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

Concurrent Programming TDA383/DIT390

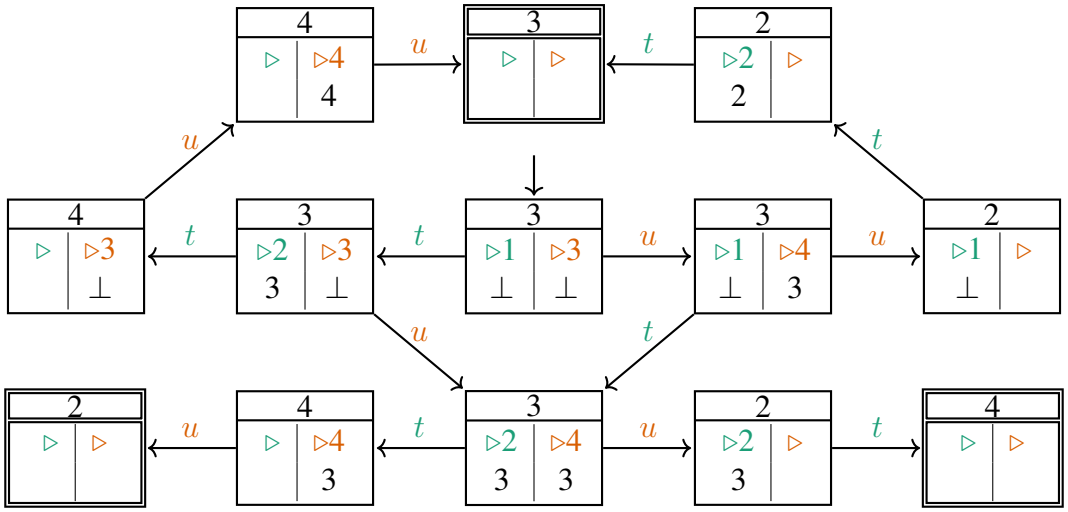
Monday, 21 August 2017, 14:00–18:00

(including example solutions)

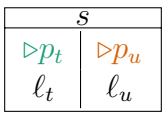
Teacher/examiner: Carlo A. Furia (`furia@chalmers.se` – 0317721675)
(the examiner will visit the exam rooms twice: around 15:00 and around 17:00)

Exercise 1: Concurrency properties *(15 points)*

Consider a **concurrent program** P, where two threads t and u execute in parallel and access a single shared integer variable `v`; each thread also uses one local variable – thread t uses a local variable `j`, and thread u uses a local variable `k`. P's behavior is completely captured by the following **state/transition diagram**.



In the diagram, a state of the form:



represents a *program state* where:

- the value of `v` is s;
- thread t is ready to execute the statement on line p_t with variable `j` equal to ℓ_t;

- thread u is ready to execute the statement on line p_u with variable k equal to ℓ_u.

Recall that the initial state is the one with a short incoming arrow (in the middle of the picture), and the final states are the ones with double edges.

In this exercise, you have to analyze properties of program P based on its state/transition diagram.

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

Since there are multiple reachable final states with different final values of variable v (values 2, 3, and 4), 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): Is program P **free from deadlock**? Justify your answer.

Since every non-final state has at least one outgoing transition, the program can always make progress, and thus it is free from deadlock.

Question 1.3 (2 points): Is program P **free from starvation**? Justify your answer.

Since every thread is eventually allowed to execute (accessing shared variable v), no thread can starve, and thus the program is free from starvation.

Question 1.4 (5 points): Write the complete code of program P, whose behavior is given by the state/transition diagram, by copying the following schema and filling in the ellipses (i.e., "...").

```
int v = ... // initial value of v
```

thread t		thread u	
`int j = ...`	1	`int k = ...`	3
`...`	2	`...`	4

```
int v = 3; // initial value of v
```

thread t		thread u	
`int j = v;`	1	`int k = v;`	3
`v = j + 1;`	2	`v = k - 1;`	4

Question 1.5 (3 points): List all **data races** that exist in program P.

There are three data races, corresponding to the statement pairs on lines $(1, 4)$, $(2, 3)$, and $(2, 4)$.

Question 1.6 (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 the shared variable v in every statement of their code, their critical sections correspond to the whole code they each execute.

Exercise 2: Semaphores *(10 points)*

Consider two threads t and u that execute in parallel. The threads synchronize by means of a shared (binary) **semaphore** instance `sem` – initialized to value `0` (the semaphore's *capacity*):

```
Semaphore sem = new Semaphore(0); // shared semaphore instance
```

Question 2.1 (5 points): Implement a **method** `void sync()`, which behaves as follows. When thread t calls `sync()`, it does not wait and just continues execution; when thread u calls `sync()`, it blocks *until t has called `sync()`*, and then continues.

You can assume that no threads other than t and u call `sync()` or access the semaphore, and that both threads call `sync()` exactly once (at any time). Your solution can call a method `id()`, which returns the identifier of calling thread t or u. Your solution must only use `sem` for synchronization.

```
void sync() {
  if (id() == "t")
    sem.up();
  else
    sem.down();
}
```

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 to solve the previous question?

