Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Saturday, 24 October 2020

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

(Exam set by N. Piterman, based on the course given September-October 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 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 tex template are available on Canvas.

Q1 (8p). 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 <= 1024) {
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 }
```

Notice that the stopping condition is changed from `high - low <= 1` to `high - low <= 1024` (according to the advice of slides 23 and 24) but the forking thread is left idle.

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 7000 elements, answer the following.

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

Answer: there are $8 + 4 + 2 + 1 = 15$ *nodes.*

(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)*

Answer: 8 calls – the width of the dependency graph.

Q2 (17p). In this question we create a modified version of the readers-writers problem. You have to design a monitor class called TypedAccess. TypedAccess controls a resource that has two types of operations: A and B. Many A actions can occur at the same time and many B actions can occur at the same time. But you cannot allow As and Bs together to occur at the same time.

The monitor should support two methods *enter* and *exit*. They both get a parameter telling which type of access is required ('a' or 'b'). Busy waiting or polling is not allowed.

You should also design a client that uses your monitor.

A skeleton for both the TypedAccess and Client are given on the next page (this code is also available for you to download in Canvas).

Your task is to implement the following parts:

(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 method `enter(int)` according to the description above. *(5p)*

(Part c). Complete the implementation of the method `exit(int)` according to the description above. *(5p)*

(Part d). Complete the implementation of the Client run method. *(3p)*

The rest of this page is left blank on purpose.

```
1  public class MultiAccess {
2
3    private TypedAccess monitor;
4
5    public MultiAccess() {
6      this.monitor =  new TypedAccess();
7    }
8
9    class TypedAccess  {
10     // Introduce the variables that you need
11
12     TypedAccess() {
13       // Implemented the constructor
14     }
15
16     void enter(char type) {
17       // Implement the enter method
18     }
19
20     void exit(char type) {
21       // Implement the exit method
22     }
23
24   }
25
26   static class Client implements Runnable {
27     protected char type;
28     protected final TypedAccess l;
29
30     Client (char type, TypedAccess l) {
31       this.l = l;
32       this.type = type;
33     }
34
35     public void run() {
36       compute();
37     }
38
39     void compute() {
40       System.out.println("Client " + Thread.currentThread().getId() +
41           " of type " + this.type + " (" + Thread.currentThread().getName() + ")");
42     }
43   }
44 }
```

Answer:

```
1 import java.util.concurrent.locks.Lock;
2 import java.util.concurrent.locks.ReentrantLock;
3 import java.util.concurrent.locks.Condition;
```

```
4
5  public class MultiAccess2 {
6
7     static final int CLIENTS = 8;
8     static final int ROUNDS = 100;
9     private TypedAccess monitor;
10
11    public MultiAccess2() {
12       this.monitor =  new TypedAccess();
13    }
14
15    public static void main(String[] args) {
16       MultiAccess2 j = new MultiAccess2();
17       for (int i=0 ; i< CLIENTS ; ++i) {
18          Thread c = new Thread(new Client((i%2==0 ? 'a': 'b'),j.monitor));
19          c.start();
20       }
21    }
22
23    class TypedAccess  {
24       private final Lock monitor = new ReentrantLock(true);
25       private final Condition isAllowedA = monitor.newCondition();
26       private final Condition isAllowedB = monitor.newCondition();
27       private int aClients;
28       private int bClients;
29
30
31       TypedAccess() {
32          aClients=0;
33          bClients=0;
34       }
35
36       void enter(char type) throws InterruptedException {
37          monitor.lock();
38          try {
39             if (type == 'b') {
40                while (aClients > 0) {
41                   isAllowedB.await();
42                }
43                ++bClients;
44                return;
45             } else {
46                while (bClients > 0) {
47                   isAllowedA.await();
```

