**Chalmers** | Göteborgs Universitet

# Concurrent Programming TDA384/DIT391

Wednesday, 19 August 2020

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

(Exam set by N. Piterman, based on the course given January-March 2020)

**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 hand-written 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.

**Q1** (9p). Below is the pseudo-code of a program with two threads, `p` and `q`. The variables `x` and `y` are shared between `p` and `q`. The function $f(\cdot,\cdot)$ accepts two integers as parameters and returns an integer. You do not need to know anything else about $f$.

| | **int** x = 1000; **int** y = 1000; | | |
|---|---|---|---|
| | p | | q |
| $p_1$: | **while**(x>0) { | $q_1$: | **while**(y>0) { |
| $p_2$: | x = x - 1; | $q_2$: | y = y - 1; |
| | } | $q_3$: | x = f(x,y) |
| | | | } |

The labels $p_1, p_2, q_1, q_2$ and $q_3$ are given only for ease of reference.

**(Part a)** Construct a scenario for which the program terminates. *(4p)*

**(Part b)** Does the program terminate for all scenarios? *(5p)*

*Answers:*

**(Part a)** *Process q runs until y is 0 and terminates. Then if $x \leq 0$ process p terminates as well. Otherwise, the only possible change to x is p reducing it. Hence, x will reduce below* 1.

**(Part b)** *Yes. The program terminates for all scenarios. The only change from the reasoning above is that if p is allowed to run long enough without q interfering then p will terminate first.*

**Q2** (11p). The pseudo-code below tries to solve the critical section (CS) problem with two threads, `p` and `q`. Remember that CS must exit after a finite time, but NCS may loop.

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 $p_i$.

| **boolean** `t`= false; **boolean** `tp`= false; **boolean** `tq`= false; | | | |
|---|---|---|---|
| | `p` | | `q` |
| | **while**(true) { | | **while**(true) { |
| $p_1$ | *//NCS (non-critical section)* | $q_1$ | *//NCS (non-critical section)* |
| $p_2$: | `tp`= true; | $q_2$: | `tq`= true; |
| $p_3$: | `t`= false; | $q_3$: | `t`= true; |
| $p_4$: | **while**(`tq`!= `t`) { }; | $q_4$: | **while**(`tp`== `t`) { }; |
| $p_5$ | *//CS (critical section)* | $q_5$ | *//CS (critical section)* |
| $p_6$: | `tp`= false; | $q_5$: | `tq`= false; |
| | } | | } |

**(Part a)** Show that ($p_2 \iff \neg$`tp`) 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)*

*Answer: It holds initially. The transition from $p_2$ to $p_3$ sets `tp` to true, keeping the invariant true. The transition from $p_6$ to $p_2$ sets `tp` to $false$, keeping the invariant true. Other transitions do not change `tp` and the truth value of $p_2$.*

Use the invariant ($q_2 \iff \neg$`tq`) without proof. Notice that these are equivalent to (($p_3 \vee p_4 \vee p_6$) $\iff$ `tp`) and (($q_3 \vee q_4 \vee q_6$) $\iff$ `tq`).

**(Part b)** Show that ($p_4 \implies \neg t \vee q_4$) is an invariant of the program. Show that it holds initially and that it is preserved under every transition of **every process**. *(3p)*

*Answer: The invariant holds if $p$ is not in location 4.*

*When $p$ moves into location 4 (i.e., $p_4$ becomes true) it does so by setting $t$ to false. Thus establishing the invariant.*

*We care only about transitions of $q$ if they change $q_4$ or change $t$. Thus, move from 3 to 4 and move from 4 to 6.*

*The move of $q$ from 3 to 4 sets $q_4$ to true. Thus if $q$ moves into $q_4$ the invariant holds.*

*Consider the transition of $q$ from 4 to 6 and the case that $p$ is in location 4 (i.e., from (a) `tp` is true). When `tp` is true, process $q$ can move to location 6 only if $t$ is false. Thus, after this transition the invariant is established.*

The invariant ($q_4 \implies t \vee p_4$) holds as well.

**(Part c)** Show that $(p_6 \implies \neg t \vee q_4)$ is an invariant of the program. *(3p)*

*Answer: We know that $p_4 \implies \neg t \vee q_4$ is an invariant. When $p$ moves from location 4 to location 6 both $t$ and $q$'s location do not change. Thus, the invariant is established.*

*We care only about transitions of $q$ if they change $q_4$ or change $t$. Thus, move from 3 to 4 and move from 4 to 6.*

*The move of $q$ from 3 to 4 sets $q_4$ to true. Thus if $q$ moves into $q_4$ the invariant holds.*

*Consider the transition of $q$ from 4 to 6 and the case that $p$ is in location 6 (i.e., from (a)* `tp` *is true). When* `tp` *is true, process $q$ can move to location 6 only if $t$ is false. Thus, after this transition the invariant is established.*

Use the invariant $(q_6 \implies t \vee p_4)$ without proof.

