CSE 428: Solutions of exercises on Concurrency and Java

The superscript "(d)" stands for "difficult". Exercises similar to those marked with "(d)" might appear in candidacy exams, but not in the standard exams of CSE 428.

---

1. Consider the example of the producers and consumers in the LN of Lecture 16. It is possible to see that the system is deadlock-free, in fact
   - Initially the buffer is empty and no threads are in the waiting state (i.e. in the wait set associated to the buffer).
   - When a producer inserts a datum in the buffer, all threads exit the wait set of the buffer. One consumer will be able to proceed.
   - When a consumer removes the datum from the buffer, all threads exit the wait set of the buffer. One producer will be able to proceed.

   For each of the following modifications, say whether we lose the property of deadlock-freedom or not. Justify your answers.
   1. One producer instead of three, and still two consumers.
   2. Use `notify()` instead of `notifyAll()` in the `get()`method. Note: the method `notify()` removes an arbitrary thread from the wait set of the object.
   3. One producer, two consumers, and use `notify()` instead of `notifyAll()` in the `put()` method.
   4. Invert the order of the statements `is_empty = false` and `notifyAll()` in the `put()` method.

   ### Solution

   1. The system is still deadlock-free. The justification given above is still valid.
   2. The system is not deadlock-free anymore. When a consumer executes the `get()`, the invocation of `notify()` might remove from the wait set another consumer instead of a producer. If all the producers are all in the wait set, no thread will be able to proceed.
   3. In this case the use of `notify()` is not harmful, because the producer, when it executes the `put()`, can only remove a consumer from the wait set, since there are no other producers around.
   4. Inverting the order of the statements in the `put()` method is not harmful, because the method is synchronized. No thread will be able to perform a new test on `is_empty` until the `put()` terminates.

---

2. Define a class Stack (LIFO buffer) in Java, as an implementation of the following interface:

```
interface Stack_interface(){
   void push(int);
   int pop();
   boolean is_empty();
}
```

   Hint: you can follow the scheme of the definition of a Queue (FIFO buffer) given in the program Eratosthenes of Lecture 17. Note that there was no method `is_empty` because, for that particular application, it was not required.

   ### Solution

```
class Stack implements Stack_interface {
    private class Element {
        int info;
        Element next;
        Element(int n, Element e) { info = n; next = e; }
    }
    private Element first;
    public Stack() { first = null; }
    public synchronized void push(int n){
       first = new Element(n,first);
       if (first.next == null) then notifyAll(); //if the stack was empty then notify all waiting consumers
    }
    public synchronized int pop(){
        try {
            while (first == null) wait();
```

```
            } catch (InterruptedException e) {
                return 1;
            }
            int n = first.info;
            first = first.next;
            return n;
        }
        public synchronized boolean is_empty(){
            return first == null;
        }
}
```

---

3. Consider the program Eratosthenes of Lecture 17. Would the program still be correct if the FIFO buffers were replaced by LIFO buffers? Motivate your answer.

Solution

The program would not be correct. Consider for instance the configuration in which we have one filter only, for the prime 2. Suppose that this filter sends to its output buffer the data 3, 5, 7 and 9 before the sieve process is able to make any operation on this buffer. Then suppose that sieve starts picking data from it. Since the buffer is LIFO, sieve will read the number 9 first, and it will treat it as a prime (print it, and create a new filter for it), which is incorrect.

---

4. Consider the solution to the problem of the dining philosophers (Assignment 6).
    1. Is the system deadlock-free?
    2. Is it possible that a philosopher never eats?
    3. Is it possible that a chopstick is never taken?
    4. What happens if `notify()` in `Chopstick.release()` is replaced by `notifyAll()`?
    5. What happens if `notify()` in `Ticket_box.release_ticket()` is replaced by `notifyAll()`?
    6. What happens if the `while` in `Ticket_box.take_ticket()` is replaced by `if`?
    7. What happens if we declare the method `Philosopher.try_to_eat()` as synchronized? With this modification, could we avoid declaring synchronized the methods of `Ticket_box` and of `Chopstick`?

    Motivate your answers.

Solution

1. The system is deadlock-free, because there must be at least a philopher that is able to get both chopsticks, since there are n philosophers, n chopsticks, and only n-1 tickets, and only the philosophers with a ticket are allowed to try to get a chopstick.
2. It is possible that a philosopher never eats (though quite unlikely). This can happen if he is never able to get a ticket, or if his neighbours are always faster than him in getting the chopsticks he has access to.
3. It is possible that a chopstick is never taken (though extremely unlikely). This can happen in the case in which its left philosopher never succeed to get the other chopstick (because his left neighbour is faster than him) and its right philosopher never succeed to get the ticket.
4. This modification does not change anything: there can be at most one thread waiting for the same chopstick, hence `notify()` and `notifyAll()` are equivalent in this case.
5. Again, the modification does not change anything: there can be at most one thread waiting for a ticket.
6. The program would be incorrect. In fact, it might be the case that a philosopher is in the wait set of the ticket box, then it gets notified that a ticket has been released, but, by the time he is scheduled and he gets the lock again, the ticket has been taken already. Thus it is incorrect to let him execute the instruction `number_of_tickets--` without doing the test again on the availability of a ticket.
7. It does not make any sense to declare `Philosopher.try_to_eat()` as synchronized. There is only one thread that executes the methods of an object of the class Philosopher: the object (philosopher) himself. Hence there is no risk of interference, and no need of synchronization. In particular, making this modification does not spare the necessity of declaring as synchronized the methods of `Ticket_box` and of `Chopstick`. It is easy to show that the philosophers can interfere with each other when operating on the common resources (tickets and chopsticks), and therefore we need to code these operations inside critical sections. For example, consider the portion of code

```
if (left_chopstick.is_free() && right_chopstick.is_free()) {
     left_chopstick.take();
     right_chopstick.take();
     ...
}
```