```
48              }
49              ++aClients;
50              return;
51           }
52        } finally {monitor.unlock(); }
53     }
54
55     void exit(char type) {
56        monitor.lock();
57        try {
58           if (type == 'b') {
59              --bClients;
60              if (bClients == 0) {
61                 // Notice that signal would work as well
62                 // because of spurious wake-ups
63                 isAllowedA.signalAll();
64              }
65              return;
66           } else {
67              --aClients;
68              if (aClients == 0) {
69                 isAllowedB.signalAll();
70              }
71              return;
72           }
73        } finally {monitor.unlock(); }
74     }
75
76  }
77
78  static class Client implements Runnable {
79     protected char type;
80     protected final TypedAccess l;
81     Client (char type, TypedAccess l) {
82        this.l = l;
83        this.type = type;
84     }
85
86     public void run() {
87        for (int i = 0 ; i<ROUNDS ; ++i) {
88           try {
89              this.l.enter(this.type);
90              compute();
91              this.l.exit(this.type);
```

```
92             } catch (InterruptedException e) {
93                System.out.println("Client " + Thread.currentThread().getId() +
94                                   " of type " + this.type + " exception"
   +
95                                   " (" + Thread.currentThread().getName() + ")");
96                e.printStackTrace();
97             }
98          }
99       }
100
101      void compute() {
102         System.out.println("Client " + Thread.currentThread().getId() +
103               " of type " + this.type + " (" + Thread.currentThread().getName() + ")
104      }
105   }
106 }
```

Q3 (12p). The pseudo-code below is a simplified version of Lamport's bakery algorithm. It is presented for two processes and with an additional atomicity assumption. The bakery algorithm solves the critical section (CS) problem for every (known) number of processes. However, it allocates an (unbounded) integer per process (and additional Boolean variables).

int n[2]= {0,0};			
	p		q
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
p_2:	n[0] = n[1]+1;	q_2:	n[1] = n[0]+1;
p_3:	**while**(n[1]>0 && n[1]<n[0]) { };	q_3:	**while**(n[0]>0 && n[0]<= n[1]) { };
p_4	*//CS (critical section)*	q_4	*//CS (critical section)*
p_5:	n[0]=0;	q_5:	n[1]=0;
	}		}

Notice that the instructions in p_2 and q_2 are executed *atomically* (!). For simplicity, we consider only locations 2, 3, and 5. So, for example, p starts in location p_2 (considered the non-critical section), it moves from p_3 to p_5 directly and from p_5 to p_2 directly, and p_5 is considered the 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.

(Part a) Show that $\mathsf{n}[0] \geq 0 \wedge \mathsf{n}[1] \geq 0$ 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 both processes. *(2p)*

Answer: It holds initially as n *is set to* $\{0, 0\}$. *There are two possible changes to* n. *In* p_2 *the value of* $\mathsf{n}[0]$ *is set to* $\mathsf{n}[1] + 1$. *If the invariant holds before executing* p_2 *then* $\mathsf{n}[1]$ *is non-negative and then the new value of* $\mathsf{n}[0]$ *is positive. The change in* q_2 *is similar.*

In p_5 *the value of* $\mathsf{n}[0]$ *is set to* 0. *The change in* q_5 *is similar.*

(Part b) Show that $((p_3 \vee p_5) \iff \mathsf{n}[0] > 0)$ is an invariant of the program. Show that it holds initially and that it is preserved under every transition of process p. *(2p)*

Answer: It holds initially. As $\mathsf{n}[1] \geq 0$, *the assignment* $\mathsf{n}[0] = \mathsf{n}[1] + 1$ *sets* $\mathsf{n}[1] > 0$. *The other change to* $\mathsf{n}[0]$ *is when executing* p_5, *which sets* $\mathsf{n}[0]$ *to* 0.

Use the invariant $((q_3 \vee q_5) \iff \mathsf{n}[1] > 0)$ without proof.

(Part c) Show that if $(p_3 \vee p_5) \wedge (\mathsf{n}[1] = 0 \vee \mathsf{n}[1] \geq \mathsf{n}[0])$ then whatever q does cannot change that. That is, if p is in locations p_3 or p_5 and it happens to be that either $\mathsf{n}[1] = 0$ or $\mathsf{n}[1] \geq \mathsf{n}[0]$ then all transitions of q maintain this. *(4p)*

Answer: From (b) above we know that $\mathsf{n}[0] > 0$. *Consider the transitions of* q *that change the value of* $n[1]$.

If q_2 *is executed then* $\mathsf{n}[1]$ *becomes larger than* $\mathsf{n}[0]$ *reestablishing the statement. If* q_5 *is executed then* $\mathsf{n}[1]$ *becomes* 0.

Similarly, p cannot change the truth value of $(q_3 \vee q_5) \wedge (\mathsf{n}[0] = 0 \vee \mathsf{n}[0] > \mathsf{n}[1])$. You can use this without a proof.

(Part d) Show that the program satisfies mutual exclusion. *(4p)*

Answer: Suppose that p *is the first to enter the critical section. When it enters the critical section it holds that* $\mathsf{n}[1] = 0$ *or* $\mathsf{n}[1] >= \mathsf{n}[0]$. *Then, the statement in (c) holds.*

In order for q *to enter the critical section it must be the case that* $\mathsf{n}[0] = 0$ *or* $\mathsf{n}[0] > \mathsf{n}[1]$. *However, as* p *is in the critical section* $\mathsf{n}[0] > 0$. *So,* q *must check that* $\mathsf{n}[0] > \mathsf{n}[1]$. *For* q *to enter the critical section* $\mathsf{n}[1]$ *must be positive as well from (b). Then, it cannot happen that at the same time* $\mathsf{n}[1] >= \mathsf{n}[0]$ *and* $\mathsf{n}[0] > \mathsf{n}[1]$.

The case where q *is the first to enter the critical setion is similar.*

Q4 (16p). In this question we create a modified barrier in Erlang. The Erlang barrier we have seen in class is below (this code is also available for you to download in Canvas):

