Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Thursday, 26 October 2022

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

(Exam set by N. Piterman, based on the course given September-October 2022)

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

Visits to the exams rooms: The examiner will not be able to visit the exam rooms. Teaching assistants will visit the exam rooms instead. Invigilators are encourated to allow students to call the examiner for questions.

Grading: You can score a maximum of 70 points. Exam grades are: between 28–41 (3), between 42–55 (4), 56 or more (5).

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows: between 40–59 (3), between 60-79 (4), 80 or more (5).

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; 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 (10p). On the next page there is an excerpt from the implementation of the simple *software transactional memory* machine presented in lecture 11. The full code is available in appendix.

Currently, the implementation does not distinguish between variables that are used for reading and for writing. That is, even if a variable is unchanged, when the result of a transaction is committed it will get a new version number.

(Part a). Change the program above so that a transaction will keep track of which variables actually changed and increase the version number of only those that have changed.

(5p)

Answer: here is one possible implementation.

```
1 % Variable: {name, version, value, changed}
2 -record(var, {name, version = 0, value = undefined, changed = false}).
3
4 % read value of variable
5 read(#var{value = Value}) ->
6   Value.
7
8 % write 'Value' to 'Var'
9 write(Var = #var{}, Value) ->
10   Var#var{value = Value, changed = true}.
11
12 stm(State = #state{storage = Storage},
13     {push, Vars}) ->
14   case try_push(Vars, Storage) of
15     {success, NewStorage} ->
16       {reply, State#state{storage =
17                 NewStorage}, success};
18     fail ->
19       {reply, State, fail}
20   end.
21
22 try_push([], Storage) ->
23   {success, Storage};
24 try_push([Var = #var{name = Name, version = Version, changed = true} | Vars],
25           Storage) ->
26   case dict:find(Name, Storage) of
27     {ok, #var{version = Version}} ->
28       try_push(Vars,
29                dict:store(Name, Var#var{version = Version + 1}, Storage));
30     _ -> fail
```

```
31   end;
32 try_push([Var = #var{name = Name, version = Version, changed = false} | Vars],
33           Storage) ->
34   case dict:find(Name, Storage) of
35     {ok, #var{version = Version}} ->
36       try_push(Vars,Storage);
37     _ -> fail
38   end.
```

(Part b). Give an example of transactions and their execution order that will expose the difference between the new and old implementations. *(5p)*

Answer:

For example, suppose that one process does:

```
1  Vara = pull(Server,a),
2  Varb = pull(Server,b),
3  Modb = write(Varb,read(Vara)+1),
4  push(Server,[Modb,Vara]).
```

and another process does:

```
1  Vara = pull(Server,a),
2  Varc = pull(Server,c),
3  Modc = write(Varc,read(Vara)+1),
4  push(Server,[Modc,Vara]).
```

In the original implementation which ever process pushes second will fail. In the new implementation both can push their results.

```
% Variable: {name, version, value}
-record(var, {name, version = 0, value = undefined}).

% check out variable 'Name' from 'Tm'
pull(Tm, Name) ->
  ...

% commit all variables in list 'Vars' to 'Tm'
push(Tm, Vars) when is_list(Vars) ->
  ...

% read value of variable
read(#var{value = Value}) ->
  Value.

% write 'Value' to 'Var'
write(Var = #var{}, Value) ->
  Var#var{value = Value}.

stm(State = #state{storage = Storage},
    {push, Vars}) ->
  case try_push(Vars, Storage) of
    {success, NewStorage} ->
      {reply, State#state{storage =
              NewStorage}, success};
    fail ->
      {reply, State, fail}
  end.

try_push([], Storage) ->
  {success, Storage};
try_push([Var = #var{name = Name, version = Version} | Vars],
         Storage) ->
  case dict:find(Name, Storage) of
    {ok, #var{version = Version}} ->
      try_push(Vars,
               dict:store(Name, Var#var{version = Version + 1}, Storage));
    _ -> fail
  end.
```

Q2 (18p). In the bounded buffer problem, multiple producers and multiple consumers have access to a shared resource and collaborate in adding items to it (producers) and removing items from it (consumers).

If only one producer and one consumer are available, it is possible to implement a simpler data structure, where two bounded buffers are allocated, inbuffer and outbuffer. The producer continously produces new data and add it to infbuffer until inbuffer is full. The consumer continuously consumes data from outbuffer until outbuffer is empty. When inbuffer is full and outbuffer is empty the two buffers are swapped.

