RE-EXAM

Concurrent Programming TDA383/DIT390

Date: 2016-08-22 Time: 14:00–18:00 Place: Maskinhuset (M)

Responsible Michał Pałka 031 772 10 79

Result Available no later than 2016-09-12

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 5 parts (8 + 16 + 16 + 14 + 14 = 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 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

Answers are shown in red.
Additional explanations, which are not part of the answers, are shown in purple.

Part 1: General knowledge (8p)

Are the following Erlang functions *tail-recursive*?

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

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

(1p) 1.3.
```
1  loop(N) ->
2    receive
3      {read, Pid} ->
4        Pid ! {read_reply, N},
5        loop(N);
6      {write, Pid, M} ->
7        Pid ! write_reply,
8        io:format("before␣loop()~n"),
9        loop(M),
10       io:format("after␣loop()~n")
11   end.
```
(A) Yes (B) No

(1p) 1.4.
```
1  loop(N) ->
2    receive
3      {add, Pid, M} ->
4        io:format("received␣add␣message~n"),
5        Pid ! add_reply,
6        loop(N + M);
7      {read, Pid} ->
8        Pid ! {read_reply, N},
9        loop(N),
10       ok
11   end.
```
(A) Yes (B) No

(2p) 1.5. Consider the following Erlang code:

```
1  child() ->
2    receive msg1 -> ok end,
3    receive X -> io:format("child␣received␣~p~n", [X]) end.
4
5  parent() ->
6    C = spawn(fun child/0),
7    C ! msg2,
8    C ! msg1,
9    C ! msg3,
10   ok.
```

What will be the result of running `parent()`?

(A) The program will not print anything.

(B) The program will print `child received msg2`.

(C) The program will print `child received msg3`.

(D) The program will print `child received msg2` or `child received msg3` depending on the particular execution.

(2p) 1.6. Consider the following Erlang code:

```
1  child() ->
2    receive start -> ok end,
3    X = receive
4      msg1 -> received1;
5      msg2 -> received2
6    end,
7    io:format("child␣received␣~p~n", [X]).
8
9  parent() ->
10   C = spawn(fun child/0),
11   C ! msg2,
12   C ! msg1,
13   C ! start,
14   ok.
```

What will be the result of running `parent()`?

(A) The program will print `child received received1`.

(B) The program will print `child received received2`.

(C) The program will not print anything.

The receive statement tries to match the first message in the inbox against all its patterns before moving to the next one.

Part 2: State spaces (16p)

The algorithm below attempts to solve the critical section problem, and is built from atomic `if` statements (p4 and q4) and atomic `await` statements (p3 and q3). 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. Similarly, the test of the condition following `await` and the corresponding action are carried in one step. The `then` actions of p4 and q4 contain an assignment and a `break` statement, whose execution terminates the execution of the innermost loop—the one starting in p2 or q2 in this case.

```
   integer T := 0

p0 loop forever                          q0 loop forever
p1   non-critical section                q1   non-critical section
p2   loop forever                        q2   loop forever
p3     await (T == 0 || T == 2)          q3     await (T == 0 || T == 2)
         T := T + 1;                              T := T + 1;
p4     if T == 3 then T := 4; break      q4     if T == 1 then T := 5; break
         else skip                                else skip
p5     T := 2                            q5     T := 0
p6   critical section                    q6   critical section
p7   T := 0                              q7   T := 2
```

Below is part of the state transition table for an abbreviated version of this program, skipping p1, p6, q1 and q6 (the critical and non-critical sections), as well as p0, q0, p2 and q2. For example, in line 3 of the table below q4 transitions directly to q7 after breaking of from the loop, skipping q6. 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 2, 3 or 10 below, one or other of the processes may be blocked. There are 10 states in total.