```
-module(barrier).
-export([init/1,wait/1]).

% initialize barrier for `Expected' processes
init(Expected) ->
   spawn(fun () -> barrier(0, Expected, []) end).

% event loop of barrier for `Expected' processes
%   Arrived: number of processes arrived so far
%   PidRefs: list of {Pid, Ref} of processes arrived so far
barrier(Arrived, Expected, PidRefs)
   when Arrived =:= Expected ->     % all processes arrived
     % notify all waiting processes
     [To ! {continue, Ref} || {To, Ref} <- PidRefs],
     % reset barrier
     barrier(0, Expected, []);
barrier(Arrived, Expected, PidRefs) ->
   receive  % still waiting for some processes
      {arrived, From, Ref} ->
        % one more arrived: add {From, Ref} to PidRefs list
        barrier(Arrived+1, Expected, [{From, Ref}|PidRefs])
   end.

% block at `Barrier' until all processes have reached it
wait(Barrier) ->
   Ref = make_ref(),
     % notify barrier of arrival
   Barrier ! {arrived, self(), Ref},
     % wait for signal to continue
   receive {continue, Ref} -> through end.
```

This barrier ensures that the difference in number of completed rounds between clients is at most 1. That is, the most advanced client can complete a round while the least advanced client has not started it yet.

Change the behaviour of the barrier so that the difference in number of rounds between different clients is at most 2 and not at most 1. This means, for example, that a client that is 2 rounds behind all other clients could do 4 rounds in a row becoming 2 rounds before all the others.

You should not change the `wait` function.

Notice that if your new barrier is not tail recursive you will not be awarded the maximal points.

(Part a). Implement the server. Explain the role of the elements of the server's state. *(8p)*

In order to test your barrier you need to try it with several clients. Design a client program and a client coordinator that will enable to test the barrier. The client should approach the barrier N times. The "work" that it should do before getting to the barrier is print the client's ID and its round number (in descending order: N, N-1, ..., 1). The client coordinator, should get the value N and a list of 2N values in $\{1, 2\}$, it should initialize two clients and try to schedule their runs so that they perform the actions in this order. For example, if the sequence is $[1, 1, 1, 1, 2, 2, 2, 2]$ client 1 should try to access the barrier 4 times (it will be stopped after 2) and then client 2 will access the barrier 4 times. A more interesting example would be $[1, 1, 2, 2, 2, 2, 1, 1]$ that should result in exactly the sequence of accesses $[1, 1, 2, 2, 2, 2, 1, 1]$.

(Part b). Implement the client and its initialization function. Explain the client state (if exists). *(4p)*

(Part c). Implement the coordinator. There is no need for the coordinator to check the consistency of its parameters. *(4p)*

Answer:

