Chalmers | Göteborgs Universitet
Carlo A. Furia, Computer Science and Engineering

Concurrent Programming TDA383/DIT390

Saturday, 18 March 2017, 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 *(15 points)*

Consider the following concurrent program P, where two threads t and u execute in parallel and access a single shared integer variable `v`.

```
                      int v = 1;
-----------------------------------------------------------
           thread t            |          thread u
                               |
1  v = 1;                      |  int i = v;            2
                               |  v = 2*i;              3
```

Question 1.1 (1 point): Does program P have **race conditions**? Justify your answer.

A following question indicates that program P may terminate with different final values of variable `v`. 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 the shared variable `v` in every statement of their code, their critical sections correspond to the whole code they each execute.

Question 1.3 (2 points): List all **data races** that exist in program P.

There are two data races, corresponding to the instruction pairs on lines $(1, 2)$ and $(1, 3)$.

Question 1.4 (4 points): Write two complete **traces** corresponding to two possible executions of program P such that the *final* value of `v` is different in the two traces. Each trace must be a sequence of program states, where each state indicates: the value of `v`, 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 variable `i`.

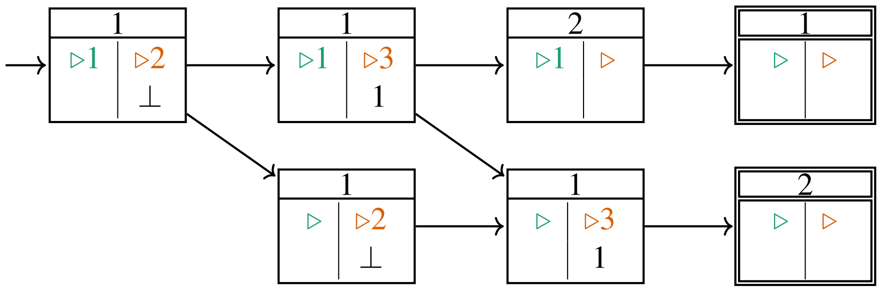
A trace where the final value of `v` is 2:

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 1	pc_u : 2 i: ⊥	v: 1
2	done	pc_u : 2 i: ⊥	v: 1
3	done	pc_u : 3 i: 1	v: 1
4	done	done	v: 2

A trace where the final value of `v` is 1:

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 1	pc_u : 2 i: ⊥	v: 1
2	pc_t : 1	pc_u : 3 i: 1	v: 1
3	pc_t : 1	done	v: 2
4	done	done	v: 1

Question 1.5 (6 points): Build a complete **state/transition diagram** modeling all possible executions of program P. Each state of the diagram should indicate: the value of `v`, 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 variable `i`. Also mark the initial and final states of the diagram.



Exercise 2: Semaphores *(10 points)*

Consider some threads that execute in parallel; the threads share an integer variable `x`, whose value is initially 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 x = 0; // shared variables
```

Question 2.1 (3 points): Implement a **method** `void mayDecrement(int n)`, which **atomically** subtracts `n` from `x` *if* `x` is greater than or equal to `n`; otherwise, it terminates without modifying `x`. Your solution must only use `sem` and `x` for synchronization.

```
void mayDecrement(int n) {
  sem.down()  // get exclusive access
  if (n <= x) x = x - n;
  sem.up();   // release exclusive access
}
```

Question 2.2 (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.

Question 2.3 (2 points): Briefly explain the difference between **strong** and **weak** semaphores.

Question 2.4 (3 points): Write the code of a generic thread t_k in a way that **starvation** may occur (assuming all threads execute the same code) if `sem` is a **weak** semaphore, but it could not occur if `sem` is a strong semaphore – whereas *deadlocks* never occur regardless of the semaphore's type.

Suppose all threads repeatedly decrement and increment back the semaphore:

```
while (true) {
  sem.down();
  sem.up();
}
```

If the semaphore is weak, it is possible that a thread blocked trying to execute `down()` will never be selected to acquire the semaphore. In contrast, if the semaphore is strong, any thread blocked trying to execute `down()` will acquire the semaphore as soon as all other threads that have been waiting longer have acquired (and released) it.

Exercise 3: Monitors – synchronization *(14 points)*

In an office environment, there are an unspecified number of desks, and two kinds of workers: *clerks* and *cleaners*. Workers behave as follows:

Clerks may arrive at the office at any time. In order to get to work, they wait until they have exclusive access to an available desk that has been cleaned. After working for an unspecified amount of time, they leave the desk, which needs cleaning.

Cleaners work on one desk at a time. Each cleaner waits until a clerk leaves a desk, at which point the cleaner starts cleaning the desk, and then signals when cleaning is completed.

We model the office environment in Java pseudo-code using **monitor classes**:

- Desks corresponds to instances of a class `Desk`, whose details are irrelevant.
- The office is an instance of type `Office` with interface:

