Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

18 August 2022

Exam supervisors: N. Piterman (piterman@chalmers.se, 073 856 49 10) and G. Schneider (gersch@chalmers.se, 072 974 4964)

(Exam set by G. Schneider and N. Piterman, based on the courses given in September-October 2021 and January-February 2022)

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
28–41	3
42–55	4
56–70	5

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
40–59	3
60–79	4
80–100	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 receive no 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; four questions, numbered Q1 through Q4. 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 (18p). This question is concerned with implementing a semaphore-like functionality in Erlang.

(Part a). (6p) Implement a semaphore server. You need to support four functions: `start`, `value`, `signal`, and `wait`. The interface is given below. Notice that the `wait` function could block until the semaphore is increased (using a `signal` call). For convenience, the code for `gserver` is given in Appendix A. You may either call `gserver` code or create your own server functionality.

```
-module(semaphore).
-compile(export_all).

start(InitialValue) ->
  ...

signal(Server) ->
  ...

wait(Server) ->
  ...

value(Server) ->
  ...
```

(Part b). (6p) Is your semaphore strongly fair? If so, explain why. If not, explain how you would change it so that it would be (no need to change the implementation).

(Part c). (6p) What is the problem with the `gserver:stop` functionality? Explain how you would change `gserver` and your own implementation to avoid this problem while maintaining correct semaphore functionality (even after termination).

Q2 (18p). In lecture 8 we have seen a *fair* solution to the Readers-Writers problem, that is, a solution that avoids writers' starvation (a first naive solution giving priority to readers over writers has also been given). The fair solution is based on an implementation of the state/transition diagram shown in Fig. 1.

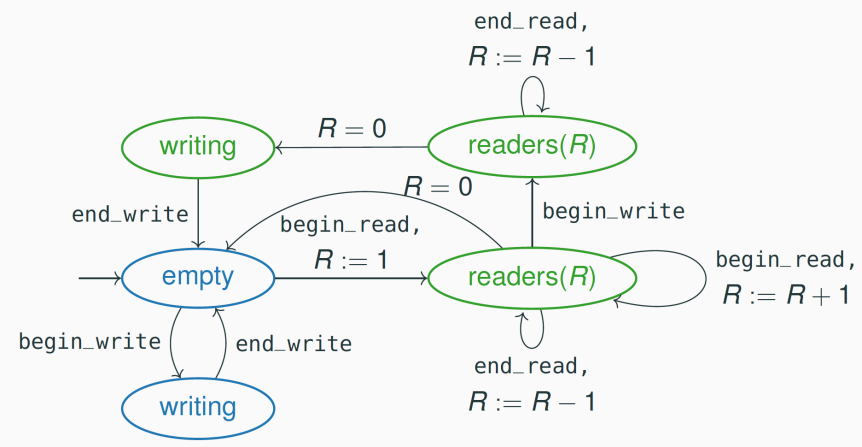

Figure 1: Q2: Diagram showing a fair solution to the readers-writers problem

```
% board with no readers and no writers
empty_board() ->
   receive
      % serve read request
      {begin_read, From, Ref} ->
           From ! {ok_to_read, Ref}, % notify reader
           readers_board(1); % board has one reader
      % serve write request synchronously
      {begin_write, From, Ref} ->
           From ! {ok_to_write, Ref}, % notify writer
           receive % wait for writer to finish
                {end_write, _From, _Ref} ->
                    empty_board() % board is empty again
           end
   end.
```

Figure 2: Q2: The server function `empty_board`

Figures 2 and 3 show the Erlang code implementing the diagram (as shown in the lectures).

(Part a). (6p) The proposed solution assumes the board is empty at the beginning so threads can only *begin* a write or a read. You, as

```
% board with no readers (and no writers)
readers_board(0) -> empty_board();

% board with 'Readers' active readers (and no writers)
readers_board(Readers) ->
   receive
      % serve write request
      {begin_write, From, Ref} ->
          % wait until all 'Readers' have finished
          [receive {end_read, _From, _Ref} -> end_read end
               || _ <- lists:seq(1, Readers)],
          From ! {ok_to_write, Ref}, % notify writer
          receive % wait for writer to finish
              {end_write, _From, _Ref} -> empty_board()
          end;
      % serve read request
      {begin_read, From, Ref} ->
          From ! {ok_to_read, Ref}, % notify reader
          readers_board(Readers+1); % board has one more reader
      % serve end read
      {end_read, _From, _Ref} ->
          readers_board(Readers-1) % board has one less reader
   end.
```

Figure 3: Q2: The server function `readers_board`

a programmer, have been requested to modify/extend the diagram to capture the more general situation when the board is not empty at the beginning. Your task is to extend the diagram so at the beginning the board is "cleaned" to empty the board. Then the board should behave as specified in the original diagram.

