Chalmers | GÖTEBORGS UNIVERSITET
Carlo A. Furia, Computer Science and Engineering

Principles of Concurrent Programming TDA384/DIT391

Saturday, 17 March 2018, 14:00–18:00

(including example solutions)

Teacher/examiner: Carlo A. Furia (`furia@chalmers.se` – 0317721675)
(the examiner will visit the exam rooms twice: around 15:00 and around 17:00)

Exercise 1: Concurrency properties *(17 points)*

Consider the following concurrent program P, where two threads t and u execute in parallel and access two shared integer variables `x` and `y`.

```
int x = 0; int y = 0;
```

thread t	thread u	
	`int a = x;`	2
1 `x = 3;`	`int b = y;`	3
	`y = a + b;`	4

Question 1.1 (2 points): Does program P have **race conditions**? Justify your answer.

The final value of `y` may be `0` (if thread u executes line 2 before thread t executes line 1) or `3` (in all other cases). Thus, the final value of P's computation depends on the interleaving of concurrent threads, which is the definition of race conditions.

Question 1.2 (2 points): What are the **critical sections** of the code executed by thread t and of the code executed by thread u?

Since both threads access shared variables (by reading or writing them) in every statement of their code, their critical sections correspond to the whole code they each execute. If no other threads in the system have access to variable `y`, u's critical section is concretely limited to line 2, since it is *as if* variable `y` was local to u.

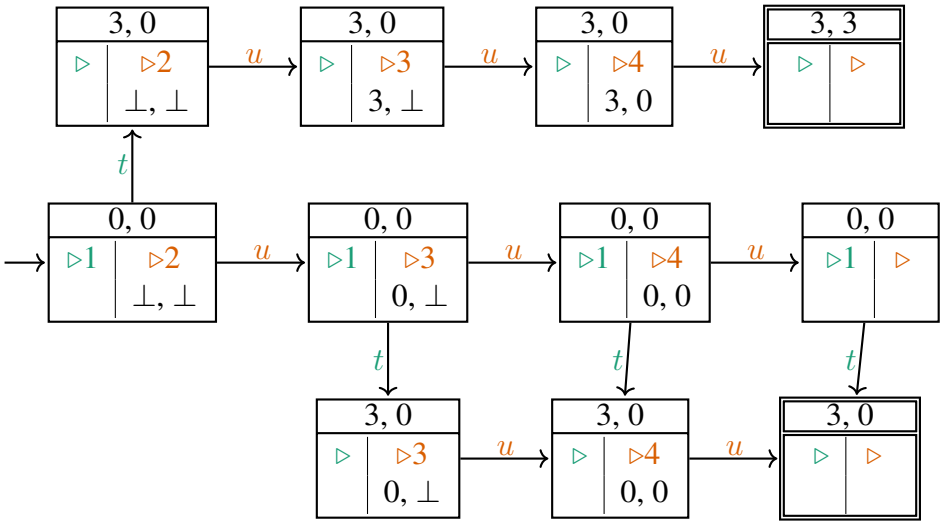
Question 1.3 (1 point): List all **data races** that exist in program P.

There is one data race, corresponding to the instruction pair on lines $(1, 2)$. Note that when two threads access *different* memory locations there is no data race.

Question 1.4 (3 points): Write a complete **trace** corresponding to a possible executions of program P such that the *final* value of `y` is `0`. The trace must be a sequence of program states, where each state indicates: the value of `x` and of `y`, the program counters (the line number of the statement to be executed next) of t and of u, and the value of u's local variables `a` and `b`.

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t: 1	pc_u: 2 a: $\perp$ b: $\perp$	x: 0 y: 0
2	pc_t: 1	pc_u: 3 a: 0 b: $\perp$	x: 0 y: 0
3	pc_t: 1	pc_u: 4 a: 0 b: 0	x: 0 y: 0
4	pc_t: 1	pc_u: done	x: 0 y: 0
5	done	pc_u: done	x: 3 y: 0

Question 1.5 (8 points): Build a complete **state/transition diagram** modeling all possible executions of program P. Each state of the diagram should indicate: the value of `x` and of `y`, the program counters (the line number of the statement to be executed next) of t and of u, and the value of u's local variables `a` and `b`. Remember to mark the *initial* and *final* states of the diagram. (Three states of the diagram are given to get you started.)