```
% event loop of barrier with diff 2 for 'Expected' processes
%   ArrivedOne: number of processes arrived so far once
%   ArrivedTwo: number of processes arrived so far twice
%   Expected: number of processes using the barrier
%   PidsOne: List of PIDs of those that arrived once
%   PidRefsTwo: List of {Pid,Ref} of those arrived twice
barrierb(ArrivedOne,ArrivedTwo,Expected,_PidsOne,PidRefsTwo)
   when ArrivedOne+ArrivedTwo =:= Expected ->
      [To ! {continue, Ref} || {To, Ref} <- PidRefsTwo],
      NewArrivedOne = [ To || {To,_Ref} <- PidRefsTwo],
      barrierb(ArrivedTwo,0,Expected,NewArrivedOne,[]);
barrierb(ArrivedOne,ArrivedTwo,Expected,PidsOne,PidRefsTwo) ->
   receive
      {arrived, From, Ref} ->
          barrierbsearch(ArrivedOne,ArrivedTwo,Expected,PidsOne,
                                              [],PidRefsTwo,From,Ref)
   end.

% search in the list of arrived once for From
% If it is there then this is the second arrival and add it to
% second list
% If it is not there this is the first arrival and add it to
% first list
barrierbsearch(ArrivedOne,ArrivedTwo,Expected,[H|T],L,PidRefsTwo,
                                                          From,Ref)
   when H =:= From ->
      barrierb(ArrivedOne-1,ArrivedTwo+1,Expected,T++L,
                                          [{From,Ref}|PidRefsTwo]);
barrierbsearch(ArrivedOne,ArrivedTwo,Expected,[H|T],L,PidRefsTwo,
                                                       From,Ref) ->
   barrierbsearch(ArrivedOne,ArrivedTwo,Expected,T,[H|L],
                                             PidRefsTwo,From,Ref);
barrierbsearch(ArrivedOne,ArrivedTwo,Expected,[],L,
                                             PidRefsTwo,From,Ref) ->
   From ! {continue, Ref},
   barrierb(ArrivedOne+1,ArrivedTwo,Expected,[From|L],PidRefsTwo).
```

```
initcli(Barrier,N,Id) ->
   spawn(fun() -> client(Barrier,N,Id) end).

client(_Barrier,0,_Id) -> ok;
client(Barrier,N,Id) ->
   receive _ ->
      io:fwrite("~p: ~p~n",[Id,N]),
      wait(Barrier),
      client(Barrier,N-1,Id)
   end.

coordinator(Barrier,N,L) ->
   Cli1 = initcli(Barrier,N,client1),
   Cli2 = initcli(Barrier,N,client2),
   cloop(Cli1,Cli2,L).

