Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT392

11 March 2024

Exam supervisor: G. Schneider (gersch@chalmers.se, 072 974 49 64)
(Exam set by G. Schneider, based on the course given Jan-Mar 2024)

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: between 28–41 (3), between 42–55 (4), 56 or more (5).

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

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

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will 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 (19p). In what follows you will get 6 assertions concerning locks, semaphores and other synchronisation algorithms. For each assertion, you need to say whether it is correct (true) or not (false). You need to justify your answer in each case (an answer without a justification will not be granted full points).

1 If you do not use a `try {...} finally {...}` after acquiring a lock in Java, there is a risk of getting other processes to starve. **(2p)**

Answer: True. Starvation may happen if the process acquiring the lock raises an exception in the critical section (thus never releasing the lock and making all the other process waiting for the lock to starve).

2 The pseudocode of the program shown in Fig. 1 guarantees mutual exclusion, it is deadlock-free and it is starvation-free. If you answer *true*, then explain how the semaphore guarantees all those properties. If you answer *false*, explain which of the three properties are violated and why (you should also provide the right code that guarantees the three properties). **(4p)**

Answer: False. It is deadlock-free and starvation-free (it is never the case that both get stuck because they cannot execute `sem.down()`*), but it is not mutual exclusive. Indeed, since the semaphore is initialised to 2 then both threads can access the critical section at the same time. The right code is to initialise the semaphore to 1.*

3 In the correct version of the pseudocode of the program shown in Fig. 1, the semaphore could be replaced by a lock. If you answer *true*, then explain how this is done (provide the replacements to be done in the program). If you answer *false*, explain why this cannot be done. NOTE: Your answer should be based on the *correct* version of the program, meaning that if you answered *true* in the previous question then you should use the original program; if you answered *false* then you should use your corrected version. **(3p)**

Answer: True. A semaphore with capacity 1 used in the way it is used in the correct version of the program acts as a lock. The replacements to be done (in the correct coce) are: a) replace `sem.down()` *by* `lock()`*; b) replace* `sem.up()` *by* `unlock()`*.*

4 The pseudocode of the program shown in Fig. 2 is a solution to the barrier synchronisation problem for 2 threads. If you answer *true*, then explain how the semaphores guarantees that the code acts as a barrier. If you answer *false*, explain what is wrong and provide a correct solution (you may write the new code or simply say which parts need to be corrected and how). **(4p)**

```
int counter; Semaphore sem = new Semaphore(2);
```

thread t

```
int c,d;

1         sem.down();
2         d = counter - 1;
3         c = d * 2;
4         counter = c + d + 1;
5         sem.up();
```

thread u

```
int c;

          sem.down();          6
          c = counter + 1;     7
          counter = c - 1;     8
          sem.up();            9
```

Figure 1: Q1-2: Pseudocode of program `someCounting`.

Answer: False. The program does not act as a barrier as each thread can execute both `up()` and `down()` without needing to wait for the other. The problem is with the initialisation of the semaphores. The correct solution would be to initialise both semaphores to 0.

5 Fig. 3 shows an attempt to solve the mutual exclusion problem without using locks or semaphores (only using atomic reads and writes). Is the code correct? (Does it satisfy mutual exclusion?) If yes, explain how this is achieved. If not, explain what is the problem and hint to a solution (you don't need to give a new (corrected) code; simply stated what the problem is and sketch a solution). **(4p)**

Answer: False. The code does not satisfy mutual exclusion.The code is a rewritten version of one presented in lecture 3 slide 21 (first failed attempt towards Peterson's algorithm): `enter` has been replaced by `goin` and the `await` instruction in the pseudocode presented in the lecture has been replaced by the (inner) `while` loop. The problem is explained in Lecture 3 slide 22: "The problem seems to be that `await` is executed before setting `enter`, so one thread may proceed ignoring that the other thread is also proceeding." One correct solution is Peterson's algorithm.

