s6	$(2, 6, 0)$		$(2, 2, x)$ = s1
s7	$(4, 2, 1)$	$(6, 2, 1)$ = s8	
s8	$(6, 2, 1)$	$(2, 2, x)$ = s1	
s9	$(4, 3, 1)$		
s10	$(4, 4, 1)$		
s11	$(4, 4, 0)$		
s12	$(4, 6, 1)$		$(4, 2, 1)$ = s7
s13	$(6, 3, 1)$	$(2, 3, x)$ = s2	
s14	$(6, 4, 0)$	$(2, 4, 0)$ = s5	
s15	$(3, 4, 0)$		
s16	$(3, 6, 0)$		$(3, 2, x)$ = s3

(Part a) Fill in the blank entries in the table. *(8p)*

(Part b) Explain why the protocol maintains mutual exclusion. *(2p)*

(Part c) Explain why under fair scheduling the protocol avoids starvation. *(7p)*

Q5 (9p). Consider this two-threaded program with threads `s` and `t`. The two threads share the variables `n1` and `n2`. The function $unknown(\cdot,\cdot)$ is an unknown function that gets two integer parameters and returns one integer. The function *unknown* is meant to remain completely unknown.

```
                    int n1 = 0;
                    int n2 = 0;
```

	s		t
s_1:	**while**(n1 <= 500) {	t_1:	**while**(n2<=500) {
s_2:	n1 = n1 + 1;	t_2:	n2 = n2 + 1;
s_3:	n2 = unknown(n1,n2)		}
	}		

The labels s_1, s_2, s_3, t_1 and t_2 are given only for ease of reference.

(Part a) Show an execution in which the program terminates. *(4p)*

(Part b) Does the program terminate in all fair executions? *(5p)*