Question 1.6 (1 point): How can you determine whether program P has race conditions *exclusively based on its state/transition diagram*?

The state/transition diagram has two *different* final states, which means P has race conditions.

Exercise 2: Erlang – servers *(14 points)*

In this exercise, we build an Erlang program in the style of **servers**, which stores information about a *bank account*.

A bank account stores its *balance* (a number, representing the money currently in the account), and supports three operations:

- `deposit` adds a given amount of money to the account's *balance*;

- `withdraw` removes a given amount of money from the account's *balance* if it is available; if the given amount of money is not available in the account, `withdraw` does not do anything;
- `balance` sends the account's current *balance* to the client.

Correspondingly, the server accepts messages in the form `{operation, Argument}` where `operation` is the atom `deposit`, `withdraw`, or `balance`, and `Argument` is the operation's argument. Note that executing `deposit` and `withdraw` does not return anything to the client.

Question 2.1 (2 points): Describe the **arguments** of a **server event loop** function `account`, which implements the server behavior described above: the function's arguments are those needed to store the server's **state**. Describe the intended *type* of each argument and what each argument represents in the state.

The server needs one argument `Balance`, of numeric type, storing the current balance: `account(Balance)`

Question 2.2 (6 points): Define the **server event loop** function `account`, which implements the server behavior described above and uses the arguments given in the answer to the previous question.

```
account(Balance) ->
   receive
      {deposit, Amount} ->
         account(Balance + Amount);
      {withdraw, Amount} when Amount =< Balance ->
         account(Balance - Amount);
      {balance, From} ->
         From ! Balance,
         account(Balance)
   end.
```

Question 2.3 (2 points): Define function `start()`, which spawns a process running the server (corresponding to an empty bank account), and returns to the caller the PID (process identifier) of the spawned process.

```
start() -> spawn(fun () -> account(0) end).
```

Question 2.4 (2 points): Define function `deposit(Account, Amount)`, which interacts with the bank account server with PID (process identifier) `Account` to execute operation `deposit` with the given `amount`. (Note that function `deposit` will be called by client processes.)

```
deposit(Account, Amount) -> Account ! {deposit, Amount}.
```

Question 2.5 (2 points): Define function `balance(Account)`, which interacts with the bank account server with PID (process identifier) `Account` to get and return the account's balance. (Note that function `balance` will be called by client processes.)

```
balance(Account) ->
   Me = self(),
   Account ! {balance, Me},
   receive Balance -> Balance end.
```

Exercise 3: Concurrent data structures *(14 points)*

Recall that a data structure implementation is *thread safe* if its operations can be executed by multiple concurrent threads without running into race conditions. In this exercise, you will evaluate different implementations of a destructive operation on a linked data structure in Java, analyzing whether they are thread safe.

Recall the various implementations of linked sets presented during the course. They are all variants implementing the same *interface*, consisting of operations to remove elements, add elements, and test whether an element is in the set. In this exercise, we extend the interface with a *new operation* `void cut(int n)`, which removes *the first n nodes* in the linked chain that originates from the head. For example, `cut(2)` removes the first element (whatever its item value is) that is directly connected to the head, as well as the element immediately following it. If `n` is greater than the number of elements stored in the set, `cut(n)` empties the set completely. Note that `cut(n)` must never remove the `head` or `tail` special nodes.

An implementation of method `cut(int n)` that works in a *sequential setting* is:

```
1  public void cut(int n) {
2    int m = 0;
3    Node<T> upto = head;              // set upto to the head node
4    do {
5      upto = upto.next;               // move lower to next node in chain
6      m += 1;                         // increment m by one
7    } while (m <= n && upto != tail)  // until n nodes are scanned or tail is reached
8    head.next = upto;                 // redirect head to upto
9  }
```

Question 3.1 (3 points): Explain why the above implementation of `cut(int n)` is **not** thread safe. To this end, describe a concrete scenario where race conditions may occur.