6 Peterson's algorithm is a solution to the mutual exclusion problem by using test-and-set operations, and it only works for 2 threads. **(2p)**

Answer: False. Peterson's algorithm is a solution to the mutual exclusion problem based on atomic reads and writes, and it can be generalised to more than 2 threads.

```
Semaphore go0 = new Semaphore(1); Semaphore go1 = new Semaphore(1)};
```

thread t0		thread t1	
1	`// code before barrier`	`// code before barrier`	6
2	`go0.up();`	`go1.up();`	7
3	`go1.down();`	`go0.down();`	8
4	`// code after barrier`	`// code after barrier`	9

Figure 2: Q1-4: Pseudocode of program `Barrier`.

```
boolean [] goin = {false, false};
```

thread t0		thread t1	
1	`while (true) {`	`while (true) {`	6
2	`  // entry protocol:`	`  // entry protocol:`	7
3	`  while (goin[1]) { };`	`  while (goin[0]) { };`	8
4	`  goin[0] = true;`	`  goin[1] = true;`	9
5	`  critical sect { ... }`	`  critical sect { ... }`	10
6	`  // exit protocol:`	`  // exit protocol:`	11
7	`  goin[0] = false;`	`  goin[1] = false;`	12
8	`}`		

Figure 3: Q1-5: Pseudocode of program `Mutual exclusion with only atomic reads and writes`.

Q2 (12p). In what follows you have 4 subquestions (Parts a to d) concerning the dining philosophers. Each part a multiple-choice question. The grading for each question is as follows:

- For a right answer you will get 3 points;
- If you do not answer the question, you will get 0 points;
- If you answer wrongly, you will get -1 (a negative point).

The total amount of points will be done summing all the points for each part. So, if you answer correctly Part a, incorrectly Part b, and do not answer any of the rest, you will get 2 points (3 points for Part a minus 1 point for Part b, and 0 for the rest). Note that no negative points for the whole question Q2 will be given (the minimum number of points you may get for the whole question is 0). That is, if you do answer wrongly in each part, you will not get -5 but 0.

Here is an implementation of a protocol for the *dining philosophers problem*. There are n philosophers and n forks that are implemented by locks. Every philosopher has an identifier in the range $1, \ldots, n$, and the `left_fork` of philosopher i is fork i and the `right_fork` of philosopher i is fork $(i \ mod \ n) + 1$.

```
entry() {
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
}
critical section { eat(); }
exit() {
 left_fork.release(); // release left fork
 right_fork.release();// release right fork
}
```

(Part a). (3p)

The four necessary conditions for a deadlock are:

1 Mutual exclusion: threads may have exclusive access to shared resources.
2 Hold and wait: a thread may request one resource while holding another one.
3 No preemption: resources cannot forcibly be released from threads that hold them.
4 Circular wait: two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

Which one of the following assertions is true on the above implementation of `entry()`?

a) The implementation satisfies mutual exclusion only.

b) The implementation satisfies all four necessary conditions.

c) The implementation satisfies no preemption and circular wait, but not the other conditions.

d) The implementation does not satisfy mutual exclusion.

e) None of the conditions are satisfied.

Answer: Option b:

- *Mutual exclusion - each lock is held by at most one philosopher at a time.*
- *Hold and Wait - each philosopher requires two forks to eat. They take one and try to acquire the second.*
- *No preemption - the only way to release the locks is by executing the exit protocol.*
- *Circular wait - in a scenario where all philosophers reach for their left fork one by one we have the all philosophers are waiting for a resource that the next thread in the chain is holding.*

(Part b). (3p)

See below a variant of the `entry()` protocol for a solution to the dining philosophers problem which uses one additional global lock. Assume that the `critical section` and `exit` protocol are as before (as well as the `eat()` and `release()` functions):

```
entry() {
  global_lock.acquire();
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
  global_lock.release();
}
```

Which of the following assertions is true on this implementation of `entry()`? (More than one option might be true.)

a) The implementation ensures that circular waiting do not occur.

b) Adding just one global lock is not enough to guarantee deadlock-freedom, even if the locks are fair.

c) Adding the global lock may introduce starvation, even if all locks are fair.

d) The implementation guarantees deadlock-freedom provided all locks are fair and philosophers eat and release fairly.

e) The new `entry()` protocol is incorrect since a philosopher may not be able to pick up both forks to eat.

