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

Principles of Concurrent Programming TDA384/DIT391

Wednesday, 22 August 2018, 14:00–18:00

(including example solutions)

Examiner: K. V. S. Prasad (prasad@chalmers.se)
(Exam prepared by Carlo A. Furia based on the course given by him in SP3)

Exercise 1: Concurrency properties *(18 points)*

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

```
                        int x = 3;
```

thread t		thread u	
`int n; // t's local variable`		`int m; // u's local variable`	
1 `n = x;`		`m = x;`	3
2 `x = 1 * n;`		`x = 1 * m;`	4

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

The final value of `x` is always `3` regardless of the order in which t and u execute. Since the final value of P's computation does not depend on the interleaving of concurrent threads, program P has no race conditions.

Question 1.2 (1 point): 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.

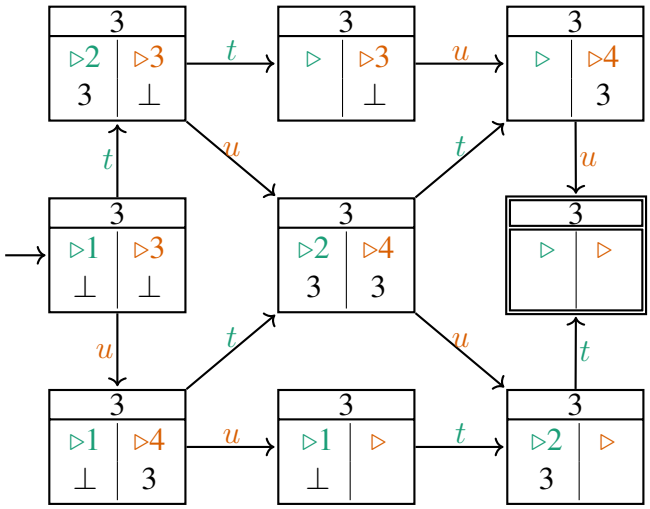
Question 1.3 (3 points): List all **data races** that exist in program P.

There are three data races, corresponding to the instruction pairs on lines $(1, 4)$, $(2, 3)$, and $(2, 4)$. Note that there are data races even if they are such that they do not determine race conditions.

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

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t: 1 n: $\perp$	pc_u: 3 m: $\perp$	x: 3
2	pc_t: 2 n: 3	pc_u: 3 m: $\perp$	x: 3
3	pc_t: 2 n: 3	pc_u: 4 m: 3	x: 3
4	done	pc_u: 4 m: 3	x: 3
5	done	done	x: 3

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`, 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 *final* states of the diagram. (The initial state of the diagram is 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 only *one* final state, which means P has no race conditions.

Exercise 2: Erlang – servers *(17 points)*

In this exercise, we build an Erlang program in the style of **servers**, which provides the functionality of a *distributed event counter*.

An event counter is a server process `Counter` that counts events up to a given *goal* `Goal` (a number, representing the number of events that must be counted); `Goal` is fixed when the event counter is initialized and never changed afterwards.

Any process can notify an event occurrence by sending a message `{event, Pid}` to the event counter – where `Pid` is the sender's PID (process identifier). The event counter keeps tracks of

how many event notification it has received, and of the PIDs of all processes that sent an event notification.

As soon as the number of notified events reaches the goal, the event counter sends a message `done` to all processes that sent event notifications. If the same process sent multiple event notifications, it should only receive message `done` once. After sending `done` messages, the server resets its state so that it can count more events up to the goal.

The event counter also accepts a message `{reset, Creator}` but only if `Creator` corresponds to the PID of the process that initialized the event counter (that is, the process that spawned the server). Under this condition, this message resets the counter of events to zero without notifying any process and forgetting all stored PIDs; even in case of reset, the goal and the creator's PID do not change.

Question 2.1 (2 points): Describe the **arguments** of a **server event loop** function `counter`, 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.

```
counter(Done, Goal, Participants, Creator)
```

- `Done`: numeric type, number of event notifications received so far;
- `Goal`: numeric type, goal (events to be counted);
- `Participants`: list type, PIDs of all processes that sent an event notification;
- `Creator`: PID type, PID of process that spawned the server.

Question 2.2 (8 points): Define the **server event loop** function `counter`, which implements the server behavior described above and uses the arguments given in the answer to the previous question.

```
counter(Done, Goal, Participants, Creator) when Done == Goal -> % goal reached
   [P ! done || P <- Participants],      % notify all participants
   counter(0, Goal, [], Creator);        % reset server state
counter(Done, Goal, Participants, Creator) ->              % goal not reached
   receive
      {event, From} ->         % receive event notification
         Registered = lists:member(From, Participants),
         if  % add PID of notifier only if it is not already in Participants
           Registered =:= true -> counter(Done+1, Goal, Participants, Creator);
           true -> counter(Done+1, Goal, [From | Participants], Creator)
         end;
      {reset, Creator} ->      % receive reset (from Creator only)
         counter(0, Goal, [], Creator)   % reset server state
   end.
```

