SAMPLE EXAM

Concurrent Programming TDA383/DIT390

Sample exam March 2016   Time: ? – ?   Place: Johanneberg

Responsible Michał Pałka 0707966066

Result Available no later than ?-?-2016

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 (9 + 15 + 8 + 12 + 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

GU 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

GU 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 exactly the correct answers 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 (9p)

Are the following Erlang functions *tail-recursive*?

(1p) 1.1.

```
1 sum([])     -> 0;
2 sum([X|Xs]) -> X + sum(Xs).
```

(A) Yes (B) No

(1p) 1.2.

```
1 rev([], Ac)     -> Ac;
2 rev([X|Xs], Ac) -> rev(Xs, [X|Ac]).
```

(A) Yes (B) No

(1p) 1.3.

```
1 loop(N) ->
2   receive
3     {incr, Pid} ->
4       Pid ! {incr_reply, N},
5       loop(N + 1);
6     {reset, Pid} ->
7       Pid ! reset_reply,
8       loop(0)
9   end.
```

(A) Yes (B) No

(1p) 1.4.

```
 1 loop(N) ->
 2   io:format("loop␣iteration~n"),
 3   receive
 4     {add, Pid, M} ->
 5       Pid ! add_reply,
 6       loop(N + M);
 7     {read, Pid} ->
 8       Pid ! {read_reply, N},
 9       loop(N)
10   end.
```

(A) Yes (B) No

Do the following snippets of Java code perform *busy-waiting*? Assume that the variables `s1` and `free` are not referenced by any other part of the code, and that `critical_section()` and `non_critical_section()` always change the state of the system (they do something meaningful).

(1p) 1.5.

```
1  Semaphore s1;
2  boolean free = true;
3  // ...
4  // Code run by threads 1 and 2
5  while(true) {
6    non_critical_section();
7    bool crit = false;
8    do {
9      s1.acquire();
10     try {
11       crit = free;
12       free = false;
13     } finally {s1.release()}
14   } while (!crit);
15   critical_section();
16   s1.acquire();
17   try { free = true;
18   } finally {s1.release()}
19 }
```

(A) Yes (B) No

(1p) 1.6.

```
1  Semaphore s1;
2  boolean free = true;
3  // ...
4  // Code run by threads 1 and 2
5  while(true) {
6    non_critical_section();
7    bool crit = false;
8    s1.acquire();
9    try {
10     crit = free;
11     free = false;
12     critical_section();
13   } finally {s1.release()}
14 }
```

(A) Yes (B) No

(3p) 1.7. Modern versions of Java provide two variants of monitors: (a) one based on the `synchronized` keyword and the `wait()`, `notify()` and `notifyAll()` methods; and (b) one based on the `ReentrantLock` class (Java 5 monitors). List at least

three important differences (there are more) between them from the point of view of the user of these mechanisms.

Part 2: State spaces (15p)

Here is yet another algorithm to solve the critical section problem, 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 even(S) then S := 4
      else S := 5
p3   await (S != 1 && S != 5)
p4   critical section
p5   if S >= 4 then S := S-4
      else skip