In order to do so, you may consider that initially there are N_R readers and M_W writers in the board, and you can assume there are two messages `begin_clean` and `end_clean` to start and end the cleaning, respectively.

NOTE: The two messages `begin_clean` and `end_clean` are included as we are assuming the whole process is going to be embedded into a bigger system and some other process will call the cleaning process.

The `clean` procedure should not allow for any reader or writer to start reading or writing, respectively.

(Part b). (6p) Write a program `clean_board` in Erlang that implements the diagram produced in *Part a*, so it is integrated in the whole

solution.

NOTE: You only need to write the code for the extended part of the diagram, and "connect" it with the existing programs shown in Figures 2 and 3.

(Part c). (6p) When you showed the solution to the problem described in *Part a* you were told that you have misunderstood the instructions: the idea was not to "clean" the board just at the beginning of the process, but at *any* time during the operation of the system.

For that, you should consider a third thread that at any moment can execute a `begin_clean` operation that blocks any new `begin_read` and `begin_write`, waits till all the active readers and writers finish their task, and restart the whole procedure again when the board is empty.

Your task is to modify / extend the diagram in Fig. 1 to capture the above specification.

Q3 (18p). A solution to concurrently access a list is to use a coarse-locking method, locking all elements of the list. In lecture 10 we have seen that though this works (it guarantees exclusive access and avoids inconsistencies), it is not satisfactory since the access is essentially sequential.

An alternative solution is to use a *fine-grained locking* approach. The questions below are concerned with statements and situations concerning fine-grained locking (with no validation) on linked lists being accessed by 2 threads t_0 and t_1.

(Part a) (4p) Answer whether the statements below concerning fine-grained locking are true or false. Justify your answer in each case (an answer without justification would not be granted full points).

1) (2p) In order to guarantee that the concurrent access works well (i.e., there are no inconsistencies), it is enough that both threads lock only their *pred* pointed node when executing the `find` method.
2) (2p) Fine-grained locking does not work well if there are too many threads executing the validation process to ensure no two threads are accessing the same node at the same time.

(Part b) (8p) This question concerns the design of a protocol to be used on parallel linked lists using a fine-grained locking approach (without validation).

Let's assume you have the list shown in Fig. 7 and two threads t_0 and t_1 accessing it:

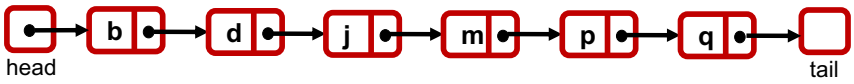

Figure 4: Q3: A list accessed concurrently by 2 threads t_0 and t_1

Let's assume the initial configuration is as follows:

- thread t_0 has *pred* pointing to the head and holds a lock on the node, and *curr* points to the first node (b) also holding a lock;
- thread t_1 has *pred* pointing to node d (holding a lock on the node), and *curr* points to node j also holding a lock.

According to this protocol, the threads would proceed as follows:

1 thread t_1 releases both locks (on d and j), and keeps pointing to d and j;
2 thread t_0 moves both *pred* and *curr* one position forward (so *pred* points to b and *curr* points to d), keeping a lock on both nodes;
3 thread t_0 moves again so *pred* points to d and *curr* points to j) (still keeping a lock on both nodes).

So, the final situation is that both t_0 and t_1 are pointing to the same nodes (d and j), but only t_0 holds a lock over the nodes.

Is this protocol "safe" (nothing can go wrong)?

If you think nothing can go wrong, explain why.

If you think there might be an issue, continue the scenario above with the next steps that would lead to a bad situation. Explain why this happens and provide a fix (what is to be done in order to avoid the bad situation).

(Part c) (6p) A programmer has been given the task to implement the `find` method for a fine-grained locking solution (without validation) to access a parallel linked list. The programmer took an existing solution and slightly modified it, producing the following code:

```
1  protected Node<T>, Node<T> find(Node<T> start, int key) {
2    Node<T> pred, curr;
3    pred = start; curr = start.next();
4    pred.lock();
5    while (curr.key < key) {
6        curr.lock();
7        pred.unlock();
8        pred = curr;
9        curr = curr.next();
10       curr.unlock();
11    }
12 return (pred, curr);
13 }
```

The supervisor is not happy at all with the solution of the programmer claiming that the code is not correct.