```
interface Office
{
  Desk arrive();             // get an available, clean desk
  void leave(Desk desk);     // leave desk ‘desk’
  Desk service();            // get a desk to clean
  void cleaned(Desk desk);   // desk ‘desk’ is now clean
}
```

- Clerks and cleaners correspond to threads sharing an instance of `Office` and calling its public methods continuously as follows:

```
Office office = new OpenOffice();
```

clerk$_n$	cleaner$_m$

`while (true) {`
`  Desk desk = office.arrive();`
`  // work at desk`
`  office.leave(desk);`
`}`
	`while (true) {`
	`  Desk desk = office.service();`
	`  // clean desk`
	`  office.cleaned(desk);`
	`}`

The goal of this exercise is to provide a *monitor class* `OpenOffice` with interface `Office`, whose behavior follows the above description. You should use the Java pseudo-code for monitors that we used in class:

```
monitor class OpenOffice implements Office
```

and assume a *signal and continue* signaling discipline.

Question 3.1 (4 points): Declare the **condition variables** and other private **attributes** that you need in `OpenOffice`. For each of them, concisely indicate in a comment what it represents.

```
Set<Desk> cleanDesks, dirtyDesks;   // sets of clean, dirty desks
Condition clean = new Condition();  // condition variable: a clean desk is available
Condition dirty = new Condition();  // condition variable: a dirty desk is available
```

Question 3.2 (4 points): Write the implementation of **methods** `arrive` and `leave`.

```
Desk arrive() {
  while (cleanDesks.size() == 0)  // wait until a clean desk is available
    clean.wait();
  return cleanDesks.get();        // return it
}

void leave(Desk desk) {
  dirtyDesks.put(desk);          // give back a desk, which is now dirty
  dirty.signal();                // signal that a dirty desk is available
}
```

Question 3.3 (4 points): Write the implementation of **methods** `service` and `cleaned`.

```
Desk service() {
  while(dirtyDesks.size() == 0)  // wait until a dirty desk is available
    dirty.wait();
  return dirtyDesks.get();       // return it
}

void cleaned(Desk desk) {
  cleanDesks.put(desk);          // give back a desk, which is now clean
  clean.signal();                // signal that a clean desk is available
}
```

Question 3.4 (2 points): Would you have to change your implementation if we used monitors following the *signal and wait* signaling discipline? Briefly justify your answer.

No changes to the given solution are needed: signaling is only done as last statement in a method, and the **`while`** loops checking for a condition while waiting also work under *signal and wait*.

Exercise 4: Monitors – signaling disciplines *(14 points)*

Consider the following monitor class MC:

```
monitor class MC
{
  private int x = 1;
  private Condition c = new Condition();
  private Condition d = new Condition();

  public void m() {
    if (x == 2) { c.wait(); }
    x = x + 1;
    d.signal();
  }

