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

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 (10p)

Are the following Erlang functions *tail-recursive*?

(1p) 1.1.

```
1 mul([], M)     -> M;
2 mul([X|Xs], M) -> mul(Xs, M * X).
```

(A) Yes (B) No

(1p) 1.2.

```
1 append([], Ys)     -> Ys;
2 append([X|Xs], Ys) -> [X|append(Xs, Ys)].
```

(A) Yes (B) No

(1p) 1.3.

```
1  loop(L) ->
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.
```

(A) Yes (B) No

(1p) 1.4.

```
1  loop(N) ->
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").
```

(A) Yes (B) No

(2p) 1.5. A formula is an invariant for a program if and only if. . .

(A) It might be true or false for the accessible states and is true for all inaccessible states

(B) It is true for all accessible and all inaccessible states

(C) It is true for all accessible states and might be true or false for the inaccessible states

(D) It is true for all accessible states and false for all inaccessible states

(E) It might be true or false for the accessible states and is false for all inaccessible states

Answer whether the following statements about monitors in Java are true.

(1p) 1.6. When using Java's `ReentrantLock` and associated condition variables, replacing any invocation of `signalAll()` with `signal()` will not compromise the correctness of the monitor.

(A) Yes (B) No

It might make the implementation incorrect, as too few threads might be woken up after `signal()` is called.

(1p) 1.7. When using Java's `ReentrantLock` and associated condition variables, replacing any invocation of `signal()` with `signalAll()` will not compromise the correctness of the monitor.

(A) Yes (B) No

More threads might be woken up as a result of calling `signalAll()`, but a correct monitor takes care of that.

(2p) 1.8. Consider a correct monitor using Java's `ReentrantLock` and condition variables. We modify it as follows.

- We replace all condition variables with a single condition variable `c0`.
- All calls to the `await()`, `signal()` and `signalAll()` methods of the removed condition variables are replaced with calls to the respective methods of `c0`.

The resulting monitor will behave in the same way as the original monitor without its correctness being compromised.

(A) Yes (B) No

The new monitor would work almost in the same way as the original one, however calling `signal()` or `signalAll()` would affect all threads waiting for all conditions as all conditions would be handled by `c0`. In particular, calling `signal()` could possibly wake up a thread waiting for any condition, and not wake up any of the threads waiting for the condition that was handled by that invocation of `signal()` in the original monitor. Therefore, some threads that should be woken up might continue waiting.

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                          q0  loop forever
p1    non-critical section                q1    non-critical section
p2    if S<2 || S>4 then S := 5           q2    if odd(S) then S := 7
         else S := 1                                else S := 2
p3    await (S != 4 && S != 1)            q3    await (S != 3 && S != 7)
p4    critical section                    q4    critical section
p5    if S < 7 then S := S+1              q5    if S >= 2 then S := S-2
       else S := S-5                              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 = (pi, qi, 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	(p2, q3, 2)	(p3, q3, 1)	(p2, q5, 2)
4	(p2, q5, 2)	(p3, q5, 1)	(p2, q2, 0)
5	(p3, q3, 7)	(p5, q3, 7)	no move
6	(p3, q3, 1)	no move	(p3, q5, 1)
7	(p2, q2, 6)	(p3, q2, 5)	(p2, q3, 2)
8	(p5, q3, 7)	(p2, q3, 2)	no move
9	(p3, q5, 1)	no move	(p3, q2, 5)
10	(p5, q2, 5)	(p2, q2, 6)	(p5, q3, 7)

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

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

(1p) 2.3. `(p5, q3, 5)` (A) Yes (B) No

(1p) 2.4. `(p2, q5, 2)` (A) Yes (B) No

(1p) 2.5. `(p3, q3, 6)` (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 (p5 and q5).

The condtion that we are going to show is that both p and q are not in states p5 and q5 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 `(p5, q5, ?)` states.

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. It is enough to check if the invariants hold for all the states from the table.

(1p) 2.7. $(p2 \wedge q2) \rightarrow (S = 5 \vee S = 7)$ (A) Yes (B) No

(1p) 2.8. $q2 \rightarrow (S = 0 \vee S = 5)$ (A) Yes (B) No

(1p) 2.9. $(p3 \wedge q3) \rightarrow S = 1$ (A) Yes (B) No

Part 3: Concurrent Java 1 (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.

