Chalmers | Göteborgs Universitet

# Concurrent Programming TDA384/DIT391

Tuesday, 16 March 2021

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

(Exam set by G. Schneider, based on the course given Jan-Mar 2021)

**Material permitted during the exam (hjälpmedel):**
As the exam is run remotely we cannot restrict your usage of material. Not allowed to use the Internet.

**Grading:** You can score a maximum of 70 points. Exam grades are:

| points in exam | Grade Chalmers | Grade GU |
|---|---|---|
| 28–41 | 3 | G |
| 42–55 | 4 | G |
| 56–70 | 5 | VG |

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

| points in exam + labs | Grade Chalmers | Grade GU |
|---|---|---|
| 40–59 | 3 | G |
| 60–79 | 4 | G |
| 80–100 | 5 | VG |

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

**Instructions and rules:**

- You should be monitored on the dedicated zoom channel while taking the exam!
- Submit the exam solution as a **PDF** file on Canvas. The solution should be typeset using your favourite software. **No** scanned hand-written notes or diagrams are allowed. (There will be a Word and Latex template for writing your solution; see comment at the end)
- 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 Latex template are available on Canvas so you can use them to deliver your answer.

**Q1** (10 p). Figure 1 shows the Java code of an implementation of strong semaphores using Java's explicit mechanism for scheduling threads (for suspending and resuming threads).

NOTE: `blocked` is a queue.

```
class SemaphoreStrong implements Semaphore {
    public synchronized void up()
    {   if (blocked.isEmpty()) count = count + 1;
        else notifyAll();    } // wake up all waiting threads

    public synchronized void down() throws InterruptedException
    {   Thread me = Thread.currentThread();
        blocked.add(me);  // enqueue me
        while (count == 0 || blocked.element() != me)
            wait();         // I'm enqueued when suspending
        // now count > 0 and it's my turn: dequeue me and decrement
        blocked.remove();  count = count - 1;   }

    private final Queue<Thread> blocked = new LinkedList<>();
```

Figure 1: Q1: A Java implementation of strong semaphores

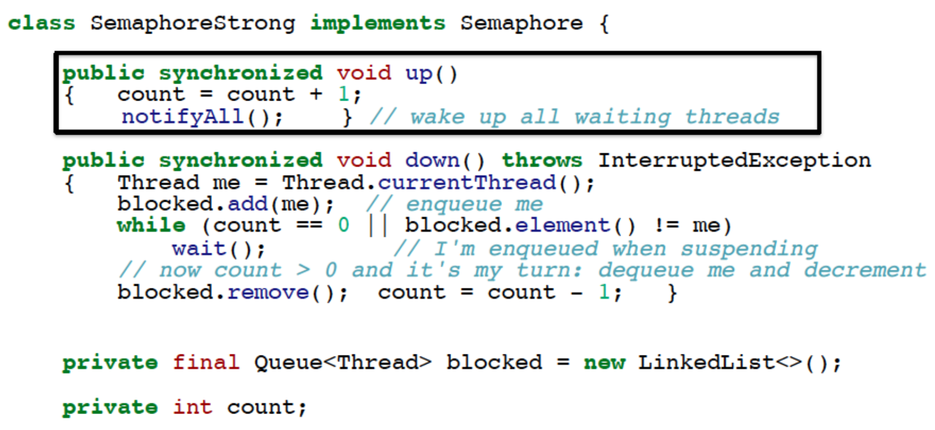
**(Part a).** *(3 p).* What makes the proposed solution to be for "strong" semaphores (in contrast to "weak" semaphores)?

*Answer: The use of notifyAll() and the fact that* `blocked` *is a FIFO queue.*

**(Part b).** *(7 p).* The programmer who suggested this solution is convinced it is correct, but when showing it to a colleague, the latter did not understand why the `up()` method checks whether the blocked queue is empty ( `if (blocked.isEmpty())` ...) before increasing the counter `count`.

Is this check really needed? If you answer YES, justify why giving an informal argument. If you answer NO, explain what happens if you do not remove the `if` checking in the code. [An answer without justification will not be sufficient to get full points.]

