Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Monday, 14 March 2022

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

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

- 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.

Q1 *(18 p).*

Figure 1 shows the pseudocode of program `countMany`. Let us assume that a main program launches the two threads and whenever both terminate, it prints the result of `counter`.

```
                    int counter = 0;     int i = 0;
          thread t                              thread u
  int cnt;                              int cnt;

1 while (i<10) {                        while (i<10) {              7
2   i = i+1;                              i = i+1;                  8
3   cnt = counter;                        cnt = counter;            9
4   counter = cnt + 1;                    counter = cnt + 1;        10
5 }                                     }                           11
6 // end                               // end                       12
```

Figure 1: Q2: Pseudocode of program `countMany`.

(Part a). *(4 p)* What is the minimum and the maximum values the program can print (the value of `counter` after termination)? Justify your answer.

(Part b). *(4 p)* How many data races the program has? List them (use the line numbers of the code).

(Part c). *(5 p)* How many possible values the program can print? List them and explain.

(Part d). *(5 p)* Let us assume that the intention of the programmer was that program `countMany` always terminates with `counter`=10. How would you guarantee that by adding synchronisation primitives? Name at least one mechanism to enforce that, and explain how to do it (Give the new code).

Q2 *(18 p)*

Figure 2 shows the Java code of an implementation of strong semaphores using Java's explicit mechanism for scheduling threads (for suspending and resuming threads), and figure 3 shows the method calling the semaphore. NOTE: `blocked` is a queue.

```
1  class SemaphoreStrong implements Semaphore {
2      public synchronized void up()
3      {   if (blocked.isEmpty()) count = count + 1;
4          else notifyAll();    } // wake up all waiting threads
5
6      public synchronized void down() throws InterruptedException
7      {   Thread me = Thread.currentThread();
8          blocked.add(me);  // enqueue me
9          while (count == 0 || blocked.element() != me)
10             wait();         // I'm enqueued when suspending
11         // now count > 0 and it's my turn: dequeue me and decrement
12         blocked.remove();  count = count - 1;   }
13
14     private final Queue<Thread> blocked = new LinkedList<>();
```

Figure 2: Q2: A Java implementation of strong semaphores

```
15 class StrongSemUser implements Runnable {
16     private SemaphoreStrong sem = new SemaphoreStrong(1);
17
18     public void run()
19     {   while (true) {
20             // Non critical
21             sem.down();
22             // Critical
23             sem.up();
24         }
25     }
```

Figure 3: Q2: Run method calling the semaphore.

(Part a). *(2 p)* We have shown in the lectures that the code is wrong. Explain why this is the case (what is the reason for the error, and what is the error). What is the fix? (You do not need to write the whole code, just say what needs to be added or removed from the wrong code.)

(Part b). *(3 p)* Can you reproduce the error if there is *at most* **one** thread active? What is the minimum number of threads you need to (re)produce the error?

(Part c). *(6 p)* Reproduce the error with the minimum number of threads.

Note: Indicate which threads are in the `blocked` queue at any moment, and the value of `count` Use names for different threads with indexes (e.g., t0, t1, t2, etc.) and indicate in which line number they are at each execution step. For instance, *t0.21, t0.7, t0.8, t1.21* indicates four steps of the execution of threads t0 and t1: the first three instructions being executed by thread t0 (before calling `sem.down()` and then taking two steps into the method), and then fact that t1 has arrived to instruction with number 21. In case there are more than one instruction in a line (e.g., l.12) and the thread executes both instructions, then you repeat that in your sequence: t0.9, t0.12, t012... Also, you should skip the comments.

Start with thread t0 in line 21, with an empty `blocked` queue (`blocked`= {}), and `count` = 1. We encourage you to write comments at each step, to help you understand what is going on. These are the first two steps:

t0.21, blocked = {}, count=1 *% First thread wants to call down*
t0.7, blocked = {}, count=1 *% First thread calls down and starts executing method*
...

(Part d). *(3 p)* Can more than one thread go into the `down()` method at the same time? Explain.

(Part e). *(2 p)* Would the error still be such if the `down()` method is not declared as `synchronized`? Explain.

(Part f). *(2 p)* Would there still be an error if the `wait()` would hold the lock? Explain.

Q3 *(10 p).*

The pseudocode below shows a sequential program `inc(k)`:

```
int n = 1;
int x;

while (n =< k) {
   x = n;
   n = n + x;
}
print(n);
```

Note that the program always prints 2^i for all values of k such that $2^{i-1} \leq k < 2^i$. So, it prints:

2 ($i = 1$) if $k = 1$;
4 ($i = 2$) for values of k s.t. $2 \leq k < 4$;
8 ($i = 3$) for values of k s.t. $4 \leq k < 8$;
...;
64 ($i = 6$) for values of k s.t. $32 \leq k < 64$;
and so on.