in `Philospopher.try_to_eat()`. Suppose that two neighbour philosophers are executing the test at the same time (this is possible even if the method `try_to_eat()` is synchronized, because the lock is per object, not per method (since it is not static)). Suppose that both the philosophers pass the test: if `Chopstick.take()` were not synchronized, they could end up in taking both the same chopstick.

---

5. Consider again the solution to the problem of the dining philosophers. For each of the following modifications, say whether we can have deadlock or not. Justify your answers.
   1. Change the method `try_to_eat()` so that, after the philosopher has eaten, he releases the ticket first, and then the chopsticks.
   2. There are n tickets
   3. There are n tickets and n+1 chopsticks (i.e. there are two neighbours philosophers such that in between them there are two chospticks instead than one, so that each of them has his own chopstick)

   Motivate your answers.

Solution

1. The system is still deadlock-free. There might be a moment in which each philosopher is holding exactly one chopstick, but at least one of them (the last philosopher who has released a ticket) is in the phase in which he has already eaten, and he is going to release the chopstick next.
2. In this situation we can have deadlock. Obviously, allowing n tickets in the box makes invalidates the tickets-based solution to deadlock: all philosophers can hold a ticket at the same time, and enter the competition for the chopstick. Then it's possible that each philosopher gets hold of exactly one chopstick.
3. In this case we cannot have a deadlock. Having one more chopsticks ensures that at least one philosopher can get hold of two chopsticks, and proceed. Note that the presence of the tickets is not relevant for this question.

---

6. Consider the second solution to the definition of a buffer with two positions given in Exercise 3.2 of (CSE 428, Spring 99) MT 2.
   1. Suppose that we replace the statement

      ```
      synchronized (from) { int k = from.get();  to.put(k); }
      ```

      in `Demon.run()` with the statement

      ```
      { int k = from.get();  to.put(k); }
      ```

      (i.e. we eliminate the synchronization on `from`). Does `Buffer2` still represent a buffer with two positions?
   2. Remember that the methods `Buffer1.put()` and `Buffer1.get()` are `synchronized`. Suppose that, instead, we make `synchronized` the methods `Buffer2.put()` and `Buffer2.get()`. Would the solution still be interference-free?
   3. Suppose now that all the methods `Buffer1.put()`, `Buffer1.get()`, `Buffer2.put()` and `Buffer2.get()` are `synchronized`. Would the solution still be deadlock-free?
   4. (d) Suppose that we replace the statement

      ```
      synchronized (from) { int k = from.get(); to.put(k); }
      ```

      in `Demon.run()` with the statement

      ```
      if (!to.is_full()) synchronized (from) { int k = from.get(); to.put(k); }
      ```

      What are the consequences?

Solution

1. `Buffer2` would become a buffer with three positions. In fact, after the demon has removed the datum from the first `Buffer1`, and before it puts the datum in the second `Buffer1`, a producer would be already allowed to put

another datum in the first `Buffer1`. In other words, we could have the situation that `Buffer2` is holding three data: one in the first `Buffer1`, one in the second `Buffer1`, and one in the temporary variable `k` of the demon.

2. The solution would not be interference-free. In fact, the demon would interfere with the producers on the first `Buffer1` (the demon requires the lock on `from`, but a producer would not), and it would interfere with the consumers on the second `Buffer1`.
3. The solution would not be deadlock-free. In fact, we could have the following situation: the second `Buffer1` full, the demon in the wait set of the second `Buffer1` and holding the lock of the first `Buffer1` (because it's trying to transfer a previous datum from the first `Buffer1` to the second `Buffer1`), and a producer getting the lock of the `Buffer2` and then waiting for the lock of the first `Buffer1` to be released. Since no consumer will then be allowed to execute `Buffer2.get()`, the situation is stuck.
   Another simpler case of deadlock is when a consumer starts first: it goes in the wait set of the second `Buffer1`, while holding the lock on `Buffer2`, so that no producer can proceed.
4. By performing the test `!to.is_full()` we could avoid locking the first `Buffer1` when the second `Buffer1` is full, and in this way a producer could test the state of the first `Buffer1` and, in case it is full, do something else (if it wishes). However, there is a particularly unlucky case in which a livelock could occur: this is when the consumers keep being scheduled exactly when the demon is performing the test `!to.is_full()`. Since this test requires the lock on the second `Buffer1`, a consumer will never be allowed to consume (thus the consumers are stuck, and the only process which keeps executing is the demon). Note that the livelock could be avoided by assuming some form of fairness (or randomized scheduling) in the implementation.
   Note: this kind of livelock, for this program, has been observed on the Java implementation on Unix.
