

EXAM

Concurrent Programming TDA383/DIT390

Date: 2016-03-19 Time: 14:00–18:00 Place: Maskinhuset (M)

Responsible Michał Pałka 0707966066

Result Available no later than 2016-04-12

Aids Max 2 books and max 4 sheets of notes on A4 paper (written or printed); additionally a dictionary is allowed

Exam grade There are 6 parts (10 + 11 + 9 + 14 + 8 + 16 = 68 points); a total of at least 24 points is needed to pass the exam. The grade for the exam is determined as follows.

Chalmers Grade 3: 24–38 points, grade 4: 39–53 points, grade 5: 54–68 points

G U Godkänd: 24–53 points, Väl Godkänd: 54–68 points

Course grade To pass the course you need to pass each lab and the exam. The grade for the whole course is based on the points obtained in the exam and the labs. More specifically, the grade (exam + lab points) is determined as follows.

Chalmers Grade 3: 40–59 points, grade 4: 60–79 points, grade 5: 80–100 points

G U Godkänd: 40–79 points, Väl Godkänd: 80–100 points

Please read the following guidelines carefully:

- Please read through all questions before you start working on the answers
- Begin each part on a new sheet
- Write your anonymous code on each sheet
- Write clearly; unreadable == wrong!
- Solutions that use busy-waiting or polling will not be accepted, unless stated otherwise
- Don't forget to write comments with your code
- Multiple choice questions are awarded full points if all the correct alternatives and none of the incorrect ones are selected, and zero points otherwise
- The exact syntax is not crucial; you will not be penalized for missing for example a parenthesis or a comma

Part 1: General knowledge (10p)

Are the following Erlang functions *tail-recursive*?

(1p) 1.1. 1 `mul([], M) -> M;` (A) Yes (B) No
2 `mul([X|Xs], M) -> mul(Xs, M * X).`

(1p) 1.2. 1 `append([], Ys) -> Ys;` (A) Yes (B) No
2 `append([X|Xs], Ys) -> [X|append(Xs, Ys)].`

(1p) 1.3. 1 `loop(L) ->` (A) Yes (B) No
2 `receive`
3 `{release, Pid, X} ->`
4 `Pid ! release_reply,`
5 `io:format("release:_reply_sent~n"),`
6 `loop([X|L]);`
7 `{acquire, Pid} when L /= [] ->`
8 `[Y|Ys] = L,`
9 `Pid ! {acquire_reply, Y},`
10 `loop(Ys)`
11 `end.`

(1p) 1.4. 1 `loop(N) ->` (A) Yes (B) No
2 `receive`
3 `{incr, Pid} ->`
4 `Pid ! incr_reply,`
5 `loop(N + 1);`
6 `{read, Pid} ->`
7 `Pid ! {read_reply, N},`
8 `loop(N)`
9 `end,`
10 `io:format("should_not_be_printed~n").`

Part 2: State spaces (11p)

The algorithm below attempts to solve the critical section problem, and is built from atomic if statements (p2, q2 and p5, q5). The test of the condition following if, and the corresponding then or else action, are both carried out in one step, which the other process cannot interrupt.

```

integer S := 0

p0 loop forever
p1  non-critical section
p2  if S < 2 || S > 4 then S := 5
    else S := 1
p3  await (S != 4 && S != 1)
p4  critical section
p5  if S < 7 then S := S+1
    else S := S-5

q0 loop forever
q1  non-critical section
q2  if odd(S) then S := 7
    else S := 2
q3  await (S != 3 && S != 7)
q4  critical section
q5  if S >= 2 then S := S-2
    else S := S+4

```

Below is part of the state transition table for an abbreviated version of this program, skipping p1, p4, q1 and q4 (the critical and non-critical sections), as well as p0 and q0. For example, in line 5 of the table below p3 transitions directly to p5, skipping p4. A state transition table is a tabular version of a state diagram. The left-hand column lists the states. The middle column gives the next state if p next executes a step, and the right-hand column gives the next state if q next executes a step. In many states both p or q are free to execute the next step, and either may do so. But in some states, such as 5 below, one or other of the processes may be blocked. There are 10 states in total.

| | State = (p _i , q _i , S) | next state if p moves | next state if q moves |
|----|---|-----------------------|-----------------------|
| 1 | (p2, q2, 0) | (p3, q2, 5) | (p2, q3, 2) |
| 2 | (p3, q2, 5) | (p5, q2, 5) | (p3, q3, 7) |
| 3 | ? | ? | ? |
| 4 | ? | ? | ? |
| 5 | (p3, q3, 7) | (p5, q3, 7) | no move |
| 6 | ? | ? | ? |
| 7 | ? | ? | ? |
| 8 | ? | ? | ? |
| 9 | ? | ? | ? |
| 10 | (p5, q2, 5) | (p2, q2, 6) | (p5, q3, 7) |

Complete the state transition table (correctness of the table will not be assessed).