cloop(Cli1,Cli2,[1|T]) ->
   Cli1 ! self(),
   timer:sleep(100),
   cloop(Cli1,Cli2,T);
cloop(Cli1,Cli2,[2|T]) ->
   Cli2 ! self(),
   timer:sleep(100),
   cloop(Cli1,Cli2,T);
cloop(Cli1,Cli2,[]) ->
   {Cli1,Cli2}.
```

Q5 (17p). The following program is an attempt to implement a Barrier with compare-and-swap. Recall that a compare-and-swap operation (cas) gets two parameters. It *atomically* compares the value of the variable to the first parameter and, if they are equivalent updates the variable to the second parameter and returns true. If they are not equivalent it does not change the value of the variable and returns false. You are requested to analyze the transition table of this concurrent program to check what went wrong.

int b= 0;			
	p		q
	while(true) {		**while**(true) {
p_1	*//Pre entrance*	q_1	*//Pre entrance*
p_2	**do**{	q_2	**do**{
p_3:	l_p= b;	q_3:	l_q= b;
p_4:	} **while**(!b.cas(l_p,l_p+1));	q_4:	} **while**(!b.cas(l_q,l_q+1));
p_5:	**while**(!b.cas(2,0) { };	q_5:	while(b>0) { };
p_6	*//Post entrance*	q_6	*//Post entrance*
	}		}

The state of the program is a quintuple, $(p_i, q_j, \mathtt{b}, \mathtt{l}_p, \mathtt{l}_q)$, where i and j range over $\{3, 4, 5\}$, and $\mathtt{b}, \mathtt{l_p}, \mathtt{l_q}$ are integers. For simplicity, whenever $\mathtt{l}_p$ and $\mathtt{l}_q$ are not defined (i.e., locations 3 and 5) we denote their value by $\perp$. It so turns out that only 14 states are reachable.

Here is a partial state transition table for the program above. As mentioned, only 14 states are reachable from the initial state $(p_3, q_3, 0, \perp, \perp)$.

state		new state if p moves	new state if q moves
s1	$(3, 3, 0, \perp, \perp)$	$(4, 3, 0, 0, \perp) = s5$	$(3, 4, 0, \perp, 0) = s2$
s2	$(3, 4, 0, \perp, 0)$		$(3, 5, 1, \perp, \perp) = s4$
s3	$(3, 5, 0, \perp, \perp)$		
s4	$(3, 5, 1, \perp, \perp)$		no move ($s4$)
s5	$(4, 3, 0, 0, \perp)$	$(5, 3, 1, \perp, \perp) = s10$	
s6	$(4, 4, 0, 0, 0)$		
s7	$(4, 5, 0, 0, \perp)$		
s8	$(4, 5, 1, 1, \perp)$		no move ($s8$)
s9	$(4, 5, 1, 0, \perp)$		
s10	$(5, 3, 1, \perp, \perp)$	no move ($s10$)	
s11	$(5, 4, 1, \perp, 0)$		
s12	$(5, 4, 1, \perp, 1)$	no move ($s12$)	
s13	$(5, 5, 1, \perp, \perp)$		
s14	$(5, 5, 2, \perp, \perp)$		

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

(Part b) Point out what goes wrong in the protocol (a very short explanation is enough). *(2p)*

(Part c) Indicate one transition in the table that needs to be removed for the protocol to work correctly. Explain how does this solution capture the notion of a barrier. *(7p)*

Answer:

state		*new state if p moves*	*new state if q moves*
s1	$(3,3,0,\perp,\perp)$	$(4,3,0,0,\perp) = s5$	$(3,4,0,\perp,0) = s2$
s2	$(3,4,0,\perp,0)$	$(4,4,0,0,0) = s6$	$(3,5,1,\perp,\perp) = s4$
s3	$(3,5,0,\perp,\perp)$	$(4,5,0,0,\perp) = s7$	$(3,3,0,\perp,\perp) = s1$
s4	$(3,5,1,\perp,\perp)$	$(4,5,1,1,\perp) = s8$	*no move (s4)*
s5	$(4,3,0,0,\perp)$	$(5,3,1,\perp,\perp) = s10$	$(4,4,0,0,0) = s6$
s6	$(4,4,0,0,0)$	$(5,4,1,\perp,0) = s11$	$(4,5,1,0,\perp) = s9$
s7	$(4,5,0,0,\perp)$	$(5,5,1,\perp,\perp) = s13$	$(4,3,0,0,\perp) = s5$
s8	$(4,5,1,1,\perp)$	$(5,5,2,\perp,\perp) = s14$	*no move (s8)*
s9	$(4,5,1,0,\perp)$	$(3,5,1,\perp,\perp) = s4$	*no move (s9)*
s10	$(5,3,1,\perp,\perp)$	*no move (s10)*	$(5,4,1,\perp,1) = s12$
s11	$(5,4,1,\perp,0)$	*no move (s11)*	$(5,3,1,\perp,\perp) = s10$
s12	$(5,4,1,\perp,1)$	*no move (s12)*	$(5,5,2,\perp,\perp) = s14$
s13	$(5,5,1,\perp,\perp)$	*no move (s13)*	*no move (s13)*
s14	$(5,5,2,\perp,\perp)$	$(3,5,0,\perp,\perp) = s3$	*no move (s14)*

For (b) state $(5,5,1,\perp,\perp)$ *is a deadlock.*

For (c) remove the transition from $(3,5,0,\perp,\perp) = s3$ *to* $(4,5,0,0,\perp) = s7$*. From* $s1$ *there are multiple ways to get to* $s4$ *or* $s10$ *corresponding to the first process to have reached the barrier. But then, from* $s4$ *to* $s14$ *and from* $s10$ *to* $s14$ *there is one way each corresponding to the other process to have reached the barrier. Finally, the remaining path from* $s14$ *to* $s3$ *and* $s1$ *reinitializes the barrier.*

It is also possible to remove the transition from $s7$ *to* $s13$ *but then the barrier does not necessarily return to the same initial state.*