In a binary semaphore, a thread other than the one that performs a `down` can execute a subsequent `up`. In a lock, the locking thread is the only one that can release the lock. In the given solution, it is crucial that only t can "release" the lock, whereas u must "acquire" it. Thus, a lock would not work to implement this solution.

Question 2.3 (3 points): Now, suppose that both threads t and u call `sync()` continuously in an infinite loop. Again, no threads other than t and u call `sync()` or access the semaphore.

a. Can **starvation** of u trying to execute `sync()` occur if `sem` is a **strong** semaphore?

b. Can **starvation** of u trying to execute `sync()` occur if `sem` is a **weak** semaphore?

c. Can **starvation** of t trying to execute `sync()` occur if `sem` is a **weak** semaphore?

Justify your answers.

a. With a strong semaphore, u will have to wait at most until t calls `sync()`; thus u cannot starve (we are assuming that the loop is infinite, so t does not terminate).

b. Even with a weak semaphore, u will have to wait at most until t calls `sync()`: since u is the only thread executing `down()`, no other thread can "overtake" u during its wait to execute `down()`.

c. Thread t only executes `up()`, which does not block. Thus t cannot starve.

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

It's the birthday party of Max. Guests to the party can arrive at any time; after they have arrived, they cannot leave unless Max is available at the party to greet them goodbye. Max can also enter and exit the party at any time.

We model this behavior in Java pseudo-code using a **monitor class**:

- Guests and Max correspond to concurrent thread – one thread per person.
- Synchronization occurs through an instance of *monitor class* `Party`

  ```
  monitor class Party implements IParty
  ```

  with interface:

  ```
  interface IParty
  {
    void enterMax();          // Max is available at the party
    void exitMax();           // Max momentarily leaves the party

    void arrive(int guest);    // a 'guest' arrives at the party
    void leave(int guest);     // a 'guest' leaves the party
  }
  ```

- Max and the guests share an instance of class `Party`; Max continuously calls methods `enterMax()` and `exitMax()`, and the guests continuously call `arrive()` and `leave()`:

  ```
  IParty party = new Party();
  ```

  max

  ```
  while (true) {
    party.enterMax();
    // stay some time
    party.exitMax();
    // leave some time
  }
  ```

  guest_n

  ```
  while (true) {
    // arrive at the party
    party.arrive(n); // n is the guest's id
    // leave the party when possible
    party.leave(n);
  }
  ```

- Methods `enterMax`, `exitMax`, and `arrive` are non-blocking; method `leave` is blocking if Max is currently *not* at the party, and terminates as soon as Max enters the party again.

The goal of this exercise is to provide an implementation of *monitor class* `Party` with interface `IParty`, whose behavior follows the above description. You should use the Java pseudo-code for monitors that we used in class, or any similar notation that describes monitors and whose semantics is clear. Assume a *signal and continue* signaling discipline.

Question 3.1 (3 points): Declare the **condition variables** and other private **attributes** that you need in class `Party`. For each of them, concisely indicate in a comment what it represents.

```
int leaving;                          // number of guests at the party waiting to leave
boolean maxIn;                        // is Max at the party?
Condition canLeave = new Condition(); // condition variable: can guests leave?
```

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

```
void enterMax() {
  maxIn = true;
  for (int k = 0; k < leaving; k++)
    canLeave.signal();
}

void exitMax() {
  maxIn = false;
}
```

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

```
void arrive(int guest) {
  // nothing to do
}

void leave(int guest) {
  leaving += 1;
  while (!maxIn)
    canLeave.wait();
  leaving -= 1;
}
```

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

The `while` loops checking for a condition while waiting also work under *signal and wait*, since Max sets `maxIn` to `true` *before* signaling any thread. However, there may be a problem with the integer `leaving` being changed by the signaled thread while Max loops on its elements, which may result in skipping signaling some blocked threads. A simple solution is to get a copy of `leaving` local to thread Max in method `enterMax` to avoid the interference.

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

Consider the following monitor class `Counter`:

```
monitor class Counter
{
  private int v = 0;
  private Condition i = new Condition();
  private Condition d = new Condition();

  public void inc() {
    if (v >= 3)
      i.wait();
    v = v + 1;
    d.signal();
  }

  public void dec() {
    if (v <= -3)
      d.wait();
    v = v - 1;
    i.signal();
  }
}
```

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 `Counter`, and may call its methods `inc()` and `dec()` at any time. We also assume spurious wakeups cannot occur.

Question 4.1 (2 points): Under a *signal and wait* signaling discipline, what values can variable `v` take at any time during the execution? Justify your answer.

Variable `v` is initialized to `0`. Then, method `inc()` increments `v` by one only when `v < 3`; and method `dec()` decrements `v` by one only when `v > -3`. Thus `v` can take any integer value between `-3` and `3` included.

Question 4.2 (3 points): Mention one value that variable `v` **can** take under a *signal and continue* signaling discipline, but which it **cannot** take under a *signal and wait* signaling discipline. Describe a concrete execution scenario, under signal and continue, where `v` takes that value; the scenario should describe a sequence of calls executed by a suitable number of threads such that `v` eventually is set to that value.