Are the following states reachable in the algorithm above?

(1p) 2.1. (p3, q3, 1) (A) Yes (B) No

(1p) 2.2. (p2, q2, 5) (A) Yes (B) No

(1p) 2.3. $(p_5, q_3, 5)$ (A) Yes (B) No

(1p) 2.4. $(p_2, q_5, 2)$ (A) Yes (B) No

(1p) 2.5. $(p_3, q_3, 6)$ (A) Yes (B) No

(3p) 2.6. Prove from your state transition table that the program ensures mutual exclusion.

Do the following invariants hold? The notation p_2, p_3, p_4 , etc. denotes the condition that process p is currently executing line 2, 3, 4, etc.

(1p) 2.7. $(p_2 \wedge q_2) \rightarrow (S = 5 \vee S = 7)$ (A) Yes (B) No

(1p) 2.8. $q_2 \rightarrow (S = 0 \vee S = 5)$ (A) Yes (B) No

(1p) 2.9. $(p_3 \wedge q_3) \rightarrow S = 1$ (A) Yes (B) No

Part 3: Concurrent Java I (9p)

The following code implements the producer-consumer pattern in Java using semaphores. The buffer is embedded in the consumer object Cons, and accessed by the producer using the put() method. The imports and handling of the InterruptedException have been omitted for clarity.

```
1 class ProdCon {
2     static class Prod implements Runnable {
3         Cons c;
4         public Prod(Cons cc) { c = cc; }
5         public void run () {
6             while (true) {
7                 int x = produce();
8                 c.put(x);
9                 System.out.println("put_element");
10            }
11        }
12        int produce() {...}
13    }
14    static class Cons implements Runnable {
15        Semaphore empty = new Semaphore(1);
16        Semaphore full = new Semaphore(1);
17        int slot;
18        public void put(int r) throws InterruptedException {
19            // Acquire the semaphore guarding the slot
20            empty.acquire();
21            slot = r;
22            // Release the semaphore guarding the element
23            full.release();
24        }
25        public void run () {
26            empty.acquire(); // The slot is initially empty
27            while (true) {
28                // Acquire the semaphore guarding the element
29                full.acquire();
30                int x = slot;
31                // Release the semaphore guarding the slot
32                empty.release();
33
34                System.out.println("got_element");
35                consume(x);
36            }
37        }
38        void consume(int x) {...}
39    }
```

```
40 public static void main (String[] args) {
41     Cons c = new Cons(); Prod p = new Prod(c);
42
43     Thread t1 = new Thread(c);
44     t1.start();
45     Thread t2 = new Thread(p);
46     t2.start();
47 }
48 }
```

The main() method creates one instance each of a producer and a consumer and runs them concurrently. A correct execution of the program involves the lines "put element" and "got element" being printed as the elements are produced and consumed.

However, there is a problem with the above code, as it does not behave correctly every time it is run.

- (3p) 3.1. Describe what erroneous behaviour you can observe by running the code.
- (3p) 3.2. Explain the cause for the erroneous behaviour that can be observed and present a concrete execution of the program that supports your explanation.
- (3p) 3.3. Fix the problem that you have found by modifying the ProdCon class. You only need to state the changes you want to make to the code.

Part 4: Concurrent Java II (14p)

In this assignment your job is to implement a fair binary semaphore using Java 5 monitors (the `ReentrantLock` class etc.). Below is the skeleton of the class you should implement.

Note A binary semaphore is either free or occupied and has at most one permit. The semaphore should be implemented using the following class.

```
1 class FairSemaphore {
2     // Initialize the semaphore with one permit
3     public FairSemaphore() {...}
4     // Acquire the semaphore (blocking if needed)
5     public void acquire () throws InterruptedException {...}
6     // Release the semaphore
7     public void release() {...}
8 }
```

(4p) 4.1. Explain what it means for a semaphore to be fair. Provide a concrete example, which demonstrates the difference between a fair semaphore and one that is not fair.

(6p) 4.2. Implement a fair semaphore using Java 5 monitors (using `ReentrantLock` etc.). You may assume that there will be at most 20 threads in the program using the semaphore. Also, at most 10 million semaphore operations will be performed during a single run of the program. You may assume that `release()` will never be called on a free semaphore. To make things simpler, you may ignore the fact that `InterruptedException` needs to be handled, omit exception handling altogether and skip the imports.