Answer: Options a and d are correct. Option a since the global lock ensures that no two philosophers try to acquire locks at the same time. Furthermore, the only way to release the main lock is when acquiring both forks. If some process is locking the main lock and waiting for one of their forks, by implication, the philosopher it is waiting for is not waiting for anything because it released the main lock. Option d because of the same reasons given for option a.

(Part c). (3p)

Which ones of the following assertions are true about the dining philosophers problem and solution? (More than one option may be true.)

a) The dining philosophers problem is a theoretical exercise in concurrency, without any connection to real problems.
b) Any working solution (i.e., guaranteeing mutual exclusion, deadlock-freedom and starvation-freedom) to the dining philosophers problem must be implemented using some kind of synchronisation mechanism (locks, semaphores, etc.).
c) A producer-consumer problem can be reduced to a dining philosopher problem.
d) There is no solution to the dinner philosopher problem for more than 10 philosophers.
e) A solution to the dining philosopher problem could be simply to ensure that just one philosopher at a time eats, while the others think, and then take turn to eat. This would work but it will be essentially a sequential algorithm, not exploiting the possibilities of concurrency.

Answer: Options b and e are true.

(Part d). (3p)

See below an Erlang implementation of `fork` and the functions `get_fork` and `put_fork` intended to be a (partial) solution to the dining philosophers problem. Remember that each fork corresponds to a process initially running `fork()` and each philosopher corresponds to a process that alternates between calling `get_fork()` twice and `put_fork()` twice.

```
% a fork not held by anyone
```

```
fork() ->
  receive
    {get, From, Ref} -> From ! {ack, Ref},
                        fork(From) % fork held
  end.

% a fork held by Owner
fork(Owner) ->
  receive
    {put, Owner, _Ref} -> fork()  % fork not held
  end.

get_fork(Fork) ->
  Ref = make_ref(),
  Fork ! {get, self(), Ref},
  receive {ack, Ref} -> ack end.

put_fork(Fork) ->
  Ref = make_ref(),
  Fork ! {put, Ref}.
```

Which ones of the following assertions are true about the above implementation? (More than one option may be true.)

a) The `fork()` implementation is wrong since it should not wait for the message `{get,From,Ref}`.
b) The `get_fork` function is incorrect as the following line of code `receive {ack,Ref}` should be replaced by this one: `receive {ack,self(),Ref}`.
c) The `put_fork` function is incorrect as the following line of code `Fork ! {put,Ref}` should be replaced by this one: `Fork ! {put,self(),Ref}`.
d) All the functions shown in the implementation are correct.
e) The lines `Ref = make_ref()` are not needed in `get_fork` and `put_fork`.

Answer: Only option c is true: the corresponding line in the code is indeed incorrect and it should be replaced by `Fork ! {put, self(), Ref}` *(which matches the* `receive` *instructions in* `fork()`*).*

Q3 (15p). In what follows you have 5 subquestions (Parts a to e) concerning different topics seen in the course. Each part a multiple-choice question. The grading for each question is as follows:

- For a right answer you will get 3 points;
- If you do not answer the question, you will get 0 points;
- If you answer wrongly, you will get -1 (a negative point).

The total amount of points will be done summing all the points for each part. So, if you answer correctly Part a, incorrectly Part b, and do not answer any of the rest, you will get 2 points (3 points for Part a minus 1 point for Part b, and 0 for the rest). Note that no negative points for the whole question Q3 will be given (the minimum number of points you may get for the whole question is 0). That is, if you do answer wrongly in each part, you will not get -5 but 0.

