Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT392

13 March 2023

Exam supervisor: G. Schneider (gersch@chalmers.se, 072 974 49 64)

(Exam set by G. Schneider, based on the course given Jan-Mar 2023)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes (single or double-sided); English dictionary (no smart phones allowed).

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; 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 (18p). This question is concerned with Peterson's algorithm (as seen in Lecture 3).

(Part a). (8p)

Fig. 1 shows Peterson's algorithm for 2 threads.

```
boolean[] enter = {false, false};   int yield = 0 || 1;
```

thread t_0

```
1  while (true) {
2   // entry protocol
3   enter[0] = true;
4   yield = 0;
5   await (!enter[1]        ||
          yield != 0);
6   critical section { ... }
7   // exit protocol
8   enter[0] = false;
9  }
```

thread t_1

```
10 while (true) {
11  // entry protocol
12  enter[1] = true;
13  yield = 1;
14  await (!enter[0]        ||
          yield != 1);
15  critical section { ... }
16  // exit protocol
17  enter[1] = false;
18 }
```

Figure 1: Q1:Peterson's Algorithm for 2 threads.

Would the algorithm be correct if we make the simultaneous replacements below?

- Line 3 is replaced by `enter[1] = true`
- Line 12 is replaced by `enter[0] = true`
- Line 8 is replaced by `enter[1] = false`
- Line 17 is replaced by `enter[0] = false`

Justify your answer. If the answer is NO, explain what would be the new behaviour): give a concrete execution trace showing why this not the case. If the answer is YES, explain it.

ANSWER: Yes, it still guarantees mutual exclusion, deadlock freedom and starvation freedom. The only change is that instead of saying "I want to enter into the critical section" (by setting `enter[i]= true` *for* `i` *equal 0 or 1 depending on the thread), it says "You want to enter into the critical section" (by inverting the value of* `i`*).*

To see that it guarantees deadlock freedom see that the await condition means that at any moment one of the threads (and only one) will be able to enter the critical section since `yield` *can be 0 or 1 but not both at the same time (and it will be 0 or 1 in all iterations). Note that there is mutual exclusion since there is no way to have* `yield=0` *and* `enter[0]= false` *(similarly, not possible that* `yield=1` *and* `enter[1]= false`*). So, at any moment only one of the threads would be able to go in.*

(Part b). (10p)

Does the generalised Peterson's algorithm (for n threads, shown in Fig. 2) when instantiated with $n = 2$ behaves the same as Peterson's algorithm for 2 threads? If so, set $n = 2$ in the generalised algorithm and explain how you get the original algorithm for 2 threads. You need to justify your answer (do not just give the instantiation of the algorithm).

```
int[] enter = new int[n]; // n elements, initially all 0s
 int[] yield = new int[n]; // use n - 1 elements 1..n-1
--------------------------------------------------------
                      thread x

while (true) {
 // entry protocol
 for (int i = 1; i < n; i++) {
   enter[x] = i;    // want to enter level i
   yield[i] = x;    // but yield first
   await (∀ t != x: enter[t] < i
          || yield[i] != x);
 }
 critical section { ... }
 // exit protocol
 enter[x] = 0;  // go back to level 0
```

Figure 2: Q1:Generalised Peterson's Algorithm (for n threads).

ANSWER: Yes, it is the same. By setting $n = 2$, note that the loop becomes redundant since i can only take the value 1, x can be only 0 or 1, and the "levels" now can only be 0 or 1, level 1 ($i = 1$) is `true` *and level 0 is* `false` *The* `await` *statement becomes (for thread 0)* `await (enter[1] < 1 || yield[1] != 0)` *which is equivalent to* `await (enter[1] = false || yield[1] != 0)` *(this is because t in the $\forall t! = x$ can only take one value: 0 if $x = 1$ and 1 if $x = 0$).*

Q2 (18p). This question is concerned with a car production line in which some robots produce brake pedals and put them on a circular belt. These pedals are taken by other robots along the production line and pass it on to be assembled in the car. Brake producer robots are called *makers* and the ones taking the brakes from the belt are called *users*.

The belt has a capacity `Max` and it is not possible to add more pedals if the belt is full. (`Max` is strictly positive, i.e., `Max`>0) Similarly, no user can take a pedal from the belt if this is empty.

You can assume that there are m makers and n users ($n, m > 0$).

A programmer who did not took the Principles of Concurrent Programming course was in charge of writing a program to handle the above with the explicit instruction that the program should be deadlock-free, starvation-free and it should work for an arbitrary number of makers and users. The programmer wrote the code in Fig. 3 defining a class `Belt` and two methods for putting pedals in the belt (`put`) and taking pedals from the belt (`take`).