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

*Answer: Suppose that both processes are in locations 6 at the same time. It follows from (c) that both $\neg t \vee q_4$ and $t \vee p_4$ hold. However, by assumption $p_6$ and $q_6$ hold. Thus, $t$ must be true and false at the same time. This is impossible.*

**Q3** (17p). The program from Q2 is repeated below for convenience.

| **boolean** t= false; **boolean** tp= false; **boolean** tq= false; | | | |
|---|---|---|---|
| | p | | q |
| | **while**(true) { | | **while**(true) { |
| $p_1$ | *//NCS (non-critical section)* | $q_1$ | *//NCS (non-critical section)* |
| $p_2$: | tp= true; | $q_2$: | tq= true; |
| $p_3$: | t= false; | $q_3$: | t= true; |
| $p_4$: | **while**(tq!= t) { }; | $q_4$: | **while**(tp== t) { }; |
| $p_5$ | *//CS (critical section)* | $q_5$ | *//CS (critical section)* |
| $p_6$: | tp= false; | $q_5$: | tq= false; |
| | } | | } |

From the onset, we expect each state to be a quintuple, $(p_i, q_j,$ `tp, tq, t`$)$, where $i$ and $j$ range over $\{2, 3, 4, 6\}$, and `tp, tq, t` are Boolean. From **Q2** we know that `tp` and `tq` can be deduced from $p_i$ and $q_j$. Hence, we use states of the form $(p_i, q_j, \texttt{t})$. As transitions into $p_4$ and $q_4$ set `t`, we can ignore the value of `t` when both $p$ and $q$ are in locations 2 or 3. Only 16 states are reachable.

**Notation:** We denote the value of `t` by $x$ when we do not care about it. For example, $(p_2, q_2, x)$ correponds to either $(p_2, q_2, false)$ or $(p_2, q_2, true)$.

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

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

*Answer:*

| *state* | | *new state if p moves* | *new state if q moves* |
|---|---|---|---|
| *s1* | $(2,2,x)$ | $(3,2,x) = s3$ | $(2,3,x) = s2$ |
| *s2* | $(2,3,x)$ | $(3,3,x) = s4$ | $(2,4,true) = s5$ |
| *s3* | $(3,2,x)$ | $(4,2,false) = s7$ | $(3,3,x) = s4$ |
| *s4* | $(3,3,x)$ | $(4,3,false) = s9$ | $(3,4,true) = s15$ |
| *s5* | $(2,4,true)$ | $(3,4,true) = s15$ | $(2,6,true) = s6$ |
| *s6* | $(2,6,true)$ | $(3,6,true) = s6$ | $(2,2,x) = s1$ |
| *s7* | $(4,2,false)$ | $(6,2,false) = s8$ | $(4,3,false) = s9$ |
| *s8* | $(6,2,false)$ | $(2,2,x) = s1$ | $(6,3,false) = s13$ |
| *s9* | $(4,3,false)$ | *no move (s9)* | $(4,4,true) = s11$ |
| *s10* | $(4,4,false)$ | *no move (s10)* | $(4,6,false) = s12$ |
| *s11* | $(4,4,true)$ | $(6,4,true) = s14$ | *no move (s11)* |
| *s12* | $(4,6,false)$ | *no move (s12)* | $(4,2,false) = s7$ |
| *s13* | $(6,3,false)$ | $(2,3,x) = s2$ | $(6,4,true) = s14$ |
| *s14* | $(6,4,true)$ | $(2,4,true) = s5$ | *no move (s14)* |
| *s15* | $(3,4,true)$ | $(4,4,false) = s10$ | *no move (s15)* |
| *s16* | $(3,6,true)$ | $(4,6,false) = s12$ | $(3,2,x) = s3$ |

*For (b) states* $(6,6,false)$ *and* $(6,6,true)$ *are not reachable.*

*For (c) from* $s4$ *there is a choice whether to go to* $s9$ *or* $s15$*. If going for* $s9$*, then there is no choice but to continue to* $s11, s14, s5$ *now we have to show that q will enter the critical section. This is if either we go directly to* $s6$ *or* $s15, s10, s12$*. The case of going from* $s4$ *to* $s15$ *is dual.*

**Q4** (14p). In this question we create a barrier in erlang. The setup should include a number (10) of clients each doing some work in rounds. They should all be doing the work on round $i$ at the same time and none of them can proceed to round $i + 1$ before all are done with round $i$.

The role of the barrier is taken by a server that interacts with the clients. The server sends a message to the clients telling them to start the $i$-th round and collects messages that notify it of the end of the work of each client on round $i$. After all clients have completed round $i$, the server initiates round $i + 1$.

Once all clients have finished 100 rounds, the server should ensure that all clients have terminated before terminating itself.

Your task is to implement the server and the client.