Explain why the solution is wrong and provide a correct version so the `find` method can be used as expected in a fine-grained locking algorithm (with no validation).

Q4 (16p). The program below guarantees mutual-exclusion between two processes. It uses a compare-and-swap operation.

boolean `turn`= false; **boolean** `flaga`= false; **boolean** `flagb`= false;			
	`p`		`q`
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
p_2:	`flaga`= true;	q_2:	`flagb`= true;
p_3:	**while**(!`turn`.CAS(false,true) && `flagb`) { };	q_3:	**while**(!`turn`.CAS(false,true) && `flaga`) { };
p_4	*//CS (critical section)*	q_4	*//CS (critical section)*
p_5:	`turn`= `flaga`= false;	q_5:	`turn`=`flagb`= false;
	}		}

For simplicity, we ignore the locations p_1 and p_4 and similarly q_1 and q_4. Process p moves directly from p_3 to p_5 and from p_5 to p_2 and similarly for q. We treat p_5 and q_5 as the critical section.

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 $\{2, 3, 5\}$, and `flaga`, `flagb`, and `turn` range over *true* and $false$. Only 8 states are reachable.

Here is a partial state transition table for the program above. As mentioned, only 8 states are reachable from the initial state (p_2, q_2, f, f, f).

state		new state if p moves	new state if q moves
s1	$(2, 2, f, f, f)$	$(3, 2, t, f, f)$ = s3	
s2	$(2, 3, f, t, f)$		
s3	$(3, 2, t, f, f)$	$(5, 2, t, f, t)$ = s6	
s4	$(3, 3, t, t, f)$		
s5	$(2, 5, f, t, t)$		
s6	$(5, 2, t, f, t)$	$(2, 2, f, f, f)$ = s1	
s7	$(5, 3, t, t, t)$		
s8	$(3, 5, t, t, t)$		

(Part a). (4p) Fill in the blank entries in the table.

(Part b). (2p) Does the protocol maintain mutual exclusion?

(Part c). (3p) Consider the condition !`turn`.$CAS(false, true)$&&`flagb` guarding the loop for process p. Argue from the table or otherwise that the second conjunct does not play a role in the evaluation of this condition.

(Part d). (4p) Based on part (c), suggest what parts of the program to remove so that the protocol becomes simpler but still maintains mutual exclusion (write the new protocol explaining what you removed).

(Pard e). (3p) For how many processes does your new protocol work?

A Q1: Erlang code for gserver

Here is the code for gserver, for reference. You can either use gserver or repeat some of this code in your own server implementation. Notice that we are *not* using the robust version as we do not want the processes making requests to timeout.

```
-module(gserver). % generic server
-export([start/2,request/2,stop/1]).

% start a server, return server's pid
start(InitialState, Handler) ->
  spawn(fun () -> loop(InitialState, Handler) end).

% event loop
loop(State, Handler) ->
  receive
      % a request from 'From' with data 'Request'
    {request, From, Ref, Request} ->
        % run handler on request
      case Handler(State, Request) of
          % get handler's output
        {reply, NewState, Result} ->
            % the requester gets the result
          From ! {response, Ref, Result},
            % the server continues with the new state
          loop(NewState, Handler)
      end;
    {stop, _From, _Ref} -> ok
  end.

% issue a request to 'Server'; return answer
request(Server, Request) ->
  Ref = make_ref(),  % unique reference number
    % send request to server
  Server ! {request, self(), Ref, Request},
    % wait for response, and return it
  receive {response, Ref, Result} -> Result end.

stop(Server) ->
  Server ! {stop, self(), 0}, % Ref is not needed
  ok.
```

Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

18 August 2022

Exam supervisors: N. Piterman (piterman@chalmers.se, 073 856 49 10) and G. Schneider (gersch@chalmers.se, 072 974 4964)

(Exam set by G. Schneider and N. Piterman, based on the courses given in September-October 2021 and January-February 2022)

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
28–41	3
42–55	4
56–70	5

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
40–59	3
60–79	4
80–100	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 receive no 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; four questions, numbered Q1 through Q4. 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 (18p). This question is concerned with implementing a semaphore-like functionality in Erlang.