(`belt` is defined as: `Belt`⟨`pedal`⟩ `belt`)

```
public class Belt<T> {
  Semaphore nPedals = new Semaphore(0);
  Semaphore nFree = new Semaphore(Max);

  Lock lock = new Lock();
  Collection store = ...; // implem. of a collection of pedals
```

```
public void put(T pedal) {
  nFree.down();
  lock.lock();
  store.add(pedal);
  lock.unlock();
  nPedals.up();
}
```

```
public T take() {
  lock.lock();
  nPedals.down();
  T pedal = store.remove();
  nFree.up();
  lock.unlock();
  return pedal;
}
```

Figure 3: Q2: Car production code for makers and users.

Besides the shown code, there is a main program that creates m threads for makers and n threads for users. Each maker keeps creating pedals and putting them into the belt by calling `belt.put(pedal)` where users keep taking them from the belt by calling `pedal = belt.take()` and then passing them to the rest of the production line.

In what follows there are a number of assertions and questions. Please answer each one of them and justify your answer. A partially correct answer will not be given full points.

1 Is the following statement true or false: The program is correct but will only work if you have exactly the same amount of *makers* and *users*. Justify your answer. **(2p)**
2 Is the program deadlock-free? If it is, explain why this is the case. If not, explain which parts of the code are wrong, correct it and give a concrete execution trace showing the deadlock. **(5p)**
3 Is the following statement true or false: "Lines 5 and 6 should not be swapped because it might produce a deadlock"? Justify your answer. **(3p)**
4 Is the following statement true or false: "Swapping lines 12 and 13 is OK as the program will not deadlock"? Justify your answer. **(3p)**
5 The manager of the programmer claims that the program allows makers to add more pedals into the belt than its capacity. Is that true? Justify. **(3p)**
6 Is the following statement true or false: A correct implementation of the problem can be done with only one semaphore and one lock. Justify your answer. **(2p)**

ANSWER:

1 *No. The program is not correct (it will deadlock), and the number of makers and users does not affect the correctness of the program.*
2 *No, the program may deadlock: in the take method you should swap lines 9 and 10. An execution trace showing this is the following:*
1) A user start executing the `take` *method and acquires the lock (line 9) and then keeps waiting in line 10 (as the counter of* `nPedals` *is 0);*
2) A maker start executing the `put` *method and after executing line 2 tries to acquire the lock (line 3);*
3) Deadlock: The user holds the lock and is waiting for a maker to execute `nPedals.up()`*, while the maker is waiting for the user to release the lock.*
(This is an instance of the producer-consumer problem - See Lecture 4 slide 26 for a correct solution.)
3 *False: lines 5 and 6 may be swapped without affecting the outcome (see Lecture 4 slide 26).*
4 *True: lines 12 and 13 could be swapped (see Lecture 4 slide 26).*
5 *No: the semaphore* `nFree` *is used correctly so it not possible to have more than* `Max` *elements in the belt.*
6 *No, it's not correct: you require one lock and at least two semaphores (one to check the belt is not empty and one to check it is not full).*

Q3 (9p). A programmer has written the program below asserting that it guarantees mutual-exclusion between two processes. The solution is based on a *compare-and-set* (CAS) 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.

(Part a). (4p)

A full state of the program is of the form $(p_i, q_j,$ `flaga`, `flagb`, `turn`$)$, where i and j range over $\{2, 3, 5\}$, and `flaga`, `flagb`, and `turn` range over *true* and *false*.

Below you find a partial state transition table for the program above. Only 8 states are reachable from the initial state (p_2, q_2, f, f, f).

Your task is to fill in the blank entries in the table.

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

ANSWER:

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*

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

ANSWER: Yes. At most one of the processes is in location 5 at every reachable state.

(Part c). (3p) Consider the condition `!turn`.$CAS(false, true)$&&`flaga` guarding the loop for process q. Does the second conjunct (`flaga`) plays any role in the evaluation of the condition? Justify your answer.