*Answer: The problem is that deadlock may arise: if* `count` *is zero, many threads may keep looping forever in the while loop containing the*



```
class SemaphoreStrong implements Semaphore {

    public synchronized void up()
    {    count = count + 1;
         notifyAll();     } // wake up all waiting threads

    public synchronized void down() throws InterruptedException
    {    Thread me = Thread.currentThread();
         blocked.add(me);  // enqueue me
         while (count == 0 || blocked.element() != me)
             wait();          // I'm enqueued when suspending
         // now count > 0 and it's my turn: dequeue me and decrement
         blocked.remove();  count = count - 1;    }


    private final Queue<Thread> blocked = new LinkedList<>();

    private int count;
```

Figure 2: Q1: A Java implementation of strong semaphores [Correct implementation]

*`wait()` instruction. See Figure 2 for a correct implementation.*

**Q2** (18 p). In our lectures we have seen how to use state/transition diagrams to reason about concurrent programs.

A programmer is learning about concurrent programming and as part of the learning wrote the pseudocode shown in Figure 3. The programmer (who didn't take the Principles of Concurrent Programming course!) learned from a colleague that it was a good idea to write a state/transition diagram and he started to do so, producing the partial diagram (represented as a table) shown in Figure 4.

```
   int counter = 1;     Lock lock = new ReentrantLock();
 ------------------------------------------------------------
              thread t          |          thread u
  int cnt;                      |
                                |
1 lock.lock();                  |   counter = counter - 1;    6
2   cnt = counter;              |   // end                    7
3   counter = cnt + 1;          |
4 lock.unlock();                |
5 // end                        |
```

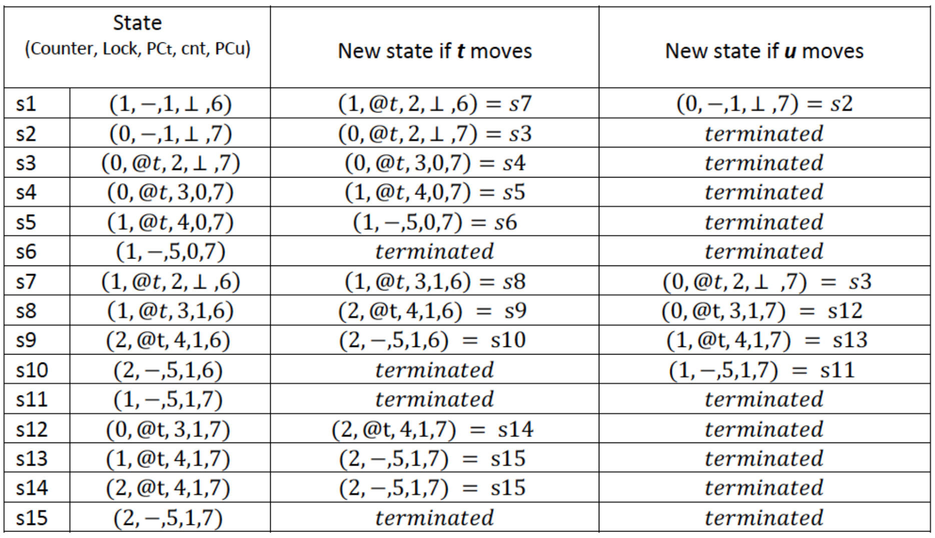
Figure 3: Q2: Pseudocode of program `counterNaive`.