(Part a). *(2 p)* How many iterations does the program do, for different values of k?

(Part b). *(5 p)* A programmer wants to write a program based on threads to parallelise `inc(k)` and writes a first version as follows:

```
int n = 1;
```

thread t_h

```
1    int x;
2
3    x = n;
4    n = n + x;
```

Besides that, there is a `main` program that prints the result (`n`) after all the threads have finished.

Answer the following questions (Note: you should only consider the case when `n` is shared and `x` is local to each thread):

1. The programmer wrote that there are h threads. What are the possible results if $h = 10$ (what would the program print)? (You don't need to enumerate all the possible results but rather explain the pattern.)
2. What is the maximum amount of threads you can have in order to get the same result as the sequential version assuming the threads are each executed atomically. That is, you should assume that there is **no** interleaving in between the commands executed by each thread: the only interleaving is between threads.

(Part c). *(3 p)* Write a parallel version of the program based on threads, taking into account the answer you gave in Part b. Your solution should guarantee determinism, and the program should give the same result as the original program. Use semaphores as a synchronisation mechanism.

Q4 *(12 p).*

A programmer wants to provide an Erlang solution to the problem of concurrent access to a spreadsheet, where many so-called *checkers* can access the shared resource simultaneously, while so-called *updaters* may access it exclusively. The programmer implements a `module(sheet)` with the following functions:

```
init(Name)            % register spreadsheet with Name
begin_update(Sheet)   % access Sheet for updating
end_update(Sheet)     % release updating access
begin_check(Sheet)    % access Sheet for checking
end_check(Sheet)      % release checking access
```

The `init()` function initialises an empty spreadsheet and registers it with name `Name`.

Checkers and updaters continuously, and asynchronously, try to access the spreadsheet, as shown below.

For checkers:

```
checker(Sheet) ->
   sheet:begin_check(Sheet),
   % code to check spreadsheet
   sheet:end_check(Sheet),
   checker(Sheet).
```

For updaters:

```
updater(Sheet) ->
   sheet:begin_update(Sheet),
   % code to update spreadsheet
   sheet:end_update(Sheet),
   updater(Sheet).
```

The programmer wrote the following implementation of the server function `sheet_CaU`, claiming it guarantees mutual exclusion concerning access to the shared spreadsheet (remember: many checkers can access the spreadsheet at the same time, but updaters must get exclusive access to it):

```
sheet_CaU(Updaters, Checkers) ->                                  1
  receive                                                         2
    {begin_update, Who, Ref} when (Updaters =:= 0)  ->            3
         Who ! {ok_to_update, Ref},                               4
         sheet_CaU(Updaters+1, Checkers);                         5
    {end_update, Who, Ref} ->                                     6
         Who ! {ok, Ref},                                         7
         sheet_CaU(Updaters-1, Checkers)                          8
    {begin_check, Who, Ref} when Updaters =:= 0 ->                9
```

```
        Who ! {ok_to_check, Ref},                                    10
        sheet_CaU(Updaters, Checkers+1);                             11
    {end_check, Who, Ref} ->                                         12
        Who ! {ok, Ref},                                             13
        sheet_CaU(Updaters, Checkers-1);                             14
end.                                                                 15
```

(Part a) *(5 p)* Is the claim that the server `sheet_CaU` guarantees mutual exclusion correct? If so, give an informal argument on why this is the case. If the claim is not true: explain what is wrong with the implementation of the server and give a correct implementation to satisfy mutual exclusion (if the new implementation only concerns a couple of lines, just indicate what is the change to be done to those lines).

(Part b) *(5 p)* Does the (correct) solution guarantee starvation freedom? Explain.

(Part c) *(2 p)* What happens with those requests that cannot be served immediately by the server? Are they lost?

Q5 *(12 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 (code shown in Figures 4 and 5) for inspiration, and you want to refactor it. In particular, you want to implement a *bounded 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`. 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.

`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 *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 said to be *full* when it has as many elements as its limit. Note then that you can always enqueue a new element provided the queue is not *full*).

For bounded queues, we have the following new operation:

`bound(Q)`: 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 related to the possibility of reusing the code for sets (the correct version of the code shown in Figures 4 and 5): 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. You need to use a `key` in the queue data structure as the elements have to be added in order according to the key.

2. 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`?

3. The bound (limit) of the queue is not really needed as we always know how many elements the queue has.

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.

5. 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.

6. 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).

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

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

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.

10. 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.

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

12. 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.

```
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 5.
```

Figure 4: 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 5: Q5: A "fine-grained locking" implementation of parallel linked sets. [CONT.]