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

```
  int cnt;

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

thread u

```
int cnt;

while (i<10) {          7
  i = i+1;              8
  cnt = counter;        9
  counter = cnt + 1;    10
}                       11
// 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.

Answer: Minimum: 1 (Thread t enters the loop once and stops in line 4, so i=1 and cnt=0; thread u executes the whole loop 9 times and exits with i=10 and counter=8 (since i is incremented at the beginning of the loop, the counter is only incremented 8 times); thread t executes line 4 making counter=1 (since cnt=0) and then exits the loop).
Maximum: 11 (Thread t enters the loop and stops before executing line 2. Thread u executes the whole loop exiting with i=10 and counter=10. Thread t now executes lines 3 and 4, making counter=1
NOTE: As mentioned in the course and in the exam hall: we assume that interleaving only happens in between instructions, and not "inside" an instruction (e.g., between lines 2 and 3, but not in line 2 — between reading the value of i, incrementing i and assigning the new updated value to i) .)

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

Answer: There are 3 concerning the variable `counter`*, one for each combination when the variable is being written (and written or read by the other thread):*
lines 3 and 10,
lines 4 and 9,

lines 4 and 10.
Besides, there are 3 more for the looping variable (`i`)
lines 2 and 7,
lines 2 and 8
lines 1 and 8.
So, 6 data races in total.

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

Answer: The program can print any value between 1 and 11. They correspond to the possible interleavings by one thread iterating a certain number of times and "recording" the value of the local variable `cnt` before updating `counter`.

(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).

Answer: The solution could be to use a lock. You need to declare a `Lock lock = new ReentrantLock();` (using Java syntax), and in both threads then use the lock as follows:
Call `lock.lock()` just before the `while`, and call `lock.unlock()` at the end of the loop.

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.

```java
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 2: Q2: A Java implementation of strong semaphores

```java
class StrongSemUser implements Runnable {
    private SemaphoreStrong sem = new SemaphoreStrong(1);

    public void run()
    {    while (true) {
              // Non critical
              sem.down();
              // Critical
              sem.up();
         }
    }
```

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.)

Answer: The problem is that it might deadlock. The reason is the line checking whether `blocked` *is empty before incrementing the counter in the* `up()` *method.* As shown in Lecture 3, slide 64, the fix is to remove the `if-then-else` in lines 3-4 (just increment and notify all)

(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?

Answer: One thread is not enough. You need at least 2 threads (the red and the blue threads in the slides). [Comment: "active" means that threads are, or have been, executing (including those blocked).]
Partially correct answer: If you interpret being active as threads not being blocked in a wait() statement, then the answer is Yes (since all the other threads will be blocked, and then, according to this interpretation they are not "active"). In any case, note that the second question does not refer to "active" threads so no confusion should arise: you still need at least 2 threads to reproduce 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*
...

Answer: Only 2 threads, t0 and t1. Deadlock can be witnessed with the following sequence of instructions: t0.21, blocked = {}, count=1
t0.7, blocked = {}, count=1
t0.8, blocked = {}, count=1
t0.9, blocked = {t0}, count=1

t0.12, blocked = {t0}, count=1
t0.12, blocked = {}, count=1
t0.23, blocked = {}, count=0
t1.21, blocked = {}, count=0
t1.7, blocked = {}, count=0
t1.8, blocked = {}, count=0
t1.9, blocked = {t1}, count=0
t1.10, blocked = {t1}, count=0
t0.3, blocked = {t1}, count=0
t0.4, blocked = {t1}, count=0
t0.24, blocked = {t1}, count=0 % thread t0 finishes executing the loop once, and then it will call down() again, getting into a deadlock.

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

Answer: Yes, because of the semantics of the `wait()`*: the lock of the synchronized method is released when calling* `wait()`*, allowing other threads to come in.*

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

Answer: Yes. Making the method not to be synchronized just would allow more threads to enter the method "at the same time", but that would not correct the error.

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

Answer: Yes, because the error is there even if only one thread is inside the `down()` *method. (Any other thread will be then deadlocked not being able to execute down().)*

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?

Answer: It loops i times for all values of k such that $2^{i-1} \leq k < 2^i$.
So, it iterates 1 time ($i = 1$) if $k = 1$; it iterates 2 times ($i = 2$) for values of k s.t. $2 \leq k < 4$; it iterates 3 times ($i = 3$) for values of k s.t. $4 \leq k < 8$; ...; it iterates 6 times ($i = 6$) for values of k s.t. $32 \leq k < 64$; and so on.

(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

```
int x;

x = n;
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.

Answer:

1. *If h=10 and all the threads are executed sequentially without interleaving, this would correspond to iterate 10 times, meaning (according to Part a) that the result would be 1024 (2^{10}). That said, if there is interleaving, it may print other values. The minimum value the program will print would be 11 (when all the threads have executed line 3 so all have the local variable x to be equal to 1 and then the threads will be updating the shared variable n as many times as threads there are starting with $n = 1$)*
2. *As many as the iterations needed for each value of k as indicated in Part a. For instance, if $k = 10$ (k is the parameter of the sequential program `inc()`) then you are in the case $8 \le k < 16$, and you need 4 iterations.*

(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.

Answer: See below (Note: there are i threads, where i is as in Part b).

```
int n = 1;     Semaphore s = new Semaphore(1); //capacity 1
```

thread t_i

```
int x;

s.down();
x = n;
n = n + x;
s.up();
```

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) ->
  receive
    {begin_update, Who, Ref} when (Updaters =:= 0)  ->
         Who ! {ok_to_update, Ref},
         sheet_CaU(Updaters+1, Checkers);
    {end_update, Who, Ref} ->
         Who ! {ok, Ref},
         sheet_CaU(Updaters-1, Checkers)
    {begin_check, Who, Ref} when Updaters =:= 0 ->
```

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

(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).

Answer: No, it doesn't guarantee mutual exclusion. The programmer did a mistake by forgetting to check that there are no checkers when trying to update. The solution is to change line 3 by adding a condition as follows:

```
{begin_update, Who, Ref} when (Updaters =:= 0)
                          and (Checkers =:= 0) ->
```

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

Answer: No, the server gives priority to checkers over updaters:

- *new checking requests get served without waiting as long as a checker is active*
- *updating requests waiting in the mailbox have to wait until the last checker sends an end_ check message*
- *if checking requests keep arriving (queuing in the mailbox), the waiting updating requests will never execute (and may thus starve).*

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

Answer: No, they are not lost. Requests that cannot be served implicitly queue in the sheet's mailbox; they will be served as soon as the spreadsheet is freed.

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.

Answer:

1. *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.*

2. *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).*

3. *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 "compute" that number).*

4. *True ADDITION: So* `FineSet` *is more general than what we need and it would be simple to use it for the implementation of queue.*

5. *False: you need to do more than simple rename. In particular you don't need the key and you need to check for the bound. ALTERNATIVE 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).*

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

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

8. *True: You can, as shown in Lecture 11*

9. *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. *True*

11. *False: Adding elements on any parallel data structure might be problematic is there are more than one thread operating on it.*

12. *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.lock();
        }
        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 4: Q5: A “fine-grained locking” implementation of parallel linked sets.

```java
    @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 {
            pred.unlock();
            curr.unlock();
        }
    }

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

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

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