| | State (Counter, Lock, PCt, cnt, PCu) | New state if ***t*** moves | New state if ***u*** moves |
|---|---|---|---|
| s1 | $(1, -, 1, \perp, 6)$ | $(1, @t, 2, \perp, 6) = s7$ | $(0, -, 1, \perp, 7) = s2$ |
| s2 | $(0, -, 1, \perp, 7)$ | $(0, @t, 2, \perp, 7) = s3$ | *terminated* |
| s3 | $(0, @t, 2, \perp, 7)$ | $(0, @t, 3, 0, 7) = s4$ | *terminated* |
| s4 | $(0, @t, 3, 0, 7)$ | $(1, @t, 4, 0, 7) = s5$ | *terminated* |
| s5 | $(1, @t, 4, 0, 7)$ | $(1, -, 5, 0, 7) = s6$ | *terminated* |
| s6 | $(1, -, 5, 0, 7)$ | *terminated* | *terminated* |
| s7 | $(1, @t, 2, \perp, 6)$ | $(1, @t, 3, 1, 6) = s8$ | |
| s8 | $(1, @t, 3, 1, 6)$ | | |
| s9 | | | |
| s10 | | | |
| s11 | | | |
| s12 | | | |
| s13 | | | |
| s14 | | | |
| s15 | | | |

Figure 4: Q2: Partial state/transition diagram (table) of `counterNaive` program.

**(Part a)** *(8 p).* Help the programmer to complete the state/transition diagram: add all the missing states and transitions so all possible computations are included.

*Answer: See Figure 5 for the complete state/transition table.*



| State (Counter, Lock, PCt, cnt, PCu) | | New state if ***t*** moves | New state if ***u*** moves |
|---|---|---|---|
| s1 | $(1,-,1,\perp,6)$ | $(1,@t,2,\perp,6) = s7$ | $(0,-,1,\perp,7) = s2$ |
| s2 | $(0,-,1,\perp,7)$ | $(0,@t,2,\perp,7) = s3$ | *terminated* |
| s3 | $(0,@t,2,\perp,7)$ | $(0,@t,3,0,7) = s4$ | *terminated* |
| s4 | $(0,@t,3,0,7)$ | $(1,@t,4,0,7) = s5$ | *terminated* |
| s5 | $(1,@t,4,0,7)$ | $(1,-,5,0,7) = s6$ | *terminated* |
| s6 | $(1,-,5,0,7)$ | *terminated* | *terminated* |
| s7 | $(1,@t,2,\perp,6)$ | $(1,@t,3,1,6) = s8$ | $(0,@t,2,\perp,7) = s3$ |
| s8 | $(1,@t,3,1,6)$ | $(2,@t,4,1,6) = s9$ | $(0,@t,3,1,7) = s12$ |
| s9 | $(2,@t,4,1,6)$ | $(2,-,5,1,6) = s10$ | $(1,@t,4,1,7) = s13$ |
| s10 | $(2,-,5,1,6)$ | *terminated* | $(1,-,5,1,7) = s11$ |
| s11 | $(1,-,5,1,7)$ | *terminated* | *terminated* |
| s12 | $(0,@t,3,1,7)$ | $(2,@t,4,1,7) = s14$ | *terminated* |
| s13 | $(1,@t,4,1,7)$ | $(2,-,5,1,7) = s15$ | *terminated* |
| s14 | $(2,@t,4,1,7)$ | $(2,-,5,1,7) = s15$ | *terminated* |
| s15 | $(2,-,5,1,7)$ | *terminated* | *terminated* |

Figure 5: Q2 (Solution): Full state/transition diagram (table) of `counterNaive` program.

**(Part b)** *(10 p).* Answer the questions below concerning the `counterNaive` program and its state/transition diagram (table). Be concise and direct (do not quote unnecessary theory about the topic and limit yourself to answer the question.)

1 How many final states are there in the final state/transition table? What can you infer about the program (in what concerns concurrency) by only observing the final states?

2 Are there data races? If your answer is positive, give all the data races. Can you see this in the state/transition diagram? (If so, how?)

3 Is there any deadlock? If you answer yes, please indicate the states with deadlocks in your state/transition table. If you answer no, say how you identify deadlocks in a state/transition diagram (or table).

4 Is the following assertion correct? *"There is no need of using a lock (nor any other synchronisation mechanism) in this program*

*as the threads* `t` *and* `u` *do not have the same code".* Justify your answer.