There are two problems in the code above. Identifying and fixing at least one of them gives full points for all the questions.

The first problem manifests with that the execution may freeze after printing "`put element`" once. After that, nothing else will be printed and the program will hang.

The second problem is that an uninitialized value is read by the consumer, which is not visible immediately in the printout.

(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.

The first problem is triggered when one of the threads may starts with a small delay. If thread `t1` starts with a delay, then the producer will execute its `put()` operation, which will acquire the `empty` semaphore and release the `full` semaphore, leaving them with 0 and 2 permits respectively. Then, the consumer will try to acquire the `empty` semaphore and block on it. Finally, the producer will again call `put()` and also block on acquiring the `empty` semaphore, which will lead to a deadlock.

The second problem occurs when the producer thread (`t2`) starts with a delay. Then, the consumer thread (`t1`) is able to acquire both semaphores, and read an uninitialized value (`0`) from the slot. After that the execution is performed normally.

(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.

The simplest way to fix the first problem is to remove line 26, which contains a call to `empty.acquire()`, and initialise the semaphore `empty` with zero permits. In order to do the latter, we replace `new Semaphore(1)` in line 15 with `new Semaphore(0)`.

In order to fix both problems, we again remove line 26. To fix the uninitialized read we replace `new Semaphore(1)` in line 16 with `new Semaphore(0)` instead of modifying line 15.

Part 4: Concurrent Java 11 (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.

When a semaphore gets released while many threads are waiting to acquire it, a fair semaphore will let the thread that waits the longest to acquire it. A non-fair semaphore might let one of the other threads acquire it.

Suppose that a semaphore is used to guard a critical section, and the execution of the critical section takes longer time than executing the non-critical section. If there are three threads competing for the critical section, then there will be at least two threads waiting to acquire the semaphore during the moment when it is released. If the semaphore is fair, then it will always be the thread that waits the longest that manages to acquire the critical section, and as a result no thread will wait longer than two executions of the critical section. Otherwise, any thread will be allowed to aquire the semaphore, which could lead to one thread never being allowed to execute the critical section.

(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.

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

(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.

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

Example solution using counters.

Each thread that attempts to acquire the semaphore must register itself in a queue, and leave the queue only after the threads before it have left it. Each thread remembers its place in the queue using a 32-bit counter.

The counter will not wrap around, as at most 10 million operations will be ever performed on the semaphore, but the code works even if the counter may wrap around. We can show that if the counter wraps around, no two threads will have the same counter, since all live counters will be within `next_to_go` and `next` if `next_to_go` $\leqslant$ `next`, or otherwise between `next` and `Integer.MAX_VALUE` or between `Integer.MIN_VALUE` and `next_to_go`.

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

class FairSemaphore {
  // We need a fair lock here
  Lock l = new ReentrantLock (true);
  Condition c = l.newCondition();
  boolean free = true;
  // It is fine for the counters to wrap around
  // as long as we have at most 2^32 threads.
  // next - next_to_go (taken as unsigned values)
  // is the number of threads waiting on the semaphore.
  int next_to_go = 0;
  int next = 0;
```

```
14
15    // Initialize the semaphore with one permit
16    public FairSemaphore() {}
17    // Acquire the semaphore (blocking if needed)
18    public void acquire () throws InterruptedException {
19      l.lock();
20      int my_number = next++;
21      while(!(free && my_number == next_to_go)) c.await();
22      free = false;
23      ++next_to_go;
24      l.unlock();
25    }
26    // Release the semaphore
27    public void release() {
28      l.lock();
29      free = true;
30      c.signalAll();
31      l.unlock();
32    }
33    // Try to acquire the semaphore without blocking.
34    // If the semaphore is free, acquire it and return true;
35    // otherwise return false.
36    public boolean tryAcquire() {
37      l.lock();
38      boolean result;
39      // We acquire only if no other threads are waiting
40      if (free && next_to_go == next) {
41        // We don't have to enter the queue at all here
42        free = false;
43        result = true;
44      } else result = false;
45      l.unlock();
46      return result;
47    }
48    // Try to acquire the semaphore without blocking.
49    // If the semaphore is free, acquire it and return true;
50    // otherwise return false. The method does not respect
51    // fairness.
52    public boolean tryAcquireNonFair() {
53      l.lock();
54      boolean result;
55      // We try to acquire regardless of the queue
56      if (free) {
57        // We don't have to enter the queue at all here
58        free = false;
```

```
59        result = true;
60      } else result = false;
61      l.unlock();
62      return result;
63    }
64  }
```

The following solution uses a bounded queue, but is really an overcomplicated and more restricted version of the first solution. It is included here for reference.