(Part a). (6p) Implement a semaphore server. You need to support four functions: `start`, `value`, `signal`, and `wait`. The interface is given below. Notice that the `wait` function could block until the semaphore is increased (using a `signal` call). For convenience, the code for `gserver` is given in Appendix A. You may either call `gserver` code or create your own server functionality.

```
-module(semaphore).
-compile(export_all).

start(InitialValue) ->
  ...

signal(Server) ->
  ...

wait(Server) ->
  ...

value(Server) ->
  ...
```

ANSWER: Here is a sample implementation.

```
-module(semaphore).
-compile(export_all).

% The state of the Handler
%
% { value, pids_refs_list }

start(InitialValue) ->
  gserver:start({InitialValue,[]}, fun semaphore_loop/2).

signal(Server) ->
  gserver:request(Server, signal).

% In order for the wait to block, it awaits another message
% with another reference.
% gserver always answers immediately, but the answer to the
% wait may be delayed a long time.
```

```
wait(Server) ->
  Ref = make_ref(),
  gserver:request(Server, { wait, self(), Ref }),
  receive { response, Ref } -> ok end.

value(Server) ->
  gserver:request(Server, value).

semaphore_loop({Val, List}, value) ->
  { reply, {Val, List}, Val };

% On signal w no waiting processes increase the value of the semaphore
semaphore_loop({Val, []}, signal) ->
  { reply, {Val+1,[]}, ok };

% On signal w processes waiting serve one waiting process
semaphore_loop({Val, [{From, Ref} | List]}, signal) ->
  From ! { response, Ref },
  { reply, {Val, List }, ok };

% On wait when value is 0 add the process to the list
semaphore_loop({0, List}, { wait, From, Ref }) ->
  { reply, {0, [ {From, Ref} | List ]}, ok };

% On wait when value is not 0 award the incoming request
semaphore_loop({Val, List}, { wait, From, Ref}) ->
  From ! { response, Ref },
  { reply, {Val-1, List}, ok}.
```

(Part b). (6p) Is your semaphore strongly fair? If so, explain why. If not, explain how you would change it so that it would be (no need to change the implementation).

ANSWER: No. My semaphore is not strongly fair. When the value of the semaphore is 0 and new requests come they are put at the head of the list. In order to make the semaphore strongly fair new requests would have to be added to the end of the list and when there is a new wait request, the value is not-zero, and the list is not empty, handle first those that are waiting and not the new comer (though it is not possible to reach such a state).

(Part c). (6p) What is the problem with the `gserver:stop` functionality? Explain how you would change `gserver` and your own implementation to avoid this problem while maintaining correct semaphore functionality (even after termination).

ANSWER: It would leave the process in the waiting list of the semaphore deadlocked forever. In order to resolve his, the `stop` *functionality would have to first send a stop request to the loop and only then stop the server. The loop would handle the stop request by sending abort messages to all the waiting processes. The* `wait` *function would have to be modified to receive either the extra response it has now or a stop message that would make the function return in a different way or raise an exception.*

Q2 (18p). In lecture 8 we have seen a *fair* solution to the Readers-Writers problem, that is, a solution that avoids writers' starvation (a first naive solution giving priority to readers over writers has also been given). The fair solution is based on an implementation of the state/transition diagram shown in Fig. 1.

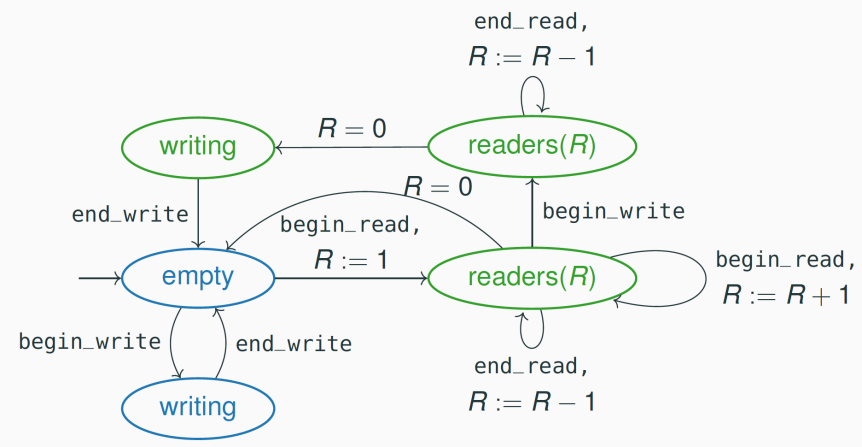

Figure 1: Q2: Diagram showing a fair solution to the readers-writers problem

```
% board with no readers and no writers
empty_board() ->
   receive
      % serve read request
      {begin_read, From, Ref} ->
           From ! {ok_to_read, Ref}, % notify reader
           readers_board(1); % board has one reader
      % serve write request synchronously
      {begin_write, From, Ref} ->
           From ! {ok_to_write, Ref}, % notify writer
           receive % wait for writer to finish
                {end_write, _From, _Ref} ->
                    empty_board() % board is empty again
           end
   end.
```

Figure 2: Q2: The server function `empty_board`

Figures 2 and 3 show the Erlang code implementing the diagram (as shown in the lectures).