5 Do you agree with the following statement? *"If we modify the code of thread* `u` *as shown below the program will have the following properties: no data races, no race conditions, mutual exclusion is guaranteed."* Justify your answer.

```
lock.lock();
counter = counter - 1;
lock.unlock();
// end
```

*Answer:*

*1 There are 3. You can see that the program has a race condition (there are more than one possible result as the counter can end up with values 1 or 2).*

*2 There are two: One when u is in line 6 and t in line 2, the other when u is in line 6 and t in line 3. It is not easy to see the state/transition diagram directly unless you look at the code.*

*3 No, there are no deadlocks in this program. In general, deadlocks are seen in the diagram (or table) by getting a sink state which is not final (a non-final state without outgoing transitions).*

*4 No, the assertion is wrong: there is a need of a synchronisation mechanism since they both operate on a shared variable (*`counter`*).*

*5 Yes, access to the counter will be exclusive and all the properties will be guaranteed. (The final value of the counter will always be 1.)*

**Q3** (12 p). In Lecture 2 we introduced the concept of *barriers* as "a form of synchronisation where there is a point (the barrier) in a program's execution that all threads in a group have to reach before any of them is allowed to continue."

In Lecture 8 we discussed how to implement barriers in Erlang following the client/server architecture: an implementation in Erlang is shown in Figure 6.

```
 1 -module(barrier).
 2 -export([init/1,wait/1]).
 3
 4 init(Expected) ->
 5    spawn(fun () -> barrier(0, Expected, []) end).
 6
 7 wait(Barrier) ->
 8    Ref = make_ref(),
 9    Barrier ! {Arrived, self(), Ref},
10    receive {continue, Ref} -> goahead end.
11
12 barrier(Arrived, Expected, PidRefs)
13    when Arrived =:= Expected ->
14      [To ! {continue, Ref} || {To, Ref} <- PidRefs],
15      barrier(0, Expected, []);
16 barrier(Arrived, Expected, PidRefs) ->
17    receive
18       {Arrived, From, Ref} ->
19         barrier(Arrived+1, Expected, [{From, Ref}|PidRefs])
20    end.
```

Figure 6: Q3: A barrier implementation in Erlang.

**(Part a)** *(6 p).*

1 Is the code correct? If not explain what is wrong and correct the code. (You don't need to write the whole program again, simply identify the lines with the error and give the correct implementation.)

2 The code in Figure 6 shows the server side of a full implementation. Write a a client that interacts with the barrier.

*Answer:*

1 *No, it is not correct. There are two errors: in the reception of the messages, the message should match an atom as first element and not a variable (it should be* `arrived, From, Ref` *and not* `Arrived, From, Ref`*; similarly in the* `wait` *function it should be* `arrived, self(), Ref` *and not* `Arrived, self(), Ref`*)*

2 *According to slide 4 of Lecture 8, the clients (processes arriving to the barrier) should execute the following function:*

```
process(Barrier) ->
    % code before barrier
    barrier:wait(Barrier) % synchronize at barrier
    % code after barrier
    process(Barrier).
```

*Other answers to be consider correct are those without the recursive call. However, if the client is too specific (executing a fixed number of times), the answer is considered to be wrong.*

**(Part b)** *(6 p).* Below follows 3 statements about the barrier implementation shown in Figure 6. Determine whether the statement is True or False. For each case, justify your answer (for both your False and True answers). The justification should be convincing showing you understand the reasons for your answer.

1 The first part of the definition of the `barrier` function (lines 12 till 15) will only be executed (pattern matched) when the expected number of processes has arrived to the barrier, in which case a message will be sent to all the processes so they can pass the barrier.

2 The implementation shown in the figure is the server implementation of a reusable barrier.

3 Line 2 of the code `-export([init/1,wait/1])` is not really needed as no process need to call the functions `init` and `wait`.