	State = (pi, qi, T)	next state if p moves	next state if q moves
1	(p3, q3, 0)	(p4, q3, 1)	(p3, q4, 1)
2	(p4, q3, 1)	(p5, q3, 1)	no move
3	(p3, q4, 1)	no move	(p3, q7, 5)
4	(p5, q3, 1)	(p3, q3, 2)	no move
5	(p3, q7, 5)	no move	(p3, q3, 2)
6	(p3, q3, 2)	(p4, q3, 3)	(p3, q4, 3)
7	(p4, q3, 3)	(p7, q3, 4)	no move
8	(p3, q4, 3)	no move	(p3, q5, 3)
9	(p7, q3, 4)	(p3, q3, 0)	no move
10	(p3, q5, 3)	no move	(p3, q3, 0)

Complete the state transition table (correctness of the table will not be assessed). Note: the order of rows in the table does not matter.

Are the following states reachable in the algorithm above? It is enough to check if they appear in the table

(1p) 2.1. (p7, q3, 5) (A) Yes (B) No

(1p) 2.2. (p3, q3, 3) (A) Yes (B) No

(1p) 2.3. (p3, q3, 2) (A) Yes (B) No

(1p) 2.4. (p3, q4, 3) (A) Yes (B) No

(1p) 2.5. (p3, q5, 4) (A) Yes (B) No

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

Mutual exclusion holds if no state is reachable where both p and q are in the critical section. We are using an abbreviated version of the program where the critical sections are part of the statements following them (p7 and q7).

The condtion that we are going to show is that both p and q are not in states p7 and q7 at the same time in our abbreviated program. To show the condition, it is enough to use the table to conclude that there are no reachable (p7, q7, ?) states.

Do the following invariants hold? The notation p3, p4, p5, etc. denotes the condition that process p is currently executing line 3, 4, 5, etc. It is enough to check if the invariants hold for all the states from the table.

(1p) 2.7. $(p3 \wedge q3) \rightarrow (T = 5 \vee T = 7)$ (A) Yes (B) No

(1p) 2.8. $p_7 \rightarrow T = 4$ (A) Yes (B) No

(1p) 2.9. $p_4 \rightarrow (T = 1 \vee T = 2)$ (A) Yes (B) No

(1p) 2.10. $q_3 \rightarrow (T = 0 \vee T = 2)$ (A) Yes (B) No

(4p) 2.11. Is the solution presented above free from deadlock? Present a proof that it is the case or give a counterexample.

The solution deadlocks, as it can enter an infinite loop when none of the threads will ever enter the critical section: `(p3, q3, 0)` → `(p4, q3, 1)` → `(p5, q3, 1)` → `(p3, q3, 2)` → `(p3, q4, 3)` → `(p3, q5, 3)` → `(p3, q3, 0)`. Such a livelock violates the deadlock-freedom property.

Part 3: Concurrent Java 1 (16p)

The following code merges two streams of non-negative integers by adding them. It uses an implementation of a thread-safe bounded buffer `BBuffer` (implementation omitted) that supports the standard operations `get` and `put`. In addition, `BBuffer` can also report its maximum capacity (which is set in the constructor) using the `capacity()` method and the current number of elements it holds using the `size()` method. The imports have been omitted for clarity.

```
1  import java.util.concurrent.locks.*;
2
3  class Merge {
4    static class BBuffer {
5      public BBuffer(int size) {...}
6      public void put(int t) throws InterruptedException {...}
7      public int get() throws InterruptedException {...}
8      public int capacity() {...}
9      public int size() throws InterruptedException {...}
10   }
11
12   // Two queues that are merged
13   BBuffer buf1 = new BBuffer(5);
14   BBuffer buf2 = new BBuffer(5);
15   // Total numbers of added and removed elements
16   int buf1total = 0;
17   int buf2total = 0;
18   int gettotal  = 0;
19   boolean limited = false;
20   int limit = 0;
21   Lock lock = new ReentrantLock();
22
23   public int put1(int t) throws InterruptedException {
24     lock.lock();
25     try {
26       if (limited && buf1total >= limit) return -1;
27       ++buf1total;
28       buf1.put(t);
29       return 0;
30     } finally { lock.unlock(); }
31   }
32   public int put2(int t) throws InterruptedException {
33     lock.lock();
34     try {
35       if (limited && buf2total >= limit) return -1;
36       ++buf2total;
37       buf2.put(t);
```

```
38        return 0;
39      } finally { lock.unlock(); }
40    }
41    public int get() throws InterruptedException {
42      lock.lock();
43      try {
44        if (limited && gettotal >= limit) return -1;
45        ++gettotal;
46        return buf1.get() + buf2.get();
47      } finally { lock.unlock(); }
48    }
49    public void terminate() {
50      lock.lock();
51      try {
52        if (limited) return;
53        limited = true;
54        limit = Math.max(buf1total, buf2total);
55      } finally { lock.unlock(); }
56    }
57  }
```