(Part a). (3p)

This question is about different signalling policies in monitors. In Fig. 4 you see the pseudocode of a monitor program `PrintCnt` that prints the value of `counter`. Threads may execute `inc()` and `print()` in every order and as many times as they want. What are the possible printed values when calling method `print()`?

```
monitor class PrintCnt {
   private int counter = 0;
   private Condition isOne = new Condition();

   public void print() {
        while (counter != 1) isOne.wait();
        System.out.println(counter);
   }

   public void inc() {
         counter += 1;
         if (counter == 1) isOne.signal();
   }
}
```

Figure 4: Q3 - Part a: Pseudocode of program `PrintCnt`.

a) It never prints "1" if the monitor uses a *signal and continue* discipline

b) It always prints "1" if the monitor uses a *signal and continue* discipline

c) It always blocks if the monitor uses a *signal and wait* discipline

d) It may may block forever or it may print "1"

e) It may print any number equal or bigger than "1"

Answer: Option d). Reason: the only way to print is by not executing the while when the counter is 1 in which case it will print 1. The other possibility is that some of the threads waiting in the `wait` *condition wakes up and then proceed to print, but this would only happen when a* `signal` *is executed, which may only happen once (given the condition* `counter==1`*). Thus, all the process waiting to wake up will be blocked as the signalling will happen only once.*

(Part b). (3p)

Fig. 5 shows four implementations of the function `up()` of a semaphore using `notify()` and the internal counter of the semaphore, `count`. The implementation of `down()` is shown in Fig. 6. Which one of the four implementations of `up()` does NOT work?

a) `up1()`

b) `up2()`

c) `up3()`

d) `up4()`

```
public synchronized void up1() {
    ++count;
    notify();
}


public synchronized void up2() {
    notify();
    ++count;
}
```

```
28  public void up3() {
29      notify();
30      synchronized (this) {
31          ++count;
32      }
33  }
34
35  public void up4() {
36      synchronized (this) {
37          ++count;
38      }
39      notify();
40  }
```

Figure 5: Q3 - Part b: Four semaphore implementations of up().

```
public synchronized void down() throws InterruptedException {
    while (count == 0) { wait(); }
    count = count - 1;
}
```

Figure 6: Q3 - Part a: Semaphore implementation of down().

Answer: Option c). Reason: `up3()` *does not work because the notify() might wake a thread that will go ahead, take the lock, see that count is* 0 *and then go back to waiting. The thread calling* `up3()` *will then increase the counter but the waiting thread will not be woken up leading to a deadlock.*

(Part c). (3p)

In Fig. 7 you can see a parallel version of a program that computes the multiplication of numbers from m to n.

Note that the function has two special cases: it gives 1 if $m > n$, and it gives m when $m == n$).

Also note that the division operation ("/") truncates the decimal part (e.g., 1/2 gives 0).

```
class ParallelMul extends RecursiveTask<Integer> {
     int m, n;

     protected Integer compute() {
         if (m == n) return m;
         if (m > n) return 1;
         if (m < n) {
          int mid = m + (n-m)/2;                    // mid point
          ParallelMul lower = new ParallelMul(m, mid);
          ParallelMul upper = new ParallelMul(mid+1, n);
          lower.fork();
          upper.fork();
          return lower.join() * upper.join();
         }
    }
}
```

Figure 7: Q3 - Part c: Java program `ParallelMul`.

The program is run to compute the product of integers from $m = 1$ to $n = k$ (with $k > 1$).

What is the maximal number of cores that would run this code effectively? That is, adding more cores will not speed up the computation.