```
1  import java.util.concurrent.locks.*;
2
3  class FairSemaphore {
4    // We need a fair lock here
5    Lock l = new ReentrantLock (true);
6    Condition c = l.newCondition();
7    boolean free = true;
8    // This implements a bounded buffer for 20 threads.
9    // (end + size - beg) % size is the number of threads
10   // waiting on the semaphore.
11   final int size = 21;
12   int[] queue = new int[size];
13   int beg = 0;
14   int end = 0;
15   // The counter will wrap around, but we
16   // will handle at most 20 threads anyway.
17   int nextid = 0;
18
19   // Initialize the semaphore with one permit
20   public FairSemaphore() {}
21   // Acquire the semaphore (blocking if needed)
22   public void acquire () throws InterruptedException {
23     l.lock();
24     int my_number = nextid++;
25     // Put my id into the queue
26     queue[end] = my_number;
27     end = (end + 1) % size;
28     while(!(free && my_number == queue[beg])) c.await();
29     free = false;
30     beg = (beg + 1) % size;
31     l.unlock();
32   }
33   // Release the semaphore
34   public void release() {
35     l.lock();
36     free = true;
```

```
37      c.signalAll();
38      l.unlock();
39    }
40    // Try to acquire the semaphore without blocking.
41    // If the semaphore is free, acquire it and return true;
42    // otherwise return false.
43    public boolean tryAcquire() {
44      l.lock();
45      boolean result;
46      // We acquire only if no other threads are waiting
47      if (free && beg == end) {
48        // We don't have to enter the queue at all here
49        free = false;
50        result = true;
51      } else result = false;
52      l.unlock();
53      return result;
54    }
55    // Try to acquire the semaphore without blocking.
56    // If the semaphore is free, acquire it and return true;
57    // otherwise return false. The method does not respect
58    // fairness.
59    public boolean tryAcquireNonFair() {
60      l.lock();
61      boolean result;
62      // We try to acquire regardless of the queue
63      if (free) {
64        // We don't have to enter the queue at all here
65        free = false;
66        result = true;
67      } else result = false;
68      l.unlock();
69      return result;
70    }
71  }
```

The last solution uses an unbounded queue implemented with a list.

```
1   import java.util.*;
2   import java.util.concurrent.locks.*;
3
4   class FairSemaphore3 {
5     Lock l = new ReentrantLock (true);
6     Condition c = l.newCondition();
7     boolean free = true;
8     // This implements an unbounded buffer
```

```
 9   List<Integer> queue = new LinkedList<Integer> ();
10   // The counter will wrap around
11   int nextid = 0;
12
13   // Initialize the semaphore with one permit
14   public FairSemaphore3() {}
15   // Acquire the semaphore (blocking if needed)
16   public void acquire () throws InterruptedException {
17     l.lock();
18     int my_number = nextid++;
19     // Put my id into the queue
20     queue.add(my_number);
21     while(!(free && my_number == queue.get(0))) c.await();
22     free = false;
23     queue.remove(0);
24     l.unlock();
25   }
26   // Release the semaphore
27   public void release() {
28     l.lock();
29     free = true;
30     c.signalAll();
31     l.unlock();
32   }
33   // Try to acquire the semaphore without blocking.
34   // If the semaphore is free, acquire it and return true;
35   // otherwise return false.
36   public boolean tryAcquire() {
37     l.lock();
38     boolean result;
39     // We acquire only if no other threads are waiting
40     if (free && queue.size() == 0) {
41       // We don't have to enter the queue at all here
42       free = false;
43       result = true;
44     } else result = false;
45     l.unlock();
46     return result;
47   }
48   // Try to acquire the semaphore without blocking.
49   // If the semaphore is free, acquire it and return true;
50   // otherwise return false. The method does not respect
51   // fairness.
52   public boolean tryAcquireNonFair() {
53     l.lock();
```

```
54      boolean result;
55      // We try to acquire regardless of the queue
56      if (free) {
57        // We don't have to enter the queue at all here
58        free = false;
59        result = true;
60      } else result = false;
61      l.unlock();
62      return result;
63    }
64  }
```

Part 5: Concurrent Erlang 1 (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 operaion.