q0 loop forever
q1   non-critical section
q2   if S < 4 then S := 3
      else S := 7
q3   await (S != 6 && S != 7)
q4   critical section
q5   if odd(S) then S := S-1
      else skip
```

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). 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 = (pi, qi, Svalue)	next state if p moves	next state if q moves
1	(p2, q2, 0)	(p3, q2, 4)	(p2, q3, 3)
2	(p3, q2, 4)	(p5, q2, 4)	(p3, q3, 7)
3	?	?	?
4	?	?	?
5	(p3, q3, 7)	(p5, q3, 7)	no move
6	?	?	?
7	?	?	?
8	?	?	?
9	?	?	?
10	(p2, q2, 2)	(p3, q2, 4)	(p2, q3, 3)

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. (p2, q3, 4) (A) Yes (B) No

(1p) 2.2. (p3, q5, 5) (A) Yes (B) No

(1p) 2.3. `(p3, q5, 4)` (A) Yes (B) No

(1p) 2.4. `(p5, q3, 7)` (A) Yes (B) No

(3p) 2.5. Prove from your state transition table that the program ensures mutual exclusion.

(2p) 2.6. State formally the property that the program does not deadlock.

(3p) 2.7. Prove from your state transition table that the program does not deadlock.

Do the following invariants hold? The notation p2, p3, p4, etc. denotes the condition that process p is currently executing line 2, 3, 4, etc.

(1p) 2.8. $q3 \rightarrow (S = 5 \lor S = 7)$ (A) Yes (B) No

(1p) 2.9. $(p3 \land q3) \rightarrow (S = 5 \lor S = 7)$ (A) Yes (B) No

(1p) 2.10. $(p3 \land q3) \rightarrow S = 7$ (A) Yes (B) No

Part 3: Concurrent Java I (8p)

The *Regional Development Bank* continues its growth, which also exposed a performance problem with its IT system. Currently, the bank uses the following Java class for holding its accounts.

```
1  class Accounts {
2    Lock l = new ReentrantLock();
3
4    Account[] store; // Map account id to Account object
5    // ...
6
7    // amount is always non-negative, source and target are always
8    // valid account ids
9    public boolean transfer(int source, int target, int amount) {
10     l.lock()
11     try {
12       int tmp = store[source].getBalance();
13       if (tmp < amount) return false; // Transfer failed
14       store[source].updateBalance(tmp - amount);
15       store[target].updateBalance(store[target].getBalance() + amount);
16     } finally {l.unlock() }
17     return true;
18   }
19 }
```

The `Account` class is defined as follows.

```
1  class Account {
2    int id;
3    int balance;
4    // ...
5    int getBalance () {
6      return balance;
7    }
8    void updateBalance (int newB) {
9      balance = newB;
10   }
11 }
```

A consultant was called in, and within two days he diagnosed the problem: *There is too much contention on the Accounts object, which causes the threads to wait for too long time.* Within two more days, he had a proposal on how to solve the problem. The proposed solution avoids contention by locking individual accounts instead of locking the whole table. Below are the new versions of the `Accounts` and `Account` classes.

```
1  class Accounts {
2    Account[] store; // Map account id to Account object
```

```
 3    // ...
 4
 5    // amount is always non-negative, source and target are always
 6    // valid account ids
 7    public boolean transfer(int source, int target, int amount) {
 8      store[source].l.lock()
 9      try {
10        store[target].l.lock()
11        try {
12          int tmp = store[source].getBalance();
13          if (tmp < amount) return false; // Transfer failed
14          store[source].updateBalance(tmp - amount);
15          store[target].updateBalance(store[target].getBalance()
16                                          + amount);
17        } finally {store[target].l.unlock() }
18      } finally {store[source].l.unlock() }
19      return true;
20    }
21  }
```

```
 1  class Account {
 2    // Account is used only internally, so public here is OK
 3    public Lock l = new ReentrantLock();
 4    int id;
 5    int balance;
 6    // ...
 7    int getBalance () {
 8      return balance;
 9    }
10    void updateBalance (int newB) {
11      balance = newB;
12    }
13  }
```

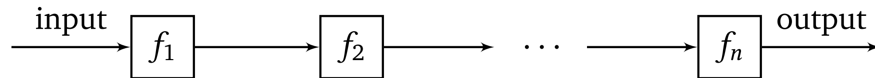
The plan is to move the changes to production next week. However, looking at the proposed update, you see a problem that can have serious consequences for the bank.

(4p) 3.1. Explain what can go wrong with the new code. For a full mark, you need to provide a concrete scenario, which demonstrates the problem. You may assume that you can populate the `store` table with any account data you wish.

(4p) 3.2. Propose how to fix the problem that you identified, and implement your proposal.

Part 4: Concurrent Java 11 (12p)

In this assignment you have to implement a pipeline of processes in Java, which transforms a stream of values by applying a number of functions in a sequence to each value.



The pipeline should be implemented using the following class.