Question 2.3 (3 points): Define function `start(Goal)`, which spawns a process running the server (corresponding to an event counter with goal `Goal`), and returns to the caller the PID of the spawned server process. Remember to keep track of the PID of the process executing `start` (the "creator" of the server).

```
start(Goal) ->
   Creator = self(),
   spawn(fun () -> counter(0, Goal, [], Creator) end).
```

Question 2.4 (2 points): Define function `event_async(Counter)`, which interacts with the event counter server with PID `Counter` to notify an event without blocking (that is, it sends a message and returns immediately).

```
event_async(Counter) ->
   Me = self(),
   Counter ! {event, Me}.
```

Question 2.5 (2 points): Define function `event_sync(Counter)`, which interacts with the event counter server with PID (process identifier) `Counter` to notify an event, and then blocks until it receives message `done` (after which it returns).

```
event_sync(Counter) ->
   event_async(Counter),
   receive done -> ok end.
```

Exercise 3: Semaphores *(12 points)*

Consider two threads t and u that execute in parallel. The threads share two (general counting) semaphore variables `x` and `y`, both initialized to value `0` (the semaphores' *capacity*).

```
Semaphore x = new Semaphore(0);  Semaphore y = new Semaphore(0); // shared variables
```

You can assume that no other threads have access to the same semaphores, and that a method `getPid()` returns the string `"t"` when called by t, and `"u"` when called by u.

Question 3.1 (4 points): Write a **method** `void go()`, which behaves differently according to whether it is called by t or by u. If t calls `go()`, it blocks until u has also called `go()`; then, `go()` returns and t is allowed to continue. If u calls `go()`, it does not block and simply goes through. Method `go()` must use one or both semaphores `x` and `y` to achieve this synchronizing behavior of threads t and u. You can assume that t and u each call `go()` exactly once (in any order).

This is similar to an asymmetric barrier, where t waits for u but u does not wait. Only one semaphore is needed to achieve this synchronization.

```
void go() {
   if (getPid() == "t")
      x.down();   // t decrements x after u has incremented it
   else
      x.up();     // u increments x unconditionally
}
```

Question 3.2 (6 points): Write a **method** `void guess(int require, int ensure)`, which synchronizes t and u as follows. Threads t and u call `guess` exactly once with any value for arguments `require` and `ensure`; suppose t calls `guess(rt, et)` and u calls `guess(ru, eu)`:

- Thread t waits until u has also called `guess`. If `rt` $\leq$ `eu`, t is then allowed to continue (that is, t's call to `guess` returns); otherwise, `rt` $>$ `eu` and t blocks forever executing `guess`.
- Thread u waits until t has also called `guess`. If `ru` $\leq$ `et`, u is then allowed to continue (that is, u's call to `guess` returns); otherwise, `ru` $>$ `et` and u blocks forever executing `guess`.

For example, if t calls `guess(2, 5)` and u calls `guess(2, 2)`, both t's and u's calls eventually return; if t calls `guess(2, 3)` and u calls `guess(4, 8)`, t's call eventually returns but u never does. *Hint:* you can use semaphore `x` to count t's permits and semaphore `y` to count u's permits.

This is similar to a barrier with the additional requirement that each thread must ensure enough semaphore permits that the other can continue. We use `x` to count the permits that t has available, and `y` to count those that u has.

```
void guess(int require, int ensure) {
   Semaphore req, ens;
   if (getPid() == "t") {
      req = x;   // t's permits are counted by x
      ens = y;   // t grants permits to u using y
   } else {
      req = y;   // u's permits are counted by y
      ens = x;   // u grants permits to t using x
    }
    for (int i = 0; i < ensure; i++)
       ens.up();     // grant ensure permits
    for (int i = 0; i < require; i++)
       req.down();   // count down at least require permits
}
```

Question 3.3 (2 points): Briefly explain the difference between a **binary semaphore** and a **general (counting) semaphore**. Does `x` or `y` behave as a binary semaphore in any of the previous questions 3.1 or 3.2? Justify your answer.

A binary semaphore is a semaphore used in a way that it always stores the values 0 or 1. `x` (and `y` trivially) is a semaphore in 3.1, but they are general semaphores in 3.2.

Exercise 4: Synchronization with locks *(10 points)*

Two threads t and u execute in parallel and access two integer shared variables `v1` and `v2`. To control access to the variables, there are two locks `g1` and `g2`: whenever a thread wants to read or write `v1` it must first acquire lock `g1`; and whenever it wants to read or write `v2` it must first acquire lock `g2`.

```
// shared variables
Lock g1 = new Lock();
int v1 = 0;
Lock g2 = new Lock();
int v2 = 0;
```

In this exercise you have to write variants of the code executed by t and u; in **all** variants, each threads increments both `v1` and `v2` by one after accessing the respective lock. The variants differ in the concurrency properties they guarantee. Each variant may only use `g1`, `v1`, `g2`, and `v2`; but you are free to introduce new thread-local variables.

Question 4.1 (3 points): Write code executed by t and by u such that there are neither race conditions nor deadlocks.