*Answer:*

1 *True: the* `when` *condition (line 13) exactly matches that the function should be executed if the number of arrived processes is as expected, and the list comprehension in line 14 will indeed send a message so processes can continue.*

2 *True: the code only shows the server part, and it is indeed a reusable barrier as indicated by the call in line 15 (recursive call reinitialising the barrier).*

3 *False: you need to call* `init` *to initialise the barrier and the client processes need to call* `wait` *to synchronise at the barrier.*

**Q4** (10 p). Figure 8 shows a parallel implementation of the recursive function `famousFun` shown in Figure 7.

The programmer who wrote the code in Figure 8 intended to have a parallel implementation of the recursive function `famousFun` only when the parameter `Fu` satisfy the following properties:

- The function `Fu` is associative (i.e., `Fu(Fu(A,B),C) = Fu(A,Fu(B,C))`).
- For every element `E` of the list, `Fu(E,A) = Fu(A,E) = E`.

```
famousFun(Fu, E, []) -> E;
famousFun(Fu, E, [H|T]) ->  Fu(H, famousFun(Fu, E, T)).
```

Figure 7: Q4: The function `famousFun`.

```
famousFunPar(Fu, E, [])  -> E;
famousFunPar(Fu, E, [H]) -> Fu(H, E);
famousFunPar(Fu, E, L) ->
   Mid = length(L) div 2,
   {L, R} = lists:split(Mid, List),
   Myself = self(),
   Rp = spawn(fun() -> Myself ! {self(), famousFunPar(Fu, E, R)} end),
   Lp = spawn(fun() -> Myself ! {self(), famousFunPar(Fu, E, L)} end),
   Fu(receive {Lp, Lr} -> Lp end, receive {Rp, Rr} -> Rr end).
```

Figure 8: Q4: A parallel implementation of `famousFun`.

**(Part a).** *(2 p).* Give the result, and a step-by-step description, of calling the following:

`famousFun(fun (X,Y) -> X * Y end, 10, [2,3,4])`.

(Code in Figure 7.)

*Answer: 240 (it will traverse the list till arriving to the empty list and then compute the product of 10 and 4, then 40 * 3, then 120 * 2).*

**(Part b).** *(3 p).* What does `famousFun` compute? (Code in Figure 7.)

*Answer: It computes the "reduce" (or "foldr") function (as defined in Lecture 9, slide 18)*

**(Part c).** *(5 p).* Five programmers discuss the `famousFunPar` function (code in Figure 8) and they come to different arguments on its correctness (assuming the properties for `Fu` as specified above). You will find below a statement done by each one of the programmers concerning the correctness of the code. Indicate which answer is correct and justify why (In case you choose one of the options saying that the implementation contains $n$ errors, then you should identify and say what the errors are).

1 The implementation is correct.

2 It is not correct: it has 1 error.

3 It is not correct: it has 2 errors.

4 It is not correct: it has 3 errors.

5 It is not correct: there are more than 3 errors.

*Answer: There are different answers to the question. Unambiguously there is at least one error: in line 9 (the first parameter of Fu should give Lr and not Lp). Now, you might consider that there are 2 or 3 errors depending on whether you consider the code to be pure Erlang or Erlang-like. Both of the following answers are correct:*
*- If the code is Erlang, there are more 2 errors, in line 3 and 4 L should be List. So, there are 3 errors in total.*
*- If you interpret that the code is Erlang-like (without unification in pattern matching), you might consider that the only additional error is in line 5, where List should be L. So, there are 2 errors in total.*

*The correct implementation is (assuming pure Erlang code):*

```
1 famousFunPar(_, E, [])  -> E;
2 famousFunPar(Fu, E, [H]) -> Fu(H, E);
3 famousFunPar(Fu, E, List) ->
4    Mid = length(List) div 2,
5    {L, R} = lists:split(Mid, List),
6    Myself = self(),
7    Rp = spawn(fun() -> Myself ! {self(), famousFunPar(Fu, E, R)} end),
8    Lp = spawn(fun() -> Myself ! {self(), famousFunPar(Fu, E, L)} end),
9    Fu(receive {Lp, Lr} -> Lr end, receive {Rp, Rr} -> Rr end).
```

**Q5** *(20 p)*. We have seen in Lecture 10 how to implement a *set* data structure using linked lists. We first showed an implementation that worked for sequential access, and then different linked set implementations allowing for parallel access.