```
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 simpicity 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 than need to be changed).

Below is an example answer.

New function `loop/1` needs to be defined, while the definition of `loop/0` needs to be changed to call `loop/1`.

```
1  loop() -> loop([]).
2
```

```
3  loop(Buffer) when length(Buffer) >= 10 ->
4    net_io:transmit(Buffer),
5    loop([]);
6  loop(Buffer) ->
7    receive
8      {request, Pid, Ref, {send, Msg}} ->
9        Pid ! {result, Ref, ok},
10       loop(Buffer ++ [Msg])
11   end.
```

(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.

The definitions of the `loop/0` and `loop/1` should be replaced with the following definitions of `loop/0` and `loop/2`

```
1  loop() ->
2    receive
3      {request, Pid, Ref, {send, Msg}} ->
4        Pid ! {result, Ref, ok},
5        Self = self(),
6        RefT = make_ref(),
7        spawn(fun () ->
8          receive after 100 -> Self!{timeout, RefT} end end),
9        loop([Msg], RefT)
10   end.
11
12 loop(Buffer, _) when length(Buffer) >= 10 ->
13   transmit(Buffer),
14   loop();
15 loop(Buffer, RefT) ->
16   receive
17     {request, Pid, Ref, {send, Msg}} ->
18       Pid ! {result, Ref, ok},
19       loop(Buffer ++ [Msg], RefT);
20     {timeout, RefT} ->
21       transmit(Buffer),
22       loop()
23   end.
```

Here `make_ref()` is used to make sure that we do not receive old timeout messages. This might be handled in some other way, but it is required to address this problem.

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.

```
request(Pid, Data) ->
  Ref = make_ref(),
  Pid!{request, self(), Ref, Data},
  receive
    {result, Ref, Result} -> Result
  end.

enter(E, Floor) ->
  request(E, {enter, Floor}).

leave_at_dest(E) ->
  request(E, leave_at_dest).

switch_floor(upper) -> lower;
switch_floor(lower) -> upper.

go_floor(upper) -> go_down();
go_floor(lower) -> go_up().
```

```
19
20 start(Max) -> spawn(fun () -> loop (lower, let_in, 0, Max) end).
21
22 loop(Floor, let_in, Max, Max) ->
23   close_door(), go_floor(Floor), open_door(),
24   loop(switch_floor(Floor), let_out, Max, Max);
25 loop(Floor, let_in, N, Max) ->
26   receive
27     {request, Pid, Ref, {enter, Floor}} ->
28       Pid ! {result, Ref, ok},
29       loop(Floor, let_in, N+1, Max)
30   end;
31 loop(Floor, let_out, 0, Max) ->
32   loop(Floor, let_in, 0, Max);
33 loop(Floor, let_out, N, Max) ->
34   receive
35     {request, Pid, Ref, leave_at_dest} ->
36       Pid ! {result, Ref, ok},
37       loop(Floor, let_out, N-1, Max)
38   end.
```

Using `make_ref()` is not required for getting the full mark.

(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.

We need to define the `enter_prio` function and modify one clause (the one starting in line 25) of `loop()`.

```
1  enter_prio(E, Floor) ->
2    request(E, {enter_prio, Floor}).
3
4  % We replace the loop() clause that starts in line 25 with this one
5  loop(Floor, let_in, N, Max) ->
6    receive
7      {request, Pid, Ref, {enter_prio, Floor}} ->
8        Pid ! {result, Ref, ok},
9        loop(Floor, let_in, N+1, Max)
10   after 0 ->
11     receive
12       {request, Pid, Ref, {enter_prio, Floor}} ->
13         Pid ! {result, Ref, ok},
```

```
14          loop(Floor, let_in, N+1, Max);
15        {request, Pid, Ref, {enter, Floor}} ->
16          Pid ! {result, Ref, ok},
17          loop(Floor, let_in, N+1, Max)
18      end
19    end;
20  % ...
```

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 }
```