(Part a). (6p) The proposed solution assumes the board is empty at the beginning so threads can only *begin* a write or a read. You, as

```
% board with no readers (and no writers)
readers_board(0) -> empty_board();

% board with 'Readers' active readers (and no writers)
readers_board(Readers) ->
   receive
      % serve write request
      {begin_write, From, Ref} ->
          % wait until all 'Readers' have finished
          [receive {end_read, _From, _Ref} -> end_read end
               || _ <- lists:seq(1, Readers)],
          From ! {ok_to_write, Ref}, % notify writer
          receive % wait for writer to finish
              {end_write, _From, _Ref} -> empty_board()
          end;
      % serve read request
      {begin_read, From, Ref} ->
          From ! {ok_to_read, Ref}, % notify reader
          readers_board(Readers+1); % board has one more reader
      % serve end read
      {end_read, _From, _Ref} ->
          readers_board(Readers-1) % board has one less reader
   end.
```

Figure 3: Q2: The server function `readers_board`

a programmer, have been requested to modify/extend the diagram to capture the more general situation when the board is not empty at the beginning. Your task is to extend the diagram so at the beginning the board is "cleaned" to empty the board. Then the board should behave as specified in the original diagram.

In order to do so, you may consider that initially there are N_R readers and M_W writers in the board, and you can assume there are two messages `begin_clean` and `end_clean` to start and end the cleaning, respectively.

NOTE: The two messages `begin_clean` and `end_clean` are included as we are assuming the whole process is going to be embedded into a bigger system and some other process will call the cleaning process.

The `clean` procedure should not allow for any reader or writer to start reading or writing, respectively.

ANSWER: See Fig. 4.

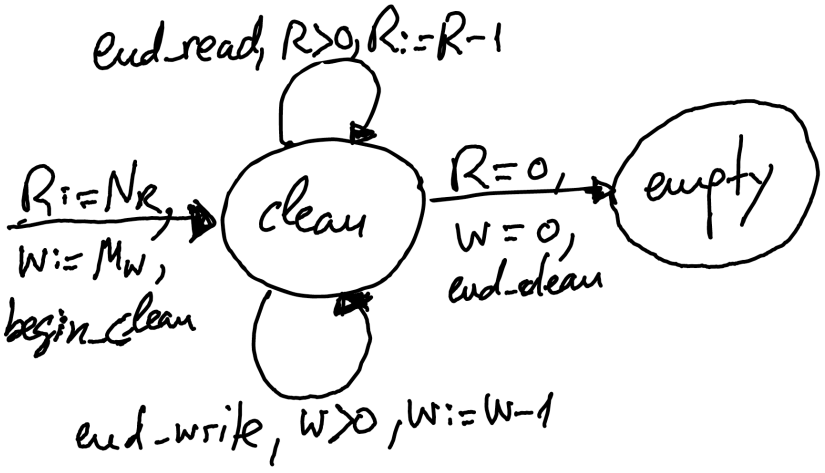

Figure 4: Q2: Solution to Part a.

(Part b). (6p) Write a program `clean_board` in Erlang that implements the diagram produced in *Part a*, so it is integrated in the whole solution.

NOTE: You only need to write the code for the extended part of the diagram, and "connect" it with the existing programs shown in Figures 2 and 3.

ANSWER: See Fig. 5.

(Part c). (6p) When you showed the solution to the problem described in *Part a* you were told that you have misunderstood the instructions: the idea was not to "clean" the board just at the beginning of the process, but at *any* time during the operation of the system.

For that, you should consider a third thread that at any moment can execute a `begin_clean` operation that blocks any new `begin_read` and `begin_write`, waits till all the active readers and writers finish their task, and restart the whole procedure again when the board is empty.

Your task is to modify / extend the diagram in Fig. 1 to capture the above specification.