(2p) 4.3. Extend your semaphore implementation with the `tryAcquire()` method, which attempts to acquire the semaphore without blocking. If the semaphore is free, it is acquired and the method returns `true`, and otherwise the method returns `false` without performing any action. The method should respect fairness, that is it should not acquire the lock if another thread is already waiting for it.

```
1     // Try to acquire the semaphore without blocking.
2     // If the semaphore is free, acquire it and return true;
3     // otherwise return false.
4     public boolean tryAcquire() {...}
5 }
```

(2p) 4.4. Extend your semaphore implementation with the `tryAcquireNonFair()` method, which works in the same way as `tryAcquire()`, but does not respect fairness. The method should be able to perform *barging*, that is to acquire the semaphore just

after it has been released skipping the queue if none of the waiting threads has managed to acquire it yet.

```
1 // Try to acquire the semaphore without blocking.
2 // If the semaphore is free, acquire it and return true;
3 // otherwise return false. The method does not respect
4 // fairness.
5 public boolean tryAcquireNonFair() {...}
6 }
```

Part 5: Concurrent Erlang I (8p)

You are developing a networking module for an application. The main operation of the module is `send(H, Msg)`, which takes a handle and a message, and sends them over the network. Here is the code of the module, which implements this operation.

```
1 -module(network).
2
3 -export([start/0, send/2]).
4
5 start() -> spawn(fun () -> loop () end).
6
7 request(Pid, Data) ->
8   Ref = make_ref(),
9   Pid!{request, self(), Ref, Data},
10  receive
11    {result, Ref, Result} -> Result
12  end.
13
14 send(Pid, Msg) ->
15   request(Pid, {send, Msg}).
16
17 loop() ->
18  receive
19    {request, Pid, Ref, {send, Msg}} ->
20     net_io:transmit([Msg]),
21     Pid ! {result, Ref, ok},
22     loop()
23  end.
```

The function `start()` starts a new instance of the service and returns a handle to it. The function `send(H, Msg)` sends a message using the low-level `net_io:transmit()` function and returns `ok`. The `net_io:transmit()` function takes a list of messages to send at once. For simplicity we assume that `net_io:transmit()` does not take any additional argument to specify the end point.

- (3p) 5.1. Since sending a single message at a time has a large overhead, your team decides that calling `send()` should not send each message at once, but wait until ten messages are accumulated and send all of them with a single call to `net_io:transmit()`. Change the network module to provide the described behaviour (you only need to write the functions that need to be changed).
- (5p) 5.2. Buffering of messages comes with its own disadvantages. In the scheme implemented in the previous task a single message might be delayed for an arbitrary amount of time if requests to send further messages arrive much later. To mitigate that, the network module should be modified so that no message is kept in the

buffer for more than 100 ms. Thus, whenever the buffer contains a message that is 100 ms old all messages should be sent off without waiting for the remaining messages to fill the buffer. Implement the behaviour described above by modifying the network module. Again, you only need to state which the parts you change.

Part 6: Concurrent Erlang II (16p)

In this assignment you should implement a program controlling an elevator in Erlang. An elevator goes between two floors marked with Erlang atoms `lower` and `upper`, and has space for N people (configurable).

To go from the lower floor to the upper floor the elevator has to call the following low-level functions in a sequence: `hw:close_door()`, `hw:go_up()` and `hw:open_door()`. Similarly, going in the other direction requires calling `hw:close_door()`, `hw:go_down()` and `hw:open_door()` in this order.

The elevator module exposes the following functions to the users.

```
1 -module(elevator).
2 -export([start/1, enter/2, leave_at_dest/1]).
3
4 start(Max) -> ...
5 enter(E, Floor) -> ...
6 leave_at_dest(E) -> ...
```

The function `start(Max)` creates an instance of the elevator that can hold at most `Max` people and returns a handle to it. The elevator starts at the lower floor with its door open. The function `enter(E, Floor)` takes a handle to an elevator and one of the atoms `lower` or `upper`. The function is called when a person wants to enter the elevator on a particular floor, and should block until that person can enter the elevator. The function `leave_at_dest(E)` is called when a person in the elevator is ready to leave and blocks until the elevator has reached the destination floor and opened the door. Both `enter(E, Floor)` and `leave_at_dest(E)` should return the atom `ok` on successful completion. The elevator may assume that `leave_at_dest(E)` is called only by a person that is already in the elevator.

There is a number of persons in the system, each of which is a process that executes a program of the following form:

```
1 person(E) ->
2   timer:sleep(some_time()),
3   enter(E, lower),
4   timer:sleep(some_time()),
5   leave_at_dest(E),
6   timer:sleep(some_time()),
7   enter(E, upper),
8   timer:sleep(some_time()),
9   leave_at_dest(E),
10  person(E).
```

Where `some_time()` returns a random non-negative integer.

