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

```
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.

```
    @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.]