Your task is to implement a class supporting this access pattern. A skeleton for the code and the way it would be used by two threads is available on the next page.

Notice that the code handles all exceptions by throwing them further, so you do not have to handle exceptions.

Hint: Think about *a*ll synchronization mechanisms we have learned.

Notice: Solutions should allow the producer/consumer to produce/-consume *a*ll the buffer without synchronization. Synchronization should occur only at points of exchange of buffers.

(Part a). Write the declarations of the variables you will use for synchronization. Pay attention to types, initialization, and scope. *(4p)*

(Part b). Complete the implementation of the constructor. *(2p)*

(Part c). Complete the implementation of the method `read` according to the description above. *(6p)*

(Part d). Complete the implementation of the method `write` according to the description above. *(6p)*

Credits: this question is modified from an exam given at the university of Twente.

The rest of this page is left blank on purpose.

```
1  public class ProducerConsumer {
2      private int[] inbuffer, outbuffer;
3      private int readloc, writeloc;
4      final int size = 10;
5
6      // Introduce synchronization
7
8      ProducerConsumer() {
9        inbuffer = new int[size];
10       outbuffer = new int[size];
11         // Finalize initialization
12     }
13
14     int read() throws ... {
15         // Implement
16         return value;
17     }
18
19     void write(int val) throws ...  {
20         // Implement
21     }
22
23     public static void main(String[] args) {
24
25       ProducerConsumer pc = new ProducerConsumer();
26
27       final Thread producer = new Thread(() -> {
28         while (true) {
29           try {
30             int val = 0; // produce
31             pc.write(val);
32           } catch (... e) {
33             throw new RuntimeException(e);
34           }});
35
36       final Thread consumer = new Thread(() -> {
37             while (true) {
38                 try {
39                     int val = pc.read(); // consume
40                 } catch (... e) {
41                     throw new RuntimeException(e);
42                 }});
43
44         producer.start();
45         consumer.start();
46     }
47 }
```

Answer:

```
1 import java.util.concurrent.BrokenBarrierException;
2 import java.util.concurrent.CyclicBarrier;
3
4 public class ProducerConsumer {
5     private int[] inbuffer;
6     private int[] outbuffer;
7
8     private final CyclicBarrier barrier;
9     private int readloc;
10    private int writeloc;
11    final int size = 10;
12
13    ProducerConsumer() {
14      inbuffer = new int[size];
15      outbuffer = new int[size];
16      barrier = new CyclicBarrier(2);
17      readloc=size;
18      writeloc=0;
19    }
20
21    int read() throws InterruptedException, BrokenBarrierException {
22      if (readloc == size) {
23        barrier.await();
24        writeloc = 0;
25        readloc=0;
26        int[] tempbuffer = inbuffer;
27        inbuffer = outbuffer;
28        outbuffer = tempbuffer;
29
30        barrier.await();
31      }
32      System.out.println("Read " + readloc);
33      return outbuffer[readloc++];
34    }
35
36    void write(int val) throws InterruptedException, BrokenBarrierException {
37      if (writeloc == size) {
38        barrier.await();
39        barrier.await();
40      }
41      System.out.println("Wrote " + writeloc);
42      inbuffer[writeloc++]=val;
43    }
44
45    public static void main(String[] args) {
46        // as above
47    }
48 }
```

Q3 (14p). We include below the implementation of a Queue that uses locking to ensure integrity of the data structure.

```
1 class SequentialNode<T> implements Node<T>
2 {
3   public T item; // value stored in node
4   public Node<T> next; // next node in chain
5
6   SequentialNode<T>(T item,Node<T> next) {
7     this.item = item;
8     this.next = next;
9 } }
10
11 class LockQueue<T> implements Queue<T>
12 { // access to front and back of queue
13   private Node<T> head = new SequentialNode<T>();
14   private Node<T> tail = new SequentialNode<T>();
15
16   // empty queue
17   public LockQueue() {
18     // value of sentinel does not matter
19     SequentialNode<T> sentinel = new SequentialNode<T>();
20     head.next = sentinel; tail.next = sentinel;
21   }
22
23   public void enqueue(T value) {
24     SequentialNode<T> newNode = new SequentialNode<T>(value,null);
25     synchronized (tail) {
26       SequentialNode<T> last = tail.next();
27       last.next = newNode;
28       tail.next = newNode;
29   } }
30
31   public T dequeue() throws EmptyException {
32     synchronized (head) {
33       SequentialNode<T> sentinel = head.next;
34       SequentialNode<T> first = sentinal.next;
35       if (first == null) throw new EmptyException();
36       T value = first.item;
37       head.next = first;
38     }
39     return value;
40 } }
```

This queue uses the same structure of the LockFreeQueue. It stores ref-

erences to the head and the tail of the queue. Furthermore, the linked list representing the queue is never empty; its first element (the sentinel) is always present but is not part of the queue. Thus, a linked list that contains only the sentinel represents the empty queue. However, **it uses normal reference and NOT atomic references**.