The elevator should use the following mode of operation. Starting from the lower floor with door open, it should let people from the lower floor in until its maximum capacity is reached. At that point no more people are allowed in, the door is closed and the elevator travels to the upper floor. After the door is open the elevator waits for the people to get

outside. And finally when it is empty, it would wait for people from the upper floor to enter it and continue in an analogous way down.

Example Below is an example trace of execution of a correct elevator implementation, with the process making each call marked. The order of lines in the trace is the order in which the calls were unblocked. For example, the call `enter(E, upper)` in line 9 might have been executed before the elevator arrived on the upper floor, but person 3 was let into the elevator only after persons 1 and 2 traveling up have left it.

```
1 E = start(2)      % main process
2 enter(E, lower)  % person 1
3 enter(E, lower)  % person 2
4 hw:close_door() % elevator
5 hw:go_up()       % elevator
6 hw:open_door()   % elevator
7 leave_at_dest(E) % person 2
8 leave_at_dest(E) % person 1
9 enter(E, upper)  % person 3
10 enter(E, upper) % person 1
11 hw:close_door() % elevator
12 hw:go_down()    % elevator
13 hw:open_door()  % elevator
14 leave_at_dest(E) % person 1
15 ...
```

- (10p) 6.1. Implement the elevator Erlang module to provide the functionality described above.
- (6p) 6.2. We would like to add one more operation to the elevator module: `enter_prio(E, Floor)`, which behaves like `enter(E, Floor)`, except that a person calling it is given priority over other people waiting for the elevator that called `enter(E, Floor)`. However, if a person that does not have priority has already entered the elevator, he or she will be allowed to stay there even if other people with priority arrive before the elevator leaves.

Appendix

Erlang builtin functions

Builtin functions (BIFs) are functions provided by the Erlang VM. Here is a reference to several of them.

register/2

`register(Name, Pid)` registers the process `Pid` under the name `Name`, which must be an atom. If there is already a process registered under that name, the function throws an exception. When the registered process terminates, it is automatically unregistered.

The registered name can be used in the send operator to send a message to a registered process (`Name ! Message`). Sending a message using a name under which no process is registered throws an exception.

Example The following example assumes that there is no processes registered as `myproc` and `myproc2` before executing the statements, and that the first created process keeps running when all other statements are executed.

```
1 1> register (myproc, spawn (fun init/0)).
2 true
3 2> register (myproc, spawn (fun init/0)).
4 ** exception error: bad argument
5     in function register/2
6     called as register(myproc,<0.42.0>)
7 3> myproc!{mymessage, 3}.
8 {mymessage, 3}
9 4> myproc2!{mymessage, 3}.
10 ** exception error: bad argument
11     in operator !/2
12     called as myproc2 ! {mymessage, 3}
```

whereis/1

`register(Name)` returns the `PID` of a registered process, or the atom `undefined` if no process is registered under this name. `unregistered`.

Example The following example assumes that there are no processes registered as `myproc` and `myproc2` before executing the statements and that the first created process is still running then the `whereis/2` calls are executed.

```
1 1> register (myproc, spawn (fun init/0)).
2 true
3 2> whereis(myproc).
4 <0.48.0>
5 3> whereis(myproc2).
6 undefined
```

is_pid/1

is_pid(Arg) returns true if its argument is a PID, and false otherwise.

Example

```
1 1> P = spawn (fun init/0).
2 <0.46.0>
3 2> is_pid(P).
4 true
5 3> is_pid(something_else).
6 false
```

Java concurrency libraries

Here is a reference of several concurrency-related classes.

Semaphore

```
1 import java.util.concurrent.Semaphore;
2
3 class Semaphore {
4     Semaphore(int permits);
5     Semaphore(int permits, boolean fair);
6
7     void acquire() throws InterruptedException;
8     void acquire(int permits) throws InterruptedException;
9     boolean tryAcquire();
10    boolean tryAcquire(int permits);
11    void release();
12    void release(int permits);
13 }
```

ReentrantLock class

```
1 import java.util.concurrent.locks.*;
2
3 class ReentrantLock {
4     ReentrantLock();
5     ReentrantLock(boolean fair);
6
7     void lock();
8     boolean tryLock();
9     void unlock();
10    Condition newCondition();
11 }
```

```
12
13 interface Condition {
14     void await() throws InterruptedException;
15     void signal();
16     void signalAll();
17 }
```

Object

The condition variable associated with each object to be used with synchronized is accessible using methods from the Object class.

```
1 class Object {
2     void wait() throws InterruptedException;
3     void notify();
4     void notifyAll();
5 }
```