An object of class `Merge` holds two bounded buffers, to which elements can be added using the `put1()` and `put2()` methods. The `get()` method returns the sum of the oldest element added by `put1()` and the oldest element added by `put2()`. If at least one of the queues is empty, `get()` should block until the elemets become available. Similarly, `put1()` and `put2()` should block if there is no space left in the queues.

Furthermore, the `Merge()` class supports clean shutdown. After the `terminate()` method is called, the object allows adding elements only to the shorter queue (or to none if they are of equal lengths), and getting the number of elements that will allow emptying the queues, not more. If more `get()`, `put1()` or `put2()` calls are made, they result in `-1` being returned to signal an error.

There is a problem with the above code, as it does not behave correctly in some cases when different threads execute the `get()`, `put1()` and `put2()` methods.

(8p) 3.1. Describe what erroneous behaviour you can observe by running code that uses different threads concurrently calling `get()`, `put1()` and `put2()` of the same `Merge` object. Describe the scenario that you are considering (2p), the expected observable behaviour (3p) and the actual observed behaviour (3p).

There are many possible scenarios, which involve concurrent calls that block and cannot be unblocked. Here is one of them.

Scenario: One problem manifests itself when the following program is run. First, an object of class `Merge` is created and an element is added to it using the `put1()` method. Then, two threads are started: one executes the `get()` method, which blocks, while the other one waits some time and then executes the `put2()` method.

Expected behaviour: The correct behaviour in this situation is that `put2()` succeeds, and after that `get()` unblocks and returns the sum of the elements added by the `put1()` and `put2()` methods.

Actual behaviour: However, the implementation is incorrect, as the actual observed behaviour is that neither the call to `put2()` nor to `get()` ever terminates. Thus, both threads hang instead of continuing.

(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.

Here the explanation depends on the scenario chosen in the answer to the previous question.

The problematic part starts when the `get()` call is executed. First, the lock is aquired in line 42, then the counters are checked and updated (lines 44 amd 45). Next, the method executes two calls to the `get()` methods of the two queues (line 46) that are stored in the `Merge` object. The first call succeeds, and returns the value placed previously by the `put1` operation. The second call blocks as `buf2` is empty, which causes the call to `Merge`'s `get()` to block as well while the lock is still locked.

When the `put2()` call is executed from the other thread, it also tries to acquire the lock (line 33), which blocks as the lock is locked by the blocked `get()` invocation. Thus, both calls in the two threads block forever.

(5p) 3.3. Fix the problem that you have found by modifying the `Merge` class. You only need to state the changes you want to make to the code.

Below is the code of the modified `Merge` class that fixes the deadlocking `get()`, `put1()` and `put2()` methods. There are 12 added lines: 22–23, 30–31, 33–36, 45–46, 48 and 57–59. All other lines are unchanged. The changes add two condition variables and ensure that the calls to `get()` and `put()` of `buf1` and `buf2` never block by checking for relevant conditions before making the calls. If the conditions are not satisfied, the `Merge` methods block on one of the condition variables. The `terminate()` method does not block or influence blocking of other methods, and thus requires no changes.