(Part a) Is it possible for two threads performing enqueue to be in a data race? What would be the consequences of the data race? *(3p)*

Answer: No. The tail of the list is fixed for an instance of the queue. Two enqueues effectively occur sequentially as they synchronize on the tail.

(Part b) Is it possible for two threads performing dequeue to be in a data race? What would be the consequences of the data race? *(3p)*

Answer: No. The head of the list is fixed for an instance of the queue. Two dequeues effectively occur sequentially as they synchronize on the head.

(Part c) Is it possible for a thread performing enqueue and a thread performing dequeue to be in a data race? What would be the consequences of the data race? *(6p)*

Answer: Yes. Consider the scenario where the queue is empty. That is, the linked list contains only the sentinel. The enqueuing thread (e) locks the tail and the dequeueing thread (d) locks the head. Both threads access the sentinel, which is the same memory location. Now, e writes to the sentinel.next while d reads the sentinel.next with no synchronization between them. The effect on the behaviour of the queue could be that the dequeue throws an exception when an items have already been added by enqueue (or multiple enqueues).

(Part d) Suggest how to fix the program so that the problems highlighted in (a), (b), and (c) above are resolved. *(4p)*

Answer: Make the next field volatile. This would mean that there is synchronization between writing and the reading.

Q4 (16p). In this question we create an Erlang "gate". This is a device that is either open or closed. When it is open it allows other processes to pass. When it is closed it blocks processes until it is opened. This is similar to the Erlang barrier shown in class whose code is given in the next page.

Implement an Erlang gate that has the following interface:

```
1  -module(gate).
2  -export([init/1,wait/1,allow/1,block/1]).
3
4  % initialize an open gate
5  init() ->
6     ...
7
8  % if the gate is open continue
9  % if the gate is closed block until the gate is opened
10 wait(Gate) ->
11    ...
12
13 % open the gate
14 open(Gate) ->
15    ...
16
17 % close the gate
18 close(Gate) ->
19    ...
```

That is, you should implement a server that is initialized as an open gate.

An open gate discards all open messages it receives. Once it receives a close message it closes. It allows arrivals at the gate to immediately pass through the gate.

A closed gate discards all close messages it receives. It blocks all arrivals at the gate. Once it receives an open message it releases all those who are waiting for the gate to open and becomes open.

(Part a). Implement the init function. *(2p)*

(Part b). Implement the wait function. *(2p)*

(Part c). Implement the open and close functions. *(4p)*

(Part d). Implement the server loop. *(8p)*

Credits: this question is modified from an exam given at the university of Twente.

Answer:

Here is an implementation:

```
1  -module(gate).
2  -export([init/1,wait/1,allow/1,block/1]).
3
4  % initialize gate for 'Expected' processes
5  init() ->
6     spawn(fun () -> gate(open) end).
7
8  % Open gate
9  % - First discard further open instructions
10 % - Then handle block instructions
11 % - Then allow arrivals to pass
12 gate(open) ->
13   receive
14      open ->
15          gate(open);
16      close ->
17          gate(closed, 0, []);
18      { arrived, From, Ref } ->
19          From ! { continue, Ref },
20          gate(open)
21   end;
22 % Closed gate
23 % - First discard further open instructions
24 % - Then handle block instructions
25 % - Then allow arrivals to pass
26 % Closed gate accumulates arrivals gives priority to open
27 % discared further blocking instructions
28 gate(closed,Num, PidRefs) ->
29   receive
30      close ->
31          gate(closed, Num, PidRefs);
32      open ->
33          [ To ! { continue, Ref} || {To, Ref} <- PidRefs],
34          gate(open);
35      { arrived, From, Ref } ->
36          gate(closed, Num+1, [ {From, Ref} | PidRefs])
37   end.
38
39 % block at 'Gate' until all processes have reached it
40 wait(Gate) ->
41    Ref = make_ref(),
42      % notify gate of arrival
```

```
43    Gate ! {arrived, self(), Ref},
44      % wait for signal to continue
45    receive {continue, Ref} -> through end.
46
47 open(Gate) ->
48    Gate ! open.
49
50 close(Gate) ->
51    Gate ! close.
```