**(Part a).** Implement the server. Explain the role of the elements of the server's state. *(6p)*

**(Part b).** Implement the client. Explain the role of the elements of the client's state (if exists). *(4p)*

**(Part c).** Implement the initialization. You can either (a) initialize the server with the IDs of all the clients and have it notify them to start round 1 or (b) initialize the clients directly at round 1. *(4p)*

*Answer:*

```
start_server(Num) ->
    Pid = spawn(fun() -> exam:server(Num, 0 , 1 ,[]) end),
    catch unregister(server),
    register(server, Pid).

% This solution is parameterized for every possible number of clients
% This is not required from your solution.
% Parameters
% Total number of clients (does not change after initialization)
% Number of messages received in this round
% Number of round (message to be received)
% List of Id-s of clients who already responded
server(S,S,100,List) ->
    % send exit message to all clients
    [ Id ! { exit } || Id <- List ];

server(S,S,M,List) when M < 100 ->
     % initiate round M+1
     [ Id ! { M + 1, self() } || Id <- List ],
     server(S, 0, M + 1,[]);

server(S,N,M,List) when N < S ->
     % collect the finish notification from one client. Log their id.
     receive
        {M, From} ->
             server(S, N + 1, M, List ++ [From]);
        {_, _} -> exit(1);
        { exit }  -> exit;
        { _ } -> exit(1)
     end.

start_client() ->
    Pid = spawn(fun() -> exam:client() end),
    Pid ! {1, server }.

start_client(1) ->
    start_client();

start_client(Num) when Num > 1 ->
    _ = Num,
    start_client(),
    start_client(Num - 1).

client() ->
  receive
           { Msg, From } ->
          % this is the part where the client does the work.
          From ! { Msg , self() },
                client();
     { exit } -> exit;
     { _ } -> exit(1)
        end.

main() ->
    Num = 10,
    start_server(Num),
    start_client(Num).
```

**Q5** (19p). You are designing two classes that are supposed to access the same resource as readers and writers. Writers repeatedly (forever) compute data and then write to the resource. Readers repeatedly (forever) read from the resource and then process the data they collected. Readers can work on the same data several times if it has not been changed since their last access. Readers can all access the resource simultaneously but writers need exclusive access.

Here is a code skeleton that includes both classes and their initialization.

```
class ReadersWriters {
    final static int NumReaders = 5;
    final static int NumeWriters = 2;

    // Synchronization declarations to be defined...

    static class Reader extends Thread {
        public void run() {
           // add synchronization
           while (true) {
              read();
              process();
           }
        }

        private void read() {
        // This function reads from the resource. // No need to implement it.
        }

        private void process() {
        // This function processes the data acquired.
        // It may take a long time. // No need to implement it.
        }
    }

    static class Writer extends Thread {
        public void run() {
           // add synchronization
           while (true) {
              compute();
              write();
           }
        }
        private void compute() {
        // This function prepares data to be written to the resource.
        // It may take a long time. // No need to implement it.
        }

        private void write() {
        // This function writes to the resource. // No need to implement it.
        }
    }

    // Starting the readers and writers
    public static void main(String[] args) {
        for (int i = 0; i<NumReaders; i++) {
            new Reader().start();
        }
        for (int i = 0; i<NumWriters; i++) {
            new Writer().start();
        }
   }
}
```

Your task is to implement the following parts:

**(Part a)**. Write the declarations of the variables you will use for synchronization (you may use either locks or semaphores). Pay attention to types, initialization, and scope. *(3p)*

**(Part b)**. Complete the implementation of the method `run()` of the class `Reader` according to the description above. *(5p)*

**(Part c)**. Complete the implementation of the method `run()` of the class `Writer` according to the description above. *(5p)*

**(Part d)**. You increase the number of readers to 1000 and notice that writers never get a chance to change the resource. Add an additional mechanism that will block new readers from accessing the resource once a writer is ready to replace the data. *(6p)*

*Answer:*

Variable definitions (including for part d):

```
    static Lock lock = new ReentrantLock();
    static Integer permits = NumReaders;
    static Integer waiting = 0;
```

The run part of the reader:

```
        public void run() {
            while (true) {
                boolean acquired = false;
                lock.lock(); {
                    if (permits > 0 && waiting == 0) {
                        permits = permits - 1;
                        acquired = true;
                    }
                } lock.unlock();

                if (acquired) {
                    read();
                    lock.lock();
                    permits = permits + 1;
                    lock.unlock();
                    compute();
                } } }
```

The run part of the writer:

```
        public void run() {
            boolean imwaiting = false;
            while (true) {
                boolean acquired = false;
                produce();

                lock.lock(); {
                    if (!imwaiting) {
                        imwaiting = true;
                        waiting = waiting + 1;
                    }
                    if (permits >= NumReaders) {
                        permits = permits - NumReaders;
                        waiting = waiting - 1;
                        imwaiting = false;
                        acquired = true;
                    }
                } lock.unlock();

                if (acquired) {
                    write();
                    lock.lock(); {
                        permits = permits + NumReaders;
                    } lock.unlock();
```

```
} } }
```