NOTE: We are assuming an idealised case where we have one thread per core without special techniques involved, and that we ignore how Java's VM (or other languages and underlying multithreading technologies implementations) behaves. That is, the question is a theoretical investigation of the problem, no making any further assumptions on hardware optimisations.

a) 1, there practically is no parallelism

b) 2^k (that is, 2 to the power of k)

c) $k!$ (that is, the factorial of k)

d) k^2 (that is, $k * k$)

e) k

Answer: Option e). Reason: the program forks till reaching the base case in which case there are k parallel threads that will then be joined performing the multiplication. All other threads are waiting.

(Part d). (3p)

According to Amdahl's law, if the fraction p of a program can be parallelised, then, the maximum speedup that can be achieved by n processes is $\frac{1}{(1-p)+\frac{p}{n}}$. You have a program where 10% of the program must be done sequentially and 90% of the program can be parallelised. What is the maximum speedup that you can achieve given unlimited resources (i.e., increase the number of processes as you wish).

a) One cannot achieve speedup at all.

b) With a very large number of processes, one can achieve any wanted speedup.

c) The program can run more than 10 times faster.

d) The program cannot run more than 10 times faster.

Answer: Option d). Reason: The denominator is at least 1/10. So the fraction is always smaller than 10.

(Part e). (3p)

Which of the following programs does not have data races? In all cases, $t1$ and $t2$ are threads executing at the same time sharing the variables at the top. The programs are numbered 1, 2, and 3 from left to right.

a) Program 2.

b) Program 3.

c) Program 1 and program 2.

d) Program 1 and program 3.

Answer: Option d). Reason: Program 1 the write are never accessible. So it cannot have a data race. Program 2 has a write in t2 that is not ordered with respect to the writes in t1. Program 3 As flag is volatile,

```
boolean x=false, y=false;

t1 {
    if (x)
        y=true;
}

t2 {
    if (y)
        x=true;
}
```

```
int cnt = 0;

t1 {
    lock.lock();
    cnt++;
    cnt++;
    lock.unlock();
}

t2 {
    cnt++;
}
```

```
volatile boolean flag = false;
int cnt = 0;

t1 {
    if (flag) {
        cnt++;
    }
}

t2 {
    cnt++;
    flag = true;
}
```

Figure 8: Q3 - Part e: Programs with or without data races.

an access to the if in t1 happens only after the flag is set to true in t2. This means that the increment in t1 happens after the increment in t2.

Q4 (12p). In what follows you have 4 subquestions (Parts a to d) concerning Erlang processes (client-server architecture, process communication, etc.). Each part a multiple-choice question. The grading for each question is as follows:

- For a right answer you will get 3 points;
- If you do not answer the question, you will get 0 points;
- If you answer wrongly, you will get -1 (a negative point).

The total amount of points will be done summing all the points for each part. So, if you answer correctly Part a, incorrectly Part b, and do not answer any of the rest, you will get 2 points (3 points for Part a minus 1 point for Part b, and 0 for the rest). Note that no negative points for the whole question Q4 will be given (the minimum number of points you may get for the whole question is 0). That is, if you do answer wrongly in each part, you will not get -5 but 0.

(Part a) (3p).

An Erlang implementation (following the client/server architecture) is shown in Fig. 9. Below follows 4 statements: which one is true?

```
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 9: Q4-(a): An Erlang implementation of a barrier.

a) The code is correct.

b) The code is not correct. There is one error: a `when` condition is missing in one of the cases defining `barrier` (after line 16).

c) The code is not correct. There is one error: the `when` condition (line 13) is not correct.

d) The code is not correct. There are two errors, both related to how Erlang treats variables and atoms.

Answer: The correct option is (d): 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}`*).*

(Part b) (3p).

Below follows 4 statements about the barrier implementation shown in Fig. 9. Only one of the statements is true, which one is it?

a) 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.

b) The implementation shown in the figure is the server implementation of a non-reusable barrier.

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

d) In order for the pattern matching to work, we should add a `when` condition after line 16 (second part of the `barrier` definition).

Answer: Option (a): 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.*

(Part c) (3p).

Given the two Erlang process of Fig. 10: what does process Q print?

a) Process q's pid (process identifier).

b) self(2).

c) A list with 2 elements.

d) The number 2.

e) The number 0.

f) Nothing.

process P