If both threads acquire and release locks in the same order there are no deadlocks; then, using the locks to make the increments atomic avoids race conditions.

```
Lock g1 = new Lock(); int v1 = 0; Lock g2 = new Lock(); int v2 = 0;
```

thread t

```
g1.lock();
g2.lock();
v1 = v1 + 1;
v2 = v2 + 1;
g1.unlock();
g2.unlock();
```

thread u

```
// identical code as t
```

Question 4.2 (3 points): Write code executed by t and by u such that there are no race conditions, but deadlocks may occur.

If t acquires `g1` first while u acquires `g2` first, a deadlock may occur.

```
Lock g1 = new Lock(); int v1 = 0; Lock g2 = new Lock(); int v2 = 0;
```

thread t	thread u
`g1.lock();`	`g2.lock();`
`g2.lock();`	`g1.lock();`
`v1 = v1 + 1;`	`v1 = v1 + 1;`
`v2 = v2 + 1;`	`v2 = v2 + 1;`
`g1.unlock();`	`g1.unlock();`
`g2.unlock();`	`g2.unlock();`

Question 4.3 (3 points): Write code executed by t and by u such that there are no deadlocks, but race conditions may occur.

If we allow interleaving between when a variable is read and when it is written to, race conditions may occur. Since both threads acquire locks in the same order a deadlock is not possible.

```
Lock g1 = new Lock(); int v1 = 0; Lock g2 = new Lock(); int v2 = 0;
```

thread t	thread u
`g1.lock();`	
`int tmp = v1;`	
`g1.unlock();`	
`g1.lock();`	
`v1 = tmp + 1;`	`// identical code as t`
`g1.unlock();`	
`g2.lock();`	
`v2 = v2 + 1;`	
`g2.unlock();`	

Question 4.4 (1 point): Can **starvation** occur in the code you provided in any of the previous questions? Justify your answer.

A deadlock is a form of starvation, therefore the code that may deadlock may also starve. Starvation without deadlock is not possible in this example since the code of each thread terminates in finite time if there is no deadlock.

Exercise 5: Concurrent data structures *(13 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 an operation on a simple data structure in Java, analyzing whether they are thread safe.

The data structure simply stores two integers `X` and `Y` in a way that it is possible to increment both variables at once. Class `Pair` is a *sequential implementation* of the data structure:

```
class Pair {
  private int X;
  private int Y;
  public int getX() { return X; }  // current value of X
  public int getY() { return Y; }  // current value of Y
  public void incXY() {            // increment X and Y
     X = getX() + 1;
     Y = getY() + 1;
  }
}
```

Question 5.1 (3 points): Explain why the above implementation of `Pair` is **not** thread safe:

- Describe a concrete scenario where race conditions may occur.
- List all operations (that is, methods) that are not thread safe.

Method `incXY()` updates `X` and `Y` non-atomically, and hence may run into race conditions. For example, if two threads call `incXY()` they may interleave so that some increments are lost (as in the concurrent counter example). The getter methods instead are atomic and hence thread safe.

Question 5.2 (3 points): Write the implementation of a class `LockedPair`, which provides the same operations as `Pair` but is *thread-safe*. To this end `LockedPair` introduces a single variable `lock` to guard access to the data structure. (Your implementation of `LockedPair` may inherit from `Pair` or directly modify its implementation.)

Variable `lock` is a private attribute attached to a `Lock` object. Only `incXY` has to be redefined since it is the only state-modifying operation.

```
class LockedPair extends Pair {
   private Lock lock = new Lock();

   @Override
   public void incXY() {
      lock.lock();
      try {
         super.incXY();
      } finally {
         lock.unlock();
      }
   }
}
```

Question 5.3 (7 points): Recall that objects of class `AtomicInteger` provide integer that can be modified *atomically* using the compare-and-set operation. If `v` is a reference to an object of class `AtomicInteger`, `v.compareAndSet(old_v, new_v)` atomically updates `v` to `new_v` and returns `true` if `v` stores value `old_v`; otherwise, it does not change `v` and returns `false`.

Using class `AtomicInteger`, write the implementation of a class `TASPair`, which provides the same operations as `Pair` but is *thread-safe without using any locks*. To this end, assume that `X` and `Y` have

type `AtomicInteger`. (Your implementation of `LockedPair` may inherit from `Pair` or directly modify its implementation.)

The basic idea is to update both `X` and `Y` in separate loops. The loops terminate if the variable have not been changed since the latest iteration; otherwise, it discards any changes and iterates anew.

```
class TASPair extends Pair {
   private AtomicInteger X = new AtomicInteger();
   private AtomicInteger Y = new AtomicInteger();
   @Override public int getX() { return X.get(); }
   @Override public int getY() { return Y.get(); }
   @Override
   public void incXY() {
      do {
         int oldX = getX();
         if (X.compareAndSet(oldX, oldX + 1))
             break;
      } while (true);

      do {
        int oldY = getY();
        if(Y.compareAndSet(oldY, oldY + 1))
            break;
      } while(true);
   }
}
```