```
1 class Pipeline {
2   // Initialize the pipeline
3   public Pipeline(Stage[] stages) {}
4   // Start the threads
5   public void run () throws InterruptedException {}
6   // Feed one more input to the pipeline
7   public void feed(T input) throws InterruptedException {}
8 }
```

The pipeline is constructed based on an array of stages (described below). The `run()` method should start a thread for each stage. The `feed()` method should feed one more input to a running pipeline, and could be called by different threads concurrently. The type `T` is an arbitrary type. The input should get processed by each of the stages in turn. If a stage is busy processing a previous input, the input should be buffered in a one-slot buffer. Thus, the `feed()` method may block. There is no way of getting the results from the pipeline. Instead, the last stage should perform a side-effect such as printing the results. Pipeline stages are specified by defining subclasses of the following class.

```
1 abstract class Stage {
2   // This class is unsynchronized
3   abstract T compute (T x);
4 }
```

Example

```
 1 class T { public int x; public T(int y) { x = y; } }
 2
 3 class Stage1 {
 4   T compute (T a) {
 5     return T(a.x + 2);
 6   }
 7 }
 8 class Stage2 {
 9   T compute (T a) {
10     return T(a.x * 3);
11   }
12 }
```

```
13 class Stage3 {
14   T compute (T a) {
15     System.out.println("Got␣a.x␣=␣" + a.x);
16     return a;
17   }
18 }
19
20 // ...
21   Stage[] stages = { new Stage1(), new Stage2(), new Stage3() };
22   Pipeline p = new Pipeline (stages);
23   p.run();
24   p.feed(new T(2));
25   p.feed(new T(4));
26   p.feed(new T(3));
27 // ...
```

The code above should print the following three lines after some time.

```
Got T.x = 12
Got T.x = 18
Got T.x = 15
```

(8p) 4.1. Your job is to implement the `Pipeline` class to provide the functionality described above.

(4p) 4.2. Additionally, implement the `void join()` method in the `Pipeline` class, which would terminate all pipeline threads after they have finished processing all the outstanding elements. The method should block until all the threads terminate.

Part 5: Concurrent Erlang 1 (8p)

Consider the following Erlang code that starts and registers a server.

```
1 start() ->
2   case whereis(myserver) of
3     undefined ->
4       Pid = spawn(myserver, init, []),
5       register(myserver, Pid),
6       {ok, Pid};
7     Pid when is_pid(Pid) ->
8       {error, already_started}
9   end.
```

The code is correct when run only by one process, but has problems when invoked concurrently.

(3p) 5.1. Explain what is the problem with the code providing a concrete example.

(5p) 5.2. Propose how to fix the problem, and implement your solution. Note that you do not have access to the `init/0` function, and cannot change it.

Part 6: Concurrent Erlang II (16p)

In this assignment you have to implement a bounded buffer in Erlang, which provides three operations.

```
1 -module(bbuffer).
2 -export([start/1, get/1, put/2]).
```

The `start(N)` operation creates a new buffer of size `N` and returns its handle. The blocking `get(Serv)` operation takes a buffer handle and gets the the oldest element from the buffer, or blocks if the buffer is empty. The blocking operation `get(Serv, X)` takes a buffer handle and an element, and inserts it into the buffer, or blocks if there is no space in the buffer.

(8p) 6.1. Implement the `bbuffer` Erlang module to provide the functionality described above.

(8p) 6.2. Int addition to the previous functionality, implement the operations `unload/1` and `release/1`. The `unload(Serv)` operation should force the buffer not to accept any `get/1` operations (they should block), but keep accepting the `put/2` operations. The `unload/1` operation should block until the buffer is empty. Furthermore, the `unload/1` operation should have precedence over any blocked `put/2` operations. The `release(Serv)` operation should put the buffer back to its regular mode.

Note that the efficiency of the solution does not affect the grade; only correctness matters.

Appendix

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
```