The solution on the exam only needs to specify the changes to the code, without neccessarly repeating all the code.

```
1 import java.util.concurrent.locks.*;
2
3 class Merge {
4   static class BBuffer {
5     public BBuffer(int size) {...}
6     public void put(int t) throws InterruptedException {...}
7     public int get() throws InterruptedException {...}
```

```
8       public int capacity() {...}
9       public int size() throws InterruptedException {...}
10    }
11
12    // Two queues that are merged
13    BBuffer buf1 = new BBuffer(5);
14    BBuffer buf2 = new BBuffer(5);
15    // Total numbers of added and removed elements
16    int buf1total = 0;
17    int buf2total = 0;
18    int gettotal  = 0;
19    boolean limited = false;
20    int limit = 0;
21    Lock lock = new ReentrantLock();
22    Condition nonEmpty = lock.newCondition();
23    Condition nonFull = lock.newCondition();
24
25    public int put1(int t) throws InterruptedException {
26      lock.lock();
27      try {
28        if (limited && buf1total >= limit) return -1;
29        ++buf1total;
30        while (!(buf1.size() < buf1.capacity()))
31          nonFull.await();
32        buf1.put(t);
33        // signalAll() is a conservative choice here-we
34        // could use signal() instead. Same applies to
35        // other occurrences of signalAll().
36        nonEmpty.signalAll();
37        return 0;
38      } finally { lock.unlock(); }
39    }
40    public int put2(int t) throws InterruptedException {
41      lock.lock();
42      try {
43        if (limited && buf2total >= limit) return -1;
44        ++buf2total;
45        while (!(buf2.size() < buf2.capacity()))
46          nonFull.await();
47        buf2.put(t);
48        nonEmpty.signalAll();
49        return 0;
50      } finally { lock.unlock(); }
51    }
52    public int get() throws InterruptedException {
```

```
53      lock.lock();
54      try {
55        if (limited && gettotal >= limit) return -1;
56        ++gettotal;
57        while (!(buf1.size() > 0 && buf2.size() > 0))
58          nonEmpty.await();
59        nonFull.signalAll();
60        return buf1.get() + buf2.get();
61      } finally { lock.unlock(); }
62    }
63    public void terminate() {
64      lock.lock();
65      try {
66        if (limited) return;
67        limited = true;
68        limit = Math.max(buf1total, buf2total);
69      } finally { lock.unlock(); }
70    }
71  }
```

Part 4: Concurrent Java II (14p)

Consider a savings account shared by several people. Each person associated with the account may deposit or withdraw money from it. The current balance in the account is the sum of all deposits to date less the sum of all withdrawals to date. Clearly, the balance must never become negative. A person making a deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds. Furthermore, an account may be frozen by calling `freeze(true)` and unfrozen by calling `freeze(false)`. The method returns a boolean, which indicates the frozen status before the call. A deposit from or a withdrawal to a frozen account should block until the account is unfrozen.

Your task You are given the task to write a Java class that implements shared accounts. More specifically, you need to provide the following interface.
The semaphore should be implemented using the following class.

```
1 class SharedSavingsAccount {
2   public SharedSavingsAccount(long initialBalance);
3   // Both deposit() and withdraw require a non-negative argument
4   public void deposit(int amount);
5   public void withdraw(int amount);
6   public boolean freeze(boolean freezeorunfreeze);
7 }
```

Implement the `SharedSavingsAccount` class. Your solution must fulfill the following criteria:

- You must use Java.
- Your solution must support concurrent usage of the account.
- You are free to use any synchronization constructs.
- Do not care about fairness in your solution.

Note Correct implementation without the `freeze()` method can earn up to 8 points.

In the solution below, `throws InterruptedException` declarations have been added to the signatures of the `deposit()` and `withdraw()` methods. In an exam solution, `InterruptedException` may be handled in any way or ignored altogether. Import statement might be omitted from the solution as well. The presented solution uses two condition variables, but for simplicity one might be used instead.

```
1 import java.util.concurrent.locks.*;
2
3 class SharedSavingsAccount {
4   long balance;
5   boolean frozen = false;
6   Lock lock = new ReentrantLock();
7   Condition moreFunds = lock.newCondition();
8   Condition unFrozen = lock.newCondition();
```

```
 9
10    public SharedSavingsAccount(long initialBalance) {
11      balance = initialBalance;
12    }
13
14    public void deposit(int amount) throws InterruptedException {
15      lock.lock();
16      try {
17        while (frozen) unFrozen.await();
18        balance += amount;
19        moreFunds.signalAll();
20      } finally { lock.unlock(); }
21    }
22    public void withdraw(int amount) throws InterruptedException {
23      lock.lock();
24      try {
25        while (frozen || balance < amount) moreFunds.await();
26        balance -= amount;
27      } finally { lock.unlock(); }
28    }
29    public boolean freeze(boolean freezeorunfreeze) {
30      lock.lock();
31      try {
32        boolean oldFrozen = frozen;
33        frozen = freezeorunfreeze;
34        if (oldFrozen && !freezeorunfreeze) {
35          // We need to wake up potentially everyone
36          unFrozen.signalAll();
37          moreFunds.signalAll();
38        }
39        return oldFrozen;
40      } finally { lock.unlock(); }
41    }
```