```
1 p() -> % Q is q's pid
2   self() ! self(),
3   receive self() ->
4    Q ! {self(), fun(X) -> X-2 end}
5   end.
```

process Q

```
q() ->                                  6
  receive {Q, G} ->                     7
    io:format("~p", [G(2)])             8
  end.                                  9
```

Figure 10: Q4-(c): Erlang processes.

Answer: Option (e): Process Q receives a message with two parameters, process p's id (P) and a function that decrements the parameter by 2. The function is received as "G" by process q, and then prints the application of the function to the parameter 2 (thus giving 2-2).

(Part d) (3p).

It follows 6 different assertions concerning Erlang processes. Only one of them is false, which one?

a) Messages sent to a server remain in the mailbox indefinitely unless there is at least one concrete receiver instruction that pattern matches the message.

b) A `when` clause can only be used in the first occurrence of a function definition, otherwise the process will block if it does not pattern match.

c) When writing a generic server if the computation of a handling function on the input fails the result may create an exception. The exception needs to be explicitly handled in order to notify the client that an error has occurred. To handle any possible exception, you can use Erlang's `catch(E)` built-in function.

d) Erlang supports functionalities to implement client-server architectures as well as other kind of process interactions based on asynchronous communication.

e) By defining handling functions in a suitable manner, it is possible to implement a "hot upgrading" in Erlang. (Note: a "hot upgrading" is when you can update a functionality without needing to stop the execution of the process.)

f) Implementing a *fair* solution in Erlang to the readers-writers problem requires a hierarchical solution forcing existing readers to finish and not accepting new readers as soon as a writer arrives.

Answer: Option (b): It is false since the `when` *clause can be used in any part of the function definition (not necessarily the first one), and has nothing to do with blocking when no pattern matching happens.*

Q5 (12p). Your project manager has asked you to implement a *queue* using a linked list as the underlying data structure. You look at the implementation of the fine-grained locking version of a parallel linked set (code shown in Figures 11 and 12) for inspiration, and you want to refactor it. In particular, you were asked to implement a *bounded queue* (instead of a *set*). Your task is to write a class `Queue<T>`.

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

`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.

`enqueue(Q,E)`: It adds element `E` to the queue `Q`. If the queue is *full*, then it is said to be an Overflow condition and no element can be added. It gives as result the updated new queue with the new element added, or the very same queue in case of an Overflow condition.

`front(Q)`: It gets the front element of the queue `Q` without removing it.

`last(Q)`: It gets the last element of the queue `Q` without removing it.

A queue is said to be *bounded* when there is a limit on the number of elements it might contain; we call the maximum number of elements the queue may contain its *bound* (or *limit*).

A queue is *full* when it has as many elements as its limit. Note that you can only enqueue a new element when the queue is not *full*).

For bounded queues, we have the following new operation:

`bound(Q)`: It gives the bound of the queue `Q` (the maximum number of elements the queue may contain).

In what follows you will get 12 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 about the possibility of reusing the code for sets (the code shown in Figures 11 and 12): refactoring `FineSet<T>` into a new class `Queue<T>`.

For each assertion, you need to state 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 bound (limit) of the queue is not needed as we always know how many elements the queue has.
Answer: False: the bound has nothing to do with the current length of a queue (it is the maximum number of elements the queue may have, and you cannot know that number in advance).

2 The queue data structure should have a `key` in order to add the elements in the right position according to the key.

Answer: False: You don't need to use the key as the elements don't need to be added in order according to the key. The keys are used for efficiency reasons.

3 The `enqueue` method is essentially the same as the `add` method (just changing names). In other words, you can use `add` as it is to implement `enqueue`.

Answer: 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). In particular, you need to check whether the queue is not full before adding the element (as the queue is bounded).

4 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.

Answer: True: The `dequeue` method only removes elements from one end of the data structure, while the set `remove` operation may remove elements from the middle of the (linked) data structure.

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).

Answer: False: You don't require locks, as shown by the implementation proposed in Lecture 11 using CAS.

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

Answer: False: You can, as semaphores are more general than locks and you can implement a parallel queue with locks.

7 Implementing a `Queue<T>` class by refactoring the `FineSet<T>` class is a good idea since there are not too many changes to be made.

Answer: True: 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, adding a check for the bound when needed).

8 It is possible to implement a class `Queue<T>` allowing for parallel access without using CAS (compare-and-set) operation

Answer: True: You can, as shown in Lecture 11.

9 The `bound` method requires the use of a lock (or any other synchronisation mechanism) as it might create inconsistency if accessed by more than one thread.

Answer: False: the `bound` *just returns a value that is not supposed to be updated anywhere (and thus it doesn't require to be protected with any sync mechanism).*

10 Adding (enqueuing) an element on a parallel queue is not problematic in general if the list has at most three elements.

Answer: False: Adding elements on any parallel data structure might be problematic is there are more than one thread operating on it, independently of the number of elements.

11 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.

Answer: True. This is the case for any data structure where threads may simultaneously add and remove elements (as the local view of the pointers may differ).

12 The implementation of a lock-free queue data structure (a class `Queue<T>` without using locks) presented in Lecture 11 is an unconditionally correct way to implement a parallel queue in any language.

Answer: False: The lock-free implementation given in Lecture 11 is not unconditionally correct since requires garbage collection (slide 17). 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.