ANSWER: See Fig. 6.

```
% board with some number NR of readers and MW of writers
init_board() ->
    receive
        {begin_clean, From, Ref, {NR, MW}} ->
                            clean_board(From, Ref, NR, NW)
    end.

clean_board(From, Ref, 0, 0) ->
    From ! {end_clean, Ref},
    empty_board();

clean_board(From, Ref, NR, MW) ->
    receive
        {end_read, _From, _Ref} ->
                        clean_board(From, Ref, NR - 1, MW);
        {end_write, _From, _Ref} ->
                         clean_board(From, Ref, NR, MW - 1)
    end.
```

Figure 5: Q2: Solution to Part b.

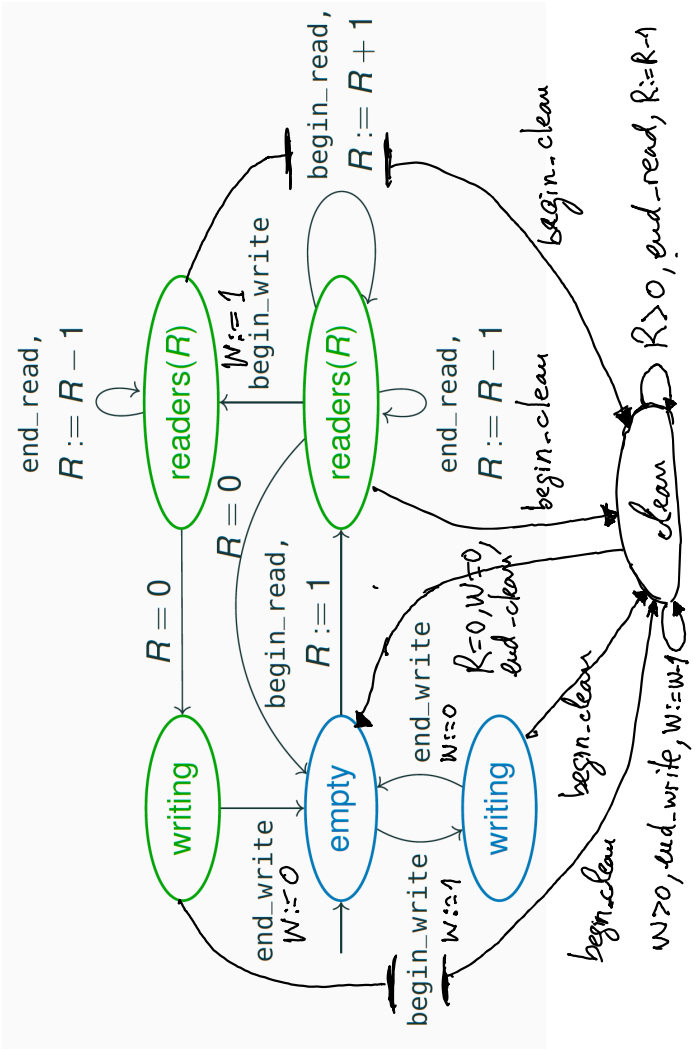

Figure 6: Q2: Solution to Part c.

Q3 (18p). A solution to concurrently access a list is to use a coarse-locking method, locking all elements of the list. In lecture 10 we have seen that though this works (it guarantees exclusive access and avoids inconsistencies), it is not satisfactory since the access is essentially sequential.

An alternative solution is to use a *fine-grained locking* approach. The questions below are concerned with statements and situations concerning fine-grained locking (with no validation) on linked lists being accessed by 2 threads t_0 and t_1.

(Part a) (4p) Answer whether the statements below concerning fine-grained locking are true or false. Justify your answer in each case (an answer without justification would not be granted full points).

1) (2p) In order to guarantee that the concurrent access works well (i.e., there are no inconsistencies), it is enough that both threads lock only their *pred* pointed node when executing the `find` method.
2) (2p) Fine-grained locking does not work well if there are too many threads executing the validation process to ensure no two threads are accessing the same node at the same time.

ANSWER:

1) False: the `find` *method is required to lock both the* pred *and* curr *nodes for all accessing threads, otherwise an inconsistency may arise when one of the threads tries to remove its* curr *node while the other thread has already updated its* curr *pointer, pointing then to a non-existing node. (See Lecture 10, slide 33.)*

2) False: there is no validation process in fine-grained locking as presented in the lectures (the `find` *function in fine-grained locking locks both* pred *and* curr *and does not need validation. Validation is needed when* `find` *does not use locking (as in the* optimistic locking *case).*

(Part b) (8p) This question concerns the design of a protocol to be used on parallel linked lists using a fine-grained locking approach (without validation).