Without any kind of concurrency control, `cut(int n)` may interfere with other threads executing operations on the list. For example, suppose a thread t is executing `cut(3)` while a thread u is executing `cut(2)`. If t and u scan the list in parallel, the final outcome depends on which thread redirects `head.next` (in its last statement) before the other. If t does it first, only u's operation has effect, and hence two elements are removed. If u does it first, only t's operation has effect, and hence three elements are removed.

Question 3.2 (2 points): Recall the implementation `CoarseSet` of the thread-safe set data structures seen during the course (also described in Section 9.4 of Herlihy & Shavit's book, where it is

called `CoarseList`). `CoarseSet` uses a variable `lock` of type `Lock` to guard access to the whole data structure. Modify the sequential implementation of `cut(int n)` given above so that it uses `lock` to avoid race conditions.

Since `lock` guards access to the whole list, it is sufficient to acquire `lock` at the beginning of `cut(int n)` and to release it at the end:

```
public void cut(int n) {
  lock.lock();
  try {
    // body of cut(int n) as above
  } finally {
    lock.unlock();
  }
}
```

Question 3.3 (6 points): Recall the implementation `FineSet` of the thread-safe set data structures seen during the course (also described in Section 9.5 of Herlihy & Shavit's book, where it is called `FineList`). `FineSet` uses a variable `lock` of type `Lock` *in each node* to guard access to the node. Modify the sequential implementation of `cut(int n)` given above so that it *locks all nodes* that will be affected by the operation, so that race conditions are avoided, *and releases them* after redirecting `head`.

Acquiring locks incrementally from `head` up to and including `upto` prevents other threads from accessing the portion of the list that is being modified. After removing the nodes, releasing the locks on `head` and `upto` is enough, because the `n` internal nodes have been disconnected and hence they will be garbage collected; thanks to the hand-over-hand locking protocol used by other methods of the class, threads waiting to acquire a lock held by the thread executing `cut` may only be waiting for the `head` node, or after the `upto` node.

```
public void cut(int n) {
  int m = 0;
  Node<T> upto = head;
  upto.lock()                        // lock head
  do {
    upto = upto.next;
    upto.lock();                     // lock next node
    m += 1;
  } while (m <= n && upto != tail)
  head.next = upto;
  head.unlock();                     // unlock head
  upto.unlock();                     // unlock upto
}
```

Question 3.4 (3 points): Explain why an implementation of `cut(int n)` in `FineSet` (as in the answer to the previous question) that locks *only the `head` node* (which is the node that is physically modified) may run into *race conditions*. Illustrate a concrete scenario where race conditions may happen (you can pick a specific value of `n` to help make the scenario concrete).

For example, calling `cut(1)` is equivalent to calling `remove(it)` where `it` is the item in first position in the set. If `cut` acquires only the lock on `head`, this is the same as `remove` only locking `pred`; then, another thread (executing another operation) may be redirecting the first node, which runs into race conditions: the removal by the thread terminating first is "overwritten" by the other thread terminating second.

Exercise 4: Semaphores *(12 points)*

Consider N threads that execute in parallel; N is fixed but can be any positive integer. The threads share an array `a` of size `N`, all of whose elements are integers initially set to 0, and synchronize by means of a shared semaphore instance `sem` – initialized to value `1` (the semaphore's *capacity*):

```
Semaphore sem = new Semaphore(1);  int[] a = new int[N]; // shared variables
```

You can assume threads have integer identifiers 0 through $N - 1$; calling method `threadId()` returns the identifier of the running thread.

Question 4.1 (4 points): Write a **method** `int count(int upto)`, which returns the number of elements of array `a` at indexes between `0` (included) and `upto` (excluded) that are equal to `1`. Method `count(int upto)` must use semaphore `sem` to ensure that there is no interference from other threads modifying the values of `a`.

```
int count(int upto) {
   int result = 0;
   sem.down();
   for (int i = 0; i < upto; i++)
      if (a[i] == 1)
         result += 1;
   sem.up()
   return result;
}
```

Question 4.2 (6 points): Write a **method** `void rankWaitLoop()`, which implements the following behavior. When a thread t_k, with identifier k, calls `rankWaitLoop()`, it is delayed until all threads *with identifiers smaller than* k have executed `rankWaitLoop()`. The solution uses **busy waiting** to implement the described behavior, by synchronizing *only* through `sem` and `a`. The implementation of `rankWaitLoop()` can call method `count(int me)` defined above. We assume that each thread calls `rankWaitLoop()` exactly once.