```
1  package sets;
2
3  public class FineSet<T> extends SequentialSet<T>
4  {
5      public FineSet() {
6          super();
7      }
8
9      @Override
10     protected Position<T> find(Node<T> start, int key) {
11         Node<T> pred, curr;
12         pred = start;
13         pred.lock();
14         curr = start.next();
15         curr.lock();
16         while (curr.key() < key) {
17             pred.unlock();
18             pred = curr;
19             curr = curr.next();
20             curr.lock();
21         }
22         return new Position<T>(pred, curr);
23     }
24
25     @Override
26     public boolean add(T item) {
27         Node<T> node = newNode(item);
28         Node<T> pred = null, curr = null;
29         try {
30             Position<T> where = find(head, node.key());
31             pred = where.pred;
32             curr = where.curr;
33             return rawAdd(pred, curr, node);
34         } finally {
35             pred.unlock();
36             curr.unlock();
37         }
38     }
39
40 \\ code continues in Figure 12.
```

Figure 11: Q5: A "fine-grained locking" implementation of parallel linked sets.

```
1     @Override
2     public boolean remove(T item) {
3         int key = item.hashCode();
4         Node<T> pred = null, curr = null;
5         try {
6             Position<T> where = find(head, key);
7             pred = where.pred;
8             curr = where.curr;
9             return rawRemove(pred, curr, key);
10        } finally {
11            pred.unlock();
12            curr.unlock();
13        }
14    }
15
16    @Override
17    public boolean has(T item) {
18        int key = item.hashCode();
19        Node<T> pred = null, curr = null;
20        try {
21            Position<T> where = find(head, key);
22            pred = where.pred;
23            curr = where.curr;
24            return rawHas(curr, key);
25        } finally {
26            pred.unlock();
27            curr.unlock();
28        }
29    }
30
31    @Override
32    protected Node<T> newNode(T item) {
33        return new LockableNode<>(item);
34    }
35
36    @Override
37    protected Node<T> newNode(int key) {
38        return new LockableNode<>(key);
39    }
40 }
```

Figure 12: Q5: A "fine-grained locking" implementation of parallel linked sets. [CONT.]