For reference, here is the implementation of the Barrier.

```
1  -module(barrier).
2  -export([init/1,wait/1]).
3
4  % initialize barrier for 'Expected' processes
5  init(Expected) ->
6     spawn(fun () -> barrier(0, Expected, []) end).
7
8  % event loop of barrier for 'Expected' processes
9  %   Arrived: number of processes arrived so far
10 %   PidRefs: list of {Pid, Ref} of processes arrived so far
11 barrier(Arrived, Expected, PidRefs)
12    when Arrived =:= Expected ->     % all processes arrived
13      % notify all waiting processes
14      [To ! {continue, Ref} || {To, Ref} <- PidRefs],
15      % reset barrier
16      barrier(0, Expected, []);
17 barrier(Arrived, Expected, PidRefs) ->
18    receive  % still waiting for some processes
19       {arrived, From, Ref} ->
20
21         % one more arrived: add {From, Ref} to PidRefs list
22         barrier(Arrived+1, Expected, [{From, Ref}|PidRefs])
23    end.
24
25 % block at 'Barrier' until all processes have reached it
26 wait(Barrier) ->
27    Ref = make_ref(),
28      % notify barrier of arrival
29    Barrier ! {arrived, self(), Ref},
30      % wait for signal to continue
31    receive {continue, Ref} -> through end.
```

Q5 (12p). The following program is a failed attempt at producing mutual exclusion between two threads using only shared variables. You are requested to analyze the transition table of this concurrent program.

boolean `flagp`= false; **boolean** `flagq`= false;			
	`p`		`q`
p_1	**while**(true) {	q_1	**while**(true) {
p_2	**await**(!`flagq`);	q_2	**await**(!`flagp`)
p_3:	`flagp`= true;	q_3:	`flagq`= true;
p_4:	// *critical section*	q_4:	*critical section*
p_5:	`flagp`= false;	q_5:	`flagq`= false;
p_6:	}	q_6:	}

The state of the program is a quadruple $(p_i, q_j, \texttt{flagp}, \texttt{flagq})$, where i and j range over $\{2, 3, 5\}$, and `flagp` and `flagq` are Booleans.

Here is a partial state transition table for the program above. Only 9 states are reachable from the initial state $(p_2, q_2, false, false)$.

state		new state if p moves	new state if q moves
s1	$(2, 2, false, false)$	FILL THIS ENTRY	FILL THIS ENTRY
s2	$(2, 3, false, false)$	FILL THIS ENTRY	FILL THIS ENTRY
s3	$(2, 5, false, true)$	FILL THIS ENTRY	FILL THIS ENTRY
s4	$(3, 2, false, false)$	FILL THIS ENTRY	FILL THIS ENTRY
s5	$(3, 3, false, false)$	FILL THIS ENTRY	FILL THIS ENTRY
s6	$(3, 5, false, true)$	FILL THIS ENTRY	FILL THIS ENTRY
s7	$(5, 2, true, false)$	FILL THIS ENTRY	FILL THIS ENTRY
s8	$(5, 3, true, false)$	FILL THIS ENTRY	FILL THIS ENTRY
s9	$(5, 5, true, true)$	FILL THIS ENTRY	FILL THIS ENTRY

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

Answer:

state		new state if p moves	new state if q moves
s1	$(2, 2, false, false)$	$(3, 2, false, false)$ $(s4)$	$(2, 3, false, false)$ $(s2)$
s2	$(2, 3, false, false)$	$(3, 3, false, false)$ $(s5)$	$(2, 5, false, true)$ $(s3)$
s3	$(2, 5, false, true)$	no move $(s3)$	$(2, 2, false, false)$ $(s1)$
s4	$(3, 2, false, false)$	$(5, 2, true, false)$ $(s7)$	$(3, 3, false, false)$ $(s5)$
s5	$(3, 3, false, false)$	$(5, 3, true, false)$ $(s8)$	$(3, 5, false, true)$ $(s6)$
s6	$(3, 5, false, true)$	$(5, 5, true, true)$ $(s9)$	$(3, 2, false, false)$ $(s4)$
s7	$(5, 2, true, false)$	$(2, 2, false, false)$ $(s1)$	no move $(s7)$
s8	$(5, 3, true, false)$	$(2, 3, false, false)$ $(s2)$	$(5, 5, true, true)$ $(s9)$
s9	$(5, 5, true, true)$	$(2, 5, false, true)$ $(s3)$	$(5, 2, true, false)$ $(s7)$

(Part b) Explain how to see from your table that the protocol does not fulfil mutual exclusion. *(2p)*

Answer: State $s9$ is reachable in the table.