Let's assume you have the list shown in Fig. 7 and two threads t_0 and t_1 accessing it:

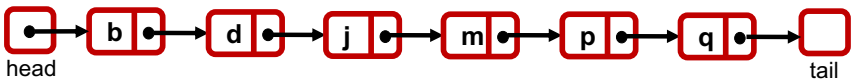

Figure 7: Q3: A list accessed concurrently by 2 threads t_0 and t_1

Let's assume the initial configuration is as follows:

- thread t_0 has *pred* pointing to the head and holds a lock on the node, and *curr* points to the first node (b) also holding a lock;
- thread t_1 has *pred* pointing to node d (holding a lock on the node), and *curr* points to node j also holding a lock.

According to this protocol, the threads would proceed as follows:

1 thread t_1 releases both locks (on d and j), and keeps pointing to d and j;
2 thread t_0 moves both *pred* and *curr* one position forward (so *pred* points to b and *curr* points to d), keeping a lock on both nodes;
3 thread t_0 moves again so *pred* points to d and *curr* points to j) (still keeping a lock on both nodes).

So, the final situation is that both t_0 and t_1 are pointing to the same nodes (d and j), but only t_0 holds a lock over the nodes.

Is this protocol "safe" (nothing can go wrong)?

If you think nothing can go wrong, explain why.

If you think there might be an issue, continue the scenario above with the next steps that would lead to a bad situation. Explain why this happens and provide a fix (what is to be done in order to avoid the bad situation).

ANSWER: The situation is not safe since thread t_1 might end up pointing to a non-existing node. This could happen if the following steps are executed:

1 thread t_0 removes node j and updates the pointer from d to m;
2 thread t_0 terminates and releases both locks;
3 thread t_1 updates pred *and* curr *so* pred *points to j and* curr *points to m) (holding a lock on both nodes).*

Clearly this is a problem since node j has been deleted by thread t_0 but thread t_1 believes it is still in the list.

The problem is that thread t_1 released the locks. The solution is to force that both threads always keep a lock over the nodes pointed by pred *and* curr.

(Part c) (6p) A programmer has been given the task to implement the `find` method for a fine-grained locking solution (without validation) to access a parallel linked list. The programmer took an existing solution and slightly modified it, producing the following code:

```
1 protected Node<T>, Node<T> find(Node<T> start, int key) {
2   Node<T> pred, curr;
3   pred = start; curr = start.next();
4   pred.lock();
5   while (curr.key < key) {
```

```
 6       curr.lock();
 7       pred.unlock();
 8       pred = curr;
 9       curr = curr.next();
10       curr.unlock();
11    }
12 return (pred, curr);
13 }
```

The supervisor is not happy at all with the solution of the programmer claiming that the code is not correct.

Explain why the solution is wrong and provide a correct version so the `find` method can be used as expected in a fine-grained locking algorithm (with no validation).

ANSWER: The code is incorrect since both pred *and* curr *should be locked before the loop starts otherwise an inconsistency may arise (e.g., another thread might remove the node pointed by* curr *before the condition of the while is checked). (We are in a setting without validation.)*

Besides, the last sentence of the loop (`curr.unlock();`) is not correct since the curr *node is not holding a lock (it now points to a new node and the lock is not acquired yet).*

The correct solution is as shown in lecture 10, slide 34 (code reproduced below)

```
protected Node<T>, Node<T> find(Node<T> start, int key) {
  Node<T> pred, curr; // predecessor and current node in iteration
  pred = start; curr = start.next(); // from start node
  pred.lock(); curr.lock(); // lock pred and curr nodes
  while (curr.key < key) {
      pred.unlock(); // unlock pred node
      pred = curr;
      curr = curr.next(); // move to next node
      curr.lock(); // lock next node
   } // until curr.key >= key
return (pred, curr); // return position
}
```

Q4 (16p). The program below guarantees mutual-exclusion between two processes. It uses a compare-and-swap operation.

boolean `turn`= false; **boolean** `flaga`= false; **boolean** `flagb`= false;			
	`p`		`q`
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
p_2:	`flaga`= true;	q_2:	`flagb`= true;
p_3:	**while**(!`turn`.CAS(false,true) && `flagb`) { };	q_3:	**while**(!`turn`.CAS(false,true) && `flaga`) { };
p_4	*//CS (critical section)*	q_4	*//CS (critical section)*
p_5:	`turn`= `flaga`= false;	q_5:	`turn`=`flagb`= false;
	}		}

For simplicity, we ignore the locations p_1 and p_4 and similarly q_1 and q_4. Process p moves directly from p_3 to p_5 and from p_5 to p_2 and similarly for q. We treat p_5 and q_5 as the critical section.