Figures 9 and 10 show one such implementation: a "fine-grained locking" implementation of parallel linked sets. This implementation extends the `SequentialSet<T>` class seen in the course (the methods `rawAdd`, `rawHas` and `rawRemove` are methods defined in the `SequentialSet<T>` class to add an element to the set, check whether an element is in the set, and remove an element from the set, respectively).

**(Part a)** *(10 p)*. The implementation contains 2 errors. Find them and propose a fix. You don't need to rewrite the whole program:

- If the error is on a specific line just point out that ("Error in Figure X, line Y") and write down the correct line which is intended to replace the faulty one.
- If the error is about some missing piece of code, just indicate in between which lines the missing code should be inserted ("Code missing in Figure X, in between lines Y1 and Y2") and provide the new code to be inserted in that place.
- If the error is about reordering two more lines, indicate which lines are the faulty ones ("Wrong order in Figure X, in between lines Y1 and Y2") and provide the right code to be inserted in their place

*Answer:*

**Error 1:** *Error in Figure 9, line 20: instead of* `curr.unlock();` *it should be* `curr.lock();`

**Error 2:** *Error in Figure 10; missing code in lines 25-26 (both locks should be released). The try-finally code of the has function should be as follows:*

```
finally {
            pred.unlock();
            curr.unlock();
        }
```

**(Part b)** *(10 p)*. Let us assume that you want to implement a queue and use a linked list as the underlying data structure. You look at the implementation of the fine-grained locking version of a parallel linked set (the correct version of the code shown in Figures 9 and 10) for inspiration, and you want to refactor it. In particular, you want to implement an *unbounded queue* (instead of a *set*), and you will then write a class `Queue<T>`.

Background: A *queue* is a FIFO (First In, First Out) data structure with the following operations:

`enqueue(Q,E)`: Adds element `E` to the queue `Q`. (It gives as result the updated new queue.)

`dequeue(Q)`: It retrieves (removes) an element of the queue. The elements are popped (dequeued) in the same order in which they are pushed (enqueued). If the queue is empty, then it is said to be an Underflow condition and no element is given; otherwise it gives as result the dequeued element.

`front(Q)`: Get the front element from the queue `Q` without removing it.

`rear(Q)`: Get the last element from the queue `Q` without removing it.

We say that a queue is *unbounded* when there is no limit on the number of elements it might contain (you can always enqueue a new element).

In what follows you will get 10 assertions concerning the implementation of a class `Queue<T>` that allows for parallel access. The assertions are both general statements about such an implementation and also related to the possibility of reusing the code for sets (the correct version of the code shown in Figures 9 and 10): refactoring `FineSet<T>` into a new class `Queue<T>`.

For each assertion, you need to say whether it is correct or not. You need to justify your answer in each case.

NOTE: An answer without a justification will not be granted full points.

1 The `enqueue` method will be exactly the same as the `add` method (just changing names). In other words, can you use `add` as it is to implement `enqueue`?

2 You don't need to use a `key` in the queue data structure as the elements don't need to be added in order according to the key.

3 The `dequeue` method is different from the `remove` among other things because in a queue we don't need to remove elements from the middle of the (linked) data structure.

4 Implementing a `Queue<T>` class by refactoring the `FineSet<T>` class is a bad idea since there are too many changes to be made (not much can be reused).

5 A class `Queue<T>` that implements a linked queue that supports parallel access requires the use of locks (in other words, it is impossible to program a linked queue that supports parallel access without using locks).

6 As for `FineSet<T>`, any implementation of a class `Queue<T>` allowing for parallel access might get an inconsistency if one thread tries to add (enqueue) an element while another tries to remove (dequeue) it.

7 Adding (enqueuing) an element on a parallel queue is not problematic in general if the list has four elements or more.

8 The implementation of a class `Queue<T>` allowing for parallel access cannot be implemented with semaphores.

9 It is not possible to implement a class `Queue<T>` allowing for parallel access without using CAS (compare-and-set) operation.

10 The implementation of a lock-free queue data structure (a class `Queue<T>` without using locks) presented in Lecture 11 is a paradigm of how to implement a parallel queue in every object oriented language, being unconditionally correct.