(Part c) Explain what goes wrong (a very short explanation is enough). *(2p)*

Answer: If both threads check that the other thread is not interested at the same time (i.e., move from 2 to 3) then they both have already made it possible to go into the critical section.

A Full code listings

A.1 Code for Q1

```
1 -module(stm).
2 -compile(exporet_all).
3
4 % Variable: {name, version, value}
5 -record(var, {name, version = 0, value = undefined}).
6
7 % initialize empty transactional memory and register as 'Tm'
8 start(Tm) ->
9   Storage = dict:new(),  % empty key/value map
10  register(Tm,
11    gserver:start(#state{storage = Storage},
12                  fun stm/2)).
13
14 % shutdown 'Tm'
15 stop(Tm) ->
16   gserver:stop(Tm).
17
18 % create variable 'Name' in 'Tm'
19 % using initial 'Value'
20 % return:
21 %  ok -> success
22 %  skip -> variable 'Name' already exists
23 create(Tm, Name, Value) ->
24   gserver:request(Tm, {create, Name, Value}).
25
26 % create variable 'Name' in 'Tm'
27 % with 'undefined' initial value
28 % return:
29 %  ok -> success
30 %  skip -> variable 'Name' already exists
31 create(Tm, Name) ->
32   create(Tm, Name, undefined).
33
34 % remove variable 'Name' from 'Tm'
35 % return:
36 %  ok -> success
37 %  skip -> variable 'Name' already exists
38 drop(Tm, Name) ->
39   gserver:request(Tm, {drop, Name}).
40
```

```
41 % check out variable 'Name' from 'Tm'
42 % return:
43 %  variable -> variable 'Name' found
44 %  not_found -> variable 'Name' not found
45 pull(Tm, Name) ->
46   gserver:request(Tm, {pull, Name}).
47
48 % commit all variables in list 'Vars' to 'Tm'
49 % return:
50 %  success -> changes committed
51 %  fail -> changes not committed, abort
52 push(Tm, Vars) when is_list(Vars) ->
53   gserver:request(Tm, {push, Vars});
54 % commit variable Var to 'Tm'
55 push(Tm, Var) ->
56   push(Tm, [Var]).
57
58 % read value of variable
59 read(#var{value = Value}) ->
60   Value.
61
62 % write 'Value' to 'Var'
63 write(Var = #var{}, Value) ->
64   Var#var{value = Value}.
65
66 stm(State = #state{storage = Storage},
67     {create, Name, Value}) ->
68   case dict:is_key(Name, Storage) of
69     true ->
70       % variable 'Name' already exists
71       {reply, State, skip};
72     false ->
73       % add variable
74       Var = #var{name = Name, value = Value},
75       {reply, State#state{storage =
76               dict:store(Name, Var, Storage)}, ok}
77   end;
78
79 stm(State = #state{storage = Storage},
80     {drop, Name}) ->
81   case dict:is_key(Name, Storage) of
82     true ->
83       {reply, State#state{storage =
84               dict:erase(Name, Storage)}, ok};
```

```
85      false ->
86        % variable 'Name' does not exist
87        {reply, State, skip}
88    end;
89
90  stm(State = #state{storage = Storage},
91      {pull, Name}) ->
92    case dict:is_key(Name, Storage) of
93      true ->
94        {reply, State, dict:fetch(Name, Storage)};
95      false ->
96        {reply, State, not_found}
97     end;
98
99  stm(State = #state{storage = Storage},
100     {push, Vars}) ->
101   case try_push(Vars, Storage) of
102     {success, NewStorage} ->
103       {reply, State#state{storage =
104               NewStorage}, success};
105     fail ->
106       {reply, State, fail}
107   end.
108
109 try_push([], Storage) ->
110   {success, Storage};
111 try_push([Var = #var{name = Name, version = Version} | Vars],
112          Storage) ->
113   case dict:find(Name, Storage) of
114     {ok, #var{version = Version}} ->
115       try_push(Vars,
116                dict:store(Name, Var#var{version = Version + 1}, Storage));
117     _ -> fail
118   end.
```