ANSWER: No, `flaga` *doesn't play any role since in all reachable states, whenever process q is in* q_3 *evaluating this condition and process p is in location* p_2 *(i.e.,* `flaga` *is false) it is always the case that the CAS evaluates to true. (You can also see this in the table: the only states where q is in position 3 are s2, s4 and s7 and it is clear that the flag doesn't play any role in any of the cases.)*

Q4 (15p). The `reduce` function is a high-order function that can be defined in Erlang as follows:

```
reduce(_,A,[])      -> A;
reduce(F,A,[H|T])   -> F(H,reduce(F,A,T)).
```

We have seen the following parallel implementation of the reduce function in Lecture 9:

```
preduce(_, A, []) -> A;
preduce(F, A, [E]) -> F(A, E);
preduce(F, A, List) ->
  Mid = length(List) div 2,
  {L, R} = lists:split(Mid, List),
  Me = self(),  % L ++ R =:= Listn
  Lp = spawn(fun() -> % on left half
      Me ! {self(), preduce(F, A, L)} end),
  Rp = spawn(fun() -> % on right half
      Me ! {self(), preduce(F, A, R)} end),
  % combine results of left, right half
  F(receive {Lp, Lr} -> Lr end,  receive {Rp, Rr} -> Rr end).
```

(Part a). (4p)

Apply both `reduce` and `preduce` to the list `L = [2,4]` with initial value 0 for `A`, where `F` is the `plus/2` function defined as follows:

```
plus (X,Y) ->  X+Y.
```

That is, write down ("simulate") the execution of the following calls: `reduce(plus/2,0,[2,4])` and `preduce(plus/2,0,[2,4])`.

Answer:

`reduce(plus/2,0,[2,4])`
plus(4,reduce(plus/2,0,[4])
plus(2, plus(4,reduce(plus/2,0,[])))
plus(2, plus(4,0))
plus(2, 4)
6

`preduce(plus/2,0,[2,4])` will split the list into 2 and spawn two processes on each half, applying the same function to each half (LP will

be `preduce(plus/2,0,[2])` *and LR will be* `preduce(plus/2,0,[4]))`*. Since both lists have one element, the second base case will be applied, getting* `plus(0,2)` *and* `plus(0,4)`*.*
And now the results are combined, giving 6.

(Part b). (5p)

Does `preduce` always give the same answer as `reduce` when applied to the same function `F`, element `A` and list `L`? If so, explain how `preduce` is indeed a correct parallel implementation of `reduce`. If not, explain why is not the case and what is required for `preduce` to be a fully correct implementation of the original function. In case you answer that both functions may not compute the same thing, give a concrete example that shows the difference.

Answer: No. As seen in Lecture 9 slide 23, this parallel version only works provided the function F is associative and we have that for every element E of the list the following is true: F(E,A)= F(A,E) = E (A is the neutral element for operation/function F).

A concrete example that shows the above is for instance if you apply both `reduce` *and* `preduce` *to the* `plus` *function with 1 instead of 0 as second argument, as 1 doesn't satisfy the property that F(A,E) = E:* `reduce(plus/2,1,[2,4])` *will give 7 while* `preduce(plus/2,1,[2,4])` *will give 8.*

(Part c). (6p)

In `preduce`, why is `A` used in both sublists calls (left and right)? Wouldn't that mean that the function `F` will apply `A` on every element of the list and give a result different from the expected? Justify your answer (you can use Part a) and b) above as a way to explain your answer.)

Answer: Yes, the statement is in general true: `F` *will apply* `A` *on every element of the list and give a different result than the expected (see example in answer to Part b). That said, the implementation works if the following two conditions hold: F is associative and A is the neutral element for F. So,* `preduce(plus/2,0,[2,4])` *will give the right result (addition is associative and 0 is the neutral element for the addition), while* `preduce(plus/2,1,[2,4])` *would give a wrong result as seen in answer to Part b.*

Q5 (10p). A solution to concurrently access a list is to use a coarse-locking method, locking all elements of the list. In lecture 10 we saw that though this works it is not satisfactory since the access is essentially sequential, and we gave different alternative approaches. This question is about fine-grained locking.

Below it follows statements and situations concerning parallel linked lists implementing sets being accessed by 2 threads t_0 and t_1. As in our lecture, we assume that the linked list is sorted by key.

(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) 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. **(2p)**
2) If there are too many threads executing the validation process (to ensure no two threads are accessing the same node at the same time), the fine-grained locking does not work and inconsistencies may arise. **(2p)**

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) (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:

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

```
        curr.lock();
        pred.unlock();
        pred = curr;
        curr = curr.next();
        curr.unlock();
    }
return (pred, curr);
}
```

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:

```
protected Node<T>, Node<T> find(Node<T> start, int key) {
  Node<T> pred, curr; // predecessor and current node in iteration
  pred = start;
  pred.lock(); // lock pred node
  curr = start.next();
  curr.lock(); // lock curr node
  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
}
```

We also accept as correct the following solution (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();
  pred.lock(); // lock pred node
  curr.lock(); // lock curr node
  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
}
```