  public void n() {
    if (x == 0) { c.signal(); d.wait(); }
    x = x - 1;
  }
}
```

Note that the methods are monitor methods, so any thread running them implicitly executes in mutual exclusion (acquires a lock on the monitor at the beginning, and releases it at the end). In all questions, we assume that an arbitrary number of threads share one instance of MC, and may call its methods m() and n() at any time.

Question 4.1 (2 points): Explain why **deadlock** is **not** possible – under the assumption that threads continuously call both m() and n().

Deadlock occurs if no threads make progress. Threads calling m() and n() can only block when calling wait() on one of the condition variables. But the conditions under which they execute wait are *mutually exclusive*: if, say, a thread blocks in n(), then x == 0, and thus x != 2, and thus any thread executing m() does not block.

Question 4.2 (4 points): Assume that the system only includes threads t, u, and v, and that the monitor follows a *signal and wait* signaling discipline. Describe a scenario where **starvation** of t trying to execute m() occurs; the answer should describe a sequence of calls executed by t, u, and v such that t starves from some point on.

Suppose t executes m() when x == 2, so that it blocks in the queue c.blocked of blocked threads on condition c. Then, u and v strictly alternate respectively calling n() and m(). Since they strictly alternate, x's values alternates between 1 (after u calls n()) and 2 (after v calls m()), so that c.signal() – which would unblock t – is never called.

Question 4.3 (4 points): Assume that the monitor follows a *signal and wait* signaling discipline. Is it possible that a thread that blocks on `c.wait()` – while executing `m()` – is unblocked, and **resumes** execution, in a **state** such that `x == 2`? If it is possible, describe a scenario where this occurs; if it is not possible, explain why it is not possible.

This is not possible. Suppose a thread t is blocked in `c.blocked`'s queue, since it called `c.wait()`. When another thread calls `c.signal()`, t **immediately** resumes execution according to the *signal and wait* signaling discipline; since `c.signal()` is called when `x == 0`, t will resume execution in a state with `x == 0 != 2`.

Question 4.4 (4 points): Answer the previous question but now assuming that the monitor follows a *signal and continue* signaling discipline: Is it possible that a thread that blocks on `c.wait()` while executing `m()` is unblocked, and **resumes** execution, in a **state** such that `x == 2`? If it is possible, describe a scenario where this occurs; if it is not possible, explain why it is not possible.

This is possible. Suppose a thread t is blocked in `c.blocked`'s queue, since it called `c.wait()` with `x == 2`. While t is blocked, other threads queue for entry to the monitor; first, two calls to `n()` execute without blocking; then, one more call to `n()` queues for entry, followed by two threads calling `m()`. The additional call to `n()` finds `x == 0`, and blocks after signaling `c.signal()` to unblock t. However, under the *signal and continue* signaling discipline, t is moved to the back of the entry queue after the two threads waiting to call `m()`. Thus, these two threads will get to execute before t, incrementing `x` all the way back up to 2. When t finally executes, it will thus find `x == 2`.

Exercise 5: Erlang – servers *(15 points)*

An integer number P is a *prime* if it is only exactly divisible by 1 and P itself. The *factorization* of a positive integer N is a collection of primes, called the factors of N, whose product is exactly N.

In this exercise, we build an Erlang program in the style of **servers**, which computes the factorization of integers.

Question 5.1 (4 points): Define an Erlang **function** `prime(X)`, which takes an integer `X`, and returns `true` if `X` is prime, and `false` otherwise. You can assume that `X` is always a positive integer. *Hint*: it may be convenient to define a helper function, which is called by `prime`'s implementation.

Recall that, in Erlang, the infix operator `rem` computes the remainder of the integer division of its arguments (similarly to `%` in Java); for example: `10 rem 4` is `2`, `10 rem 3` is `1`, and `10 rem 5` is `0`.

```
 % is 'X' prime?
prime(1) -> true;
prime(X) -> not composite(X, X-1).

 % is 'X' divisible by any integer between 2 and Y?
composite(_, 1) -> false;
composite(X, Y) -> (X rem Y == 0) or composite(X, Y-1).
```

Question 5.2 (6 points): Define a **server event loop** function `factors(N, State, Recipient)` that works as follows. It accepts messages in the form `{fac, P}` where `P` is a positive integer greater than 1, and checks whether `P` is prime (by calling function `prime` defined in the previous question). If `P` is not prime, the message is ignored; if `P` is prime and `P` is a factor of `N`, the server updates its state to store `P` as an additional factor, and goes back to accepting messages suggesting *other* factors. When the factorization of `N` is complete, the server first sends to the process with PID `Recipient` the factorization as a list of factors (in any order), and then it terminates.

```
factors(1, State, Recipient) -> Recipient ! State;
factors(N, State, Recipient) ->
  receive   % N rem P == N if P >= N
    {fac, P} -> case prime(P) and (N rem P == 0) of
      false -> factors(N, State, Recipient);
      true  -> factors(N div P, [P|State], Recipient)
     end
  end.
```

Question 5.3 (5 points): Define a **function** `test(N, TryFactors)` that tests `factors` as follows. The function `test` spawns a process running `factors(N, State, Recipient)` with `Recipient` equal to the PID of the process running `test`, and `State` a suitable initial state of the server; it then sends, to the spawned server process, several messages `{fac, F}` – precisely, one for every element `F` of the list `TryFactors`; finally, it waits for the spawned process to send back a list, and returns `true` if the sent list has the same elements as `TryFactors`. To compare the sent list and `TryFactors` you can assume a function `same_elements(L1, L2)` that compares two lists is available. Note that `test(N, TryFactors)` may not terminate if `TryFactors` is not the factorization of `N` – this is acceptable behavior.

```
test(N, TryFactors) ->
    Me = self(),
    Server = spawn(fun ()-> factors(N, [], Me) end),
    [Server ! {fac, F} || F <- TryFactors],
    receive ComputedFactors -> same_elements(ComputerFactors, TryFactors) end.
```