Part 5: Concurrent Erlang (14p)

In this assignment you should re-implement the shared account from Part 4 in Erlang. The solution should provide the `account` module that exposes the following functions to the users.

```
1 -module(account).
2 -export([start/1, deposit/2, withdraw/2, freeze/2]).
3
4 start(Balance) -> ...
5 deposit(E, Amount) -> ...
6 withdraw(E, Amount) -> ...
7 freeze(E, FreezeOrUnfreeze) -> ...
```

The behaviour of the `deposit/2`, `withdraw/2` and `freeze/2` functions should be the same as of the respective methods from Part 4—in particular the `freeze/2` function should return a boolean. As in Part 4, the solution does not need to be fair. Efficiency is also not a concern, as long as polling or busy-waiting does not occur.

Note Correct solutions without the `freeze/2` function can earn up to 8 points.

The following solution uses guards to delay handling of some requests. While this is not an efficient solution when many requests are delayed, it is perfectly acceptable as the task does not ask for an efficient solution.

```
1  -module(account).
2  -export([start/1, deposit/2, withdraw/2, freeze/2]).
3
4  start(N) ->
5    spawn(fun () -> loop(N, false) end).
6
7  loop(N, Frozen) ->
8    receive
9      {deposit, Pid, M} when not Frozen ->
10       Pid ! deposit_reply,
11       loop(N + M, Frozen);
12     {withdraw, Pid, M} when not Frozen andalso M =< N ->
13       Pid ! withdraw_reply,
14       loop(N - M, Frozen);
15     {freeze, Pid, FreezeOrUnfreeze} ->
16       Pid ! {freeze_reply, Frozen},
17       loop(N, FreezeOrUnfreeze)
18   end.
19
20 deposit(S, M) ->
21   S ! {deposit, self(), M},
22   receive deposit_reply -> ok end.
23
```

```
24 withdraw(S, M) ->
25   S ! {withdraw, self(), M},
26   receive withdraw_reply -> ok end.
27
28 freeze(S, FreezeOrUnfreeze) ->
29   S ! {freeze, self(), FreezeOrUnfreeze},
30   receive {freeze_reply, OldFrozen} -> OldFrozen end.
```

Appendix

Elang 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> register (myproc, spawn (fun init/0)).
true
2> register (myproc, spawn (fun init/0)).
** exception error: bad argument
     in function  register/2
        called as register(myproc,<0.42.0>)
3> myproc!{mymessage, 3}.
{mymessage, 3}
4> myproc2!{mymessage, 3}.
** exception error: bad argument
     in operator  !/2
        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> register (myproc, spawn (fun init/0)).
true
2> whereis(myproc).
<0.48.0>
3> whereis(myproc2).
undefined
```

`is_pid/1`

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

Example

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

Java concurrency libraries

Here is a reference of several concurrency-related classes.

`Semaphore`

```
import java.util.concurrent.Semaphore;

class Semaphore {
  Semaphore(int permits);
  Semaphore(int permits, boolean fair);

  void acquire() throws InterruptedException;
  void acquire(int permits) throws InterruptedException;
  boolean tryAcquire();
  boolean tryAcquire(int permits);
  void release();
  void release(int permits);
}
```

`ReentrantLock` class

```
import java.util.concurrent.locks.*;

class ReentrantLock {
  ReentrantLock();
  ReentrantLock(boolean fair);

  void lock();
  boolean tryLock();
  void unlock();
  Condition newCondition();
}
```

```
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 }
```