*Answer:*

*1 False: You need to make a lot of changes as you add elements only at one end of the queue (and not in a specific part of the structure). ("The* `enqueue` *method will be exactly the same as the* `add` *method (just changing names)"). ALTERNATIVE ANSWER: True: Though the changes are pretty trivial by wrapping the code with something that adds the keys to enqueued elements.*

*2 True ("You don't need to use a* `key` *in the queue data structure as the elements don't need to be added in order according to the key." The keys are used for efficiency reasons.)*

*3 True ("The* `dequeue` *method is different from the* `remove` *among other things because in a queue we don't need to remove elements from the middle of the (linked) data structure.") ADDITION: So* `FineSet` *is more general than what we need and it would be simple to use it for the implementation of queue.*

*4 True ("Implementing a* `Queue<T>` *class by refactoring the* `FineSet<T>` *class is a bad idea since there are too many changes to be made (not much can be reused')"). ALTERNATIVE ANSWER: False: you just need to wrap the methods and the implementation could be very simple (if you want to use keys and just reimplement the way you insert and remove elements).*

5 *False: You don't require locks, as shown by the implementation proposed in Lecture 11 using CAS ("A class* `Queue<T>` *that implements a linked queue that supports parallel access requires the use of locks (in other words, it is impossible to program a linked queue that supports parallel access without using locks)").*

6 *True ("As for* `FineSet<T>`*, any implementation of a class* `Queue<T>` *allowing for parallel access might get an inconsistency if one thread tries to add (enqueue) an element while another tries to remove (dequeue) it").*

7 *False: Adding elements on any parallel data structure might be problematic is there are more than one thread operating on it. ("Adding (enqueuing) an element on a parallel queue is not problematic in general if the list has four elements or more").*

8 *False: You can, as semaphores are more general than locks and you can implement a parallel queue with locks ("The implementation of a class* `Queue<T>` *allowing for parallel access cannot be implemented with semaphores.")*

9 *False: You can, as shown in Lecture 11 ("It is not possible to implement a class* `Queue<T>` *allowing for parallel access without using CAS (compare-and-set) operation").*

10 *False: The lock-free implementation given in Lecture 11 is not unconditionally correct since requires garbage collection (slide 14). You may also argue that the answer is false since the proposed solution is not "a paradigm of how to implement a parallel queue in every object oriented language" for two reasons: first, it might depend on the primitive constructs the language provides to ensure atomicity, second you may use locks to implement a parallel queue.*

```java
package sets;

public class FineSet<T> extends SequentialSet<T>
{
    public FineSet() {
        super();
    }

    @Override
    protected Position<T> find(Node<T> start, int key) {
        Node<T> pred, curr;
        pred = start;
        pred.lock();
        curr = start.next();
        curr.lock();
        while (curr.key() < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next();
            curr.unlock();
        }
        return new Position<T>(pred, curr);
    }

    @Override
    public boolean add(T item) {
        Node<T> node = newNode(item);
        Node<T> pred = null, curr = null;
        try {
            Position<T> where = find(head, node.key());
            pred = where.pred;
            curr = where.curr;
            return rawAdd(pred, curr, node);
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }

\\ code continues in Figure 5.
```

Figure 9: Q5: A “fine-grained locking” implementation of parallel linked sets.

```
    @Override
    public boolean remove(T item) {
        int key = item.hashCode();
        Node<T> pred = null, curr = null;
        try {
            Position<T> where = find(head, key);
            pred = where.pred;
            curr = where.curr;
            return rawRemove(pred, curr, key);
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }

    @Override
    public boolean has(T item) {
        int key = item.hashCode();
        Node<T> pred = null, curr = null;
        try {
            Position<T> where = find(head, key);
            pred = where.pred;
            curr = where.curr;
            return rawHas(curr, key);
        } finally {
        }
    }

    @Override
    protected Node<T> newNode(T item) {
        return new LockableNode<>(item);
    }

    @Override
    protected Node<T> newNode(int key) {
        return new LockableNode<>(key);
    }
}
```

Figure 10: Q5: A "fine-grained locking" implementation of parallel linked sets. [CONT.]