The invariant `-3 <= v <= 3` does not hold under a signal and continue discipline; in particular `v` may take value `4`. For example, suppose `v == 3` and a thread t executes `inc()`; t blocks waiting on `i`. While t is blocked, two threads d and j queue for entering the monitor. Thread d executes `dec()`, which sets `v` to `2` and unblocks t; under signal and continue, t is moved to the back of the

entry queue, behind j. Thread j executes `inc()` next, which sets `v` to `3` again. It is then t's turn to execute, but t does not check `v`'s value again and just sets it to `4`.

Question 4.3 (6 points): Write a new implementation of methods `inc()` and `dec()`, without changing the monitor's variables, such that the values that variable `v` can take in your new implementation under a *signal and continue* signaling discipline are the same as the values variable `v` can take in the implementation given above under a signal and wait signaling discipline. In other words, your implementation under signal and continue should enforce the same invariant on `v` as the given implementation does under signal and wait.

```
public void inc() {
  while (v >= 3)
    i.wait();
  v = v + 1;
  d.signal();
}

public void dec() {
  while (v <= -3)
    d.wait();
  v = v - 1;
  i.signal();
}
```

Question 4.4 (3 points): Under the *signal and continue* signaling discipline, can starvation of a thread occur in the implementation you wrote as an answer to the previous question? If it can occur, describe a scenario where starvation does occur; if it cannot occur, explain what prevents starvation from happening.

Starvation can occur: suppose a thread t executes `inc()` when `v == 3`, so that it blocks in the queue `i.blocked` of blocked threads on condition `i`. Then, two other threads d and j strictly alternate respectively calling `dec()` and `inc()` in a way that j always queues up right after d for access to the monitor. As a result, the value of `v` alternates between `3` and `2`; whenever t is unblocked (by d signaling), it executes after j, and thus finds `v == 3` and blocks again. Since t can never complete its call to `inc()`, it starves.

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

In this exercise, we build an Erlang program in the style of **servers**, which sends acknowledges to requests *in batches*. The server accepts messages in the form `{msg, Pid}` where `Pid` is the pid of some sending process; messages of any other form are ignored. For every received message `{msg, Pid}`, the server stores in its internal state the value of `Pid`. As soon as `N` such requests are received – where `N` is an argument of `batches` that is given when the server is initialized, and is not changed afterward – the server sends a message consisting of the atom `ack` to every pid that has been stored in its state; then, the server resets its state so that it can receive a new batch of `N` messages.

Question 5.1 (2 points): Write the *signature* of a **server event loop** function `batches`, which implements the server behavior described above. Precisely, list the **arguments** function `batches` needs to store the state, in addition to its argument `N` representing the number of messages that are batched. Describe the type of each argument, and what each argument represents.

We can use three arguments:

- `N`: integer number of messages that are batched;
- `Reqs`: list of pids to be notified in the next batch;
- `M`: integer `N - length(Reqs)`, corresponding to the remaining number of messages to be stored before sending acknowledgments.

Question 5.2 (6 points): Define the **server event loop** function `batches`, which implements the server behavior described above and uses the signature given in the answer to the previous question.

```
  % batch complete
batches(N, Reqs, 0) ->
    [R ! ack || R <- Reqs],  % send all acks
    batches(N, [], N);       % reset state
  % batch not complete
batches(N, Reqs, M) ->
    receive                  % store Pid, decrement M
       {msg, Pid} -> batches(N, [Pid|Reqs], M - 1)
    end.
```

Question 5.3 (3 points): Define a function `init(N)` which starts a process running the server event loop function `batches` defined in the answer to the previous question with given number `N` of batched messages; `init(N)` should return the pid of the process running the server.

```
init(N) -> spawn(fun() -> batches(N, [], N) end).
```

Question 5.4 (4 points): Consider the following Erlang functions:

```
receive_all() ->
        receive _ -> io:format("OK!~n") end,
        receive_all().

test() ->
    Me = self(),
    Server = init(3),
    Server ! {msg, Me},
    Server ! {msg, Me},
    Server ! {x, Me},
    Server ! {msg, Me},
    Server ! {x, Me},
    Server ! {msg, Me},
    Server ! {msg, Me},
    Server ! {x, Me},
    receive_all().
```

After we call `test()`, and wait sufficiently long so that all sent messages are received:

a. How many times is the string `"OK!"` printed?

b. Does the call to `test()` terminate? Why or why not?

c. What is the content of the mailbox of the `Server` spawned by `init`?

d. What is the state of the server function `batches` run by process with pid `Server`?

a. `"OK!"` is printed 3 times (corresponding to the first and only complete batch).

b. It does not terminate because `receive_all()` does not terminate.

c. The `Server`'s mailbox includes all non-processed messages, namely three copies of `{x, Me}`.

d. The server function has state `3, [Me, Me], 1`, corresponding to arguments `N`, `Reqs`, and `M`.