As soon as thread t_k executes `rankWaitLoop()` it records its "arrival" by setting `a[k]` to `1`; then it keeps on checking whether the threads with smaller ID have arrived yet. The semaphore is used as a lock to regulate access to array `a`.

```
void rankWaitLoop() {
   int me = threadId();
   sem.down();
   a[me] = 1;
```

```
        sem.up();
        int done;
        do {
           done = count(me);
        } while (done < me);
    }
```

Question 4.3 (2 points): Briefly explain the difference between a **binary semaphore** (such as `sem`) and a **lock**. Could you have used a lock instead of a binary semaphore in solving the previous question?

In a binary semaphore, a thread other than the one which performs a `down` can execute a subsequent `up`. In a lock, the locking thread is the only one that can release the lock. The solution to the previous question uses the binary semaphore as a lock, so the additional power of semaphores is not needed.

Exercise 5: Fork/join parallelism *(13 points)*

Fibonacci numbers are the numeric sequence $F_0, F_1, F_2, \ldots$ defined recursively as:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n > 1 \end{aligned}$$

In this exercise, you will evaluate different implementations that compute Fibonacci numbers according to their recursive structure using parallel tasks (fork/join parallelism).

Question 5.1 (4 points): Here is an implementation of the computation of Fibonacci numbers, in a class `Fibonacci` that inherits from `RecursiveTask<Integer>` (part of Java's library for fork/join parallelism). Complete the given implementation by providing the four missing parts (on lines 11, 13, 14, and 18) without changing the order of the statements.

```
 1  public class Fibonacci extends RecursiveTask<Integer> {
 2    private Integer n;    // compute the Fibonacci number F_n
 3
 4    Fibonacci(int n) {
 5      this.n = n;
 6    }
 7
 8    @Override
 9    protected Integer compute() {
10      if (n == 0 || n == 1) {
11         /*  ADD CODE HERE  */
12      }
13      Fibonacci F_n1 =  /*  ADD CODE HERE  */
```

```
14      Fibonacci F_n2 =  /*  ADD CODE HERE  */
15      Integer result = 0;
16      F_n1.fork();
17      result += F_n1.join();
18      /*  ADD CODE HERE  */
19      result += F_n2.join();
20      return result;
21    }
```

- Line 11: `return 1;`
- Line 13: `new Fibonacci(n - 1);`
- Line 14: `new Fibonacci(n - 2);`
- Line 18: `F_n2.fork();`

Question 5.2 (4 points): Write the implementation of an **Erlang** function `fibonacci(N)` that computes F_N by replicating the behavior of the fork/join algorithm given in Java.

```
fibonacci(N) when N == 0; N == 1 ->
   1;
fibonacci(N) ->
   Me = self(),
   spawn(fun () -> Me ! fibonacci(N-1) end),
   Result = receive F_n1 -> F_n1 end,
   spawn(fun () -> Me ! fibonacci(N-2) end),
   receive F_n2 -> Result + F_n2 end.
```

Question 5.3 (2 points): How many tasks (corresponding to instances of `Fibonacci`) *can actually run in parallel* (as opposed to being blocked in a waiting operation) in the Java solution – assuming an unlimited amount of physical cores and threads that can execute tasks? Justify your answer.

Each task forks a subtask (`F_n1.fork()`) and then immediately waits for it to complete (`F_n1.join()`), before forking another subtask (`F_n2.fork()`) and then immediately waiting for it to complete (`F_n2.join()`). Therefore, at any given time there is just *one task that is running* – the one corresponding to the innermost recursive call – while all other tasks are recursively waiting for it to complete.

Question 5.4 (3 points): Explain how to modify the implementation of `Fibonacci` in Java in a way that it *increases the number of tasks that can run in parallel* (analyzed in the answer to the previous question). Note: the modification does not need to maximize the amount of parallelism; it just has to increase it – that is, it has to be somewhat better than the given implementation.

Switching lines 17 and 18 is enough to introduce some real parallelism: both tasks `F_n1` and `F_n2` can execute in parallel while the forking task waits for them to complete.