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 $\{2, 3, 5\}$, and `flaga`, `flagb`, and `turn` range over *true* and $false$. Only 8 states are reachable.

Here is a partial state transition table for the program above. As mentioned, only 8 states are reachable from the initial state (p_2, q_2, f, f, f).

state		new state if p moves	new state if q moves
s1	$(2, 2, f, f, f)$	$(3, 2, t, f, f)$ = s3	
s2	$(2, 3, f, t, f)$		
s3	$(3, 2, t, f, f)$	$(5, 2, t, f, t)$ = s6	
s4	$(3, 3, t, t, f)$		
s5	$(2, 5, f, t, t)$		
s6	$(5, 2, t, f, t)$	$(2, 2, f, f, f)$ = s1	
s7	$(5, 3, t, t, t)$		
s8	$(3, 5, t, t, t)$		

(Part a). (4p) Fill in the blank entries in the table.

(Part b). (2p) Does the protocol maintain mutual exclusion?

(Part c). (3p) Consider the condition !`turn`.$CAS(false, true)$&&`flagb` guarding the loop for process p. Argue from the table or otherwise that the second conjunct does not play a role in the evaluation of this condition.

(Part d). (4p) Based on part (c), suggest what parts of the program to remove so that the protocol becomes simpler but still maintains mutual exclusion (write the new protocol explaining what you removed).

(Part e). (3p) For how many processes does your new protocol work?

ANSWER:

Part (a):

state		*new state if p moves*	*new state if q moves*
s1	$(2, 2, f, f, f)$	$(3, 2, t, f, f) = s3$	$(2, 3, f, t, f) = s2$
s2	$(2, 3, f, t, f)$	$(3, 3, t, t, f) = s4$	$(2, 5, f, t, t) = s5$
s3	$(3, 2, t, f, f)$	$(5, 2, t, f, t) = s6$	$(3, 3, t, t, f) = s4$
s4	$(3, 3, t, t, f)$	$(5, 3, t, t, t) = s7$	$(3, 5, t, t, t) = s8$
s5	$(2, 5, f, t, t)$	$(3, 5, t, t, t) = s8$	$(2, 2, f, f, f) = s1$
s6	$(5, 2, t, f, t)$	$(2, 2, f, f, f) = s1$	$(5, 3, t, t, t) = s7$
s7	$(5, 3, t, t, t)$	$(2, 3, f, t, f) = s2$	—
s8	$(3, 5, t, t, t)$	—	$(3, 2, t, f, f) = s3$

For (b): Yes. At most one of the processes is in location 5 at every reachable state.

For (c): In all reachable states, whenever process p is in p_3 evaluating this condition and process q is in location 2 (i.e., `!flagb`*) it is always the case that the CAS evaluates to true.*

For (d): the following symmetric protocol works:

boolean `turn`= *false;*	
	while*(true) {*
p_1:	*//NCS (non-critical section)*
p_2:	**while***(*`!turn`$.CAS(false, true)$*) {};*
p_3:	*//CS (critical section)*
p_4:	`turn`*= false;}*

For (e): it works for every number of processes.

A Q1: Erlang code for gserver

Here is the code for gserver, for reference. You can either use gserver or repeat some of this code in your own server implementation. Notice that we are *not* using the robust version as we do not want the processes making requests to timeout.

```
-module(gserver). % generic server
-export([start/2,request/2,stop/1]).

% start a server, return server's pid
start(InitialState, Handler) ->
  spawn(fun () -> loop(InitialState, Handler) end).

% event loop
loop(State, Handler) ->
  receive
      % a request from 'From' with data 'Request'
    {request, From, Ref, Request} ->
        % run handler on request
      case Handler(State, Request) of
          % get handler's output
        {reply, NewState, Result} ->
            % the requester gets the result
          From ! {response, Ref, Result},
            % the server continues with the new state
          loop(NewState, Handler)
      end;
    {stop, _From, _Ref} -> ok
  end.

% issue a request to 'Server'; return answer
request(Server, Request) ->
  Ref = make_ref(),  % unique reference number
    % send request to server
  Server ! {request, self(), Ref, Request},
    % wait for response, and return it
  receive {response, Ref, Result} -> Result end.

stop(Server) ->
  Server ! {stop, self(), 0